fhwedel
UNIVERSITY OF APPLIED SCIENCES

Department of Computer Science

Bachelor Thesis

Vergleich eines Usecases mit Serverless Technologie gegenüber Spring Boot Technologie

am Beispiel von Instant Payments

Eingereicht am:

25. August 2021

Eingereicht von:
Silas Hoffmann
Traberweg 52
22159 Hamburg
Tel.: (040) 643 94 73
E-mail: inf103088@stud.fh-wedel.de

Referent:
Prof. Dr. Dennis Säring
Fachhochschule Wedel
Feldstraße 143
22880 Wedel
Phone: (041 03) 80 48-43
E-mail: dennis.saering@fh-wedel.de

Betreut von:
Kai Roßdeutscher
DPS Engineering GmbH
Eiffestraße 78
20537 Hamburg
Phone: (040) 25 15 41-44
E-mail: kai.rossdeutscher@dps.de

Inhaltsverzeichnis

Abbildungsverzeichnis

List of Listings

1 Einleitung ✓✓

1.1 Überblick ✓✓

Google, Spotify oder Netflix: Immer mehr Unternehmen setzen heutzutage auf Virtualisierungsplattformen. Aber warum sind Technologien wie zum Beispiel *Docker* als Containerisierungsplattform oder *Docker Swarm* als Orchestrator, so beliebt? In der folgenden Thesis werde ich zunächst erläutern welche Technik hinter dem Docker-Ökosystem steckt, um anschließend einen Überblick über die verfügbaren Technologien geben und diverse Eigenarten näher erläutern (was bedeutet es serverless zu deployen etc.). Es wird außerdem einen detaillierten Vergleich zweier moderner Technologien hinsichtlich der Eignung für diese Art von Plattform erarbeitet.

1.2 Motivation ✓✓

Die Idee und die Umsetzung der Arbeit erfolgte in Zusammenarbeit mit der DPS Engineering Gmbh. Hierbei handelt es sich um ein europaweit tätiges Software- und Consultingunternehmen für die Finanz- und Retailbranche. Das Leistungsportfolio umfasst Softwareprodukte und IT-Services für die bekannten Vertreter der deutschen Kreditwirtschaft und des Handels [1]. Die mir zugeteilte Abteilung verwaltet und erweitert eine Plattform zur Abwicklung von Echtzeitüberweisungen. Um diese in Zukunft noch effizienter zu gestalten, möchte sich das Unternehmen von den altbewerten Java-Enterprise Technologien hin zu neuartigen Cloud-Technologien orientieren. Der besondere Fokus hierbei liegt auf der ressourcenschonenderen Arbeitsweise dieser Technologien. Man möchte in Zukunft darauf Verzichten stets alle verfügbaren Ressourcen direkt zu verwalten und sich auf das automatische Skalieren dieser Systeme verlassen können. So ist es beispielsweise möglich, dynamisch neue Instanzen benötigter Komponenten zu erzeugen, als sich wie bisher auf ein starres System zu verlassen, welches fortdauernd mit der gleichen Arbeitslast arbeitet. Dies erspart dem Unternehmen nicht nur Kapital, sondern führt ebenfalls zu einer besseren Klimabilanz (Stichwort „*new green economy*“). Ein Beispiel wäre hier die Abwicklung von Zahlungsverkehr mit Hilfe von Echtzeitüberweisungen[2], hierbei gilt es hinsichtlich schwankenden Zahlungsverkehrs über einen gewissen Zeitraum das laufende System dem Umständen angepasst zu skalieren. Des Weiteren verändert dieser Technologiezweig die Art des produktiven Deployments. Hierbei möchte sich das Unternehmen ebenfalls von den herkömmlichen Application Servern verabschieden, welche im Vergleich einen relativ hohen Wartungsaufwand mit sich bringen. Dabei soll der Übergang zu den Cloud-Technologien erst mittels einer hauseigenen Lösung erfolgen und im Nachgang auf einen Cloud-Provider wie Amazon (AWS) oder Microsof (Azure) gesetzt werden. Die behandelten Themen dieser Thesis beziehen sich vorallem auf eine beispielhafte Konfiguration einer hausinternen Lösung, um zu analysieren, inwiefern sich dies überhaupt rentiert.

[1] Linkedin: https://de.linkedin.com/company/dps_group
[2] Erklärung, siehe Abschnitt 4.2.1

Neben diesen vor allem an das Kapital der Firma gebundenen Gründen steht auch die Wartbarkeit der implementierten Lösungen im Fokus. Mittels lose gekoppelter Elemente verschwimmt die Grenze zwischen Operation- und Entwicklungsabteilung. Projekte erleben weniger Handoffs zwischen Teams, ein Entwickler betreut ein Projekt bis in die Deploymentphase, was vorher in dem Maße nicht möglich gewesen ist. Abhängig von Abstraktion der Cloud Technologie ermöglicht die lose Kopplung der Komponenten eine geordnetere Arbeitsweise, die eine verbesserte Codequalität mit sich bringen kann.

Ein Fokus, welchen ich mir in dieser Thesis im Detail anschauen möchte, betrifft das Start-up-Verhalten einer Anwendung in genau solch einer virtualisierten Umgebung. Denn unter den genannten Effizienzgesichtspunkten möchte man heutzutage auch nötige Ressourcen vor allem auf Anfrage verwenden und nicht mehr rund um die Uhr laufen lassen, selbst wenn dies zu einem gegebenen Zeitpunkt eigentlich gar nicht nötig wäre. Um das Prinzip der Ressourcennutzung auf Anfrage etwas anschaulicher zu gestalten, werde ich eine vereinfachte Kopie einer realen Anwendung vom Unternehmen nachbauen und hinsichtlich der Startzeiten von Containern untersuchen. Der Prototyp beinhaltet mehrere verschiedene Komponenten zur Abarbeitung der Logik, um im Nachhinein auf die Performanz hinsichtlich der genannten Effiziensgesichtspunkte zu veranschaulichen und zwischen den verwendeten Technologien zu vergleichen.

2

Zielsetzung ✓✓

Nachdem im letzten Kapitel ein kurzer Einblick in die Motivation für die Portierung alter JEE Technologie in neuere IT-Infrastruktur beschrieben wurde, wird im Folgenden darauf eingegangen, welcher wesentlichen Problemstellung sich die DPS gegenüber konfrontiert sieht und wie ich versuchen werde dies zu untersuchen.

2.0.1 Begriffserklärung ✓✓

Cloudtechnologie Der Begriff „*Cloud-Infrastruktur*“ wird oft auch unter dem Namen „*Infrastructure-as-a-Service (IaaS)*“ vermarktet. Er bezieht sich auf das Abstrahieren der Unternehmensinfrastruktur von konkreter Hardware. Es wird generell in drei verschiedene Kategorien unterschieden [BA18, Seite 54]:

1. *Private Cloud:* hauseigene Hardware wird virtualisiert in eigenen Rechenzentren betrieben. Hierbei werden die Server bespielsweise als Virtuelle Maschine eingerichtet, anstatt auf einer „*Baremetal-Lösung*“ zu basieren, bei der keinerlei Abstrahierung verbaut wurde. Die Grenze zur traditionellen Infrastruktur wird hierbei dadurch gezogen, wie diese Infrastruktur verwaltet wird. Wenn dies durch API-Schnittstellen oder SDKs wie zum Beispiel VMware oder vSphere geschieht, spricht man von der *Cloud*.
2. *Public Cloud:* Bei der privaten Cloud ist das Unternehmen allerdings noch selbst für die Wartung und Instandhaltung der tatsächlich involvierten Hardware zuständig. Bei der Public-Cloud wird dies an externe Dienstleister wie zum Beispiel Amazon (AWS) oder Microsoft (Azure) ausgelagert. Hierbei kommt insbesondere der Begriff *IaaS* zum tragen, da diese Dienstleistung als ein Service betrachtet wird, welcher vom Benutzer nur noch verwendet wird, dessen Ausführung allerdings völlig irrelevant ist.
3. *Hybrid Cloud:* dieser Begriff beschreibt eine Mischung der beiden Varianten.

Komponentenstack Dieser Begriff beschreibt eine Anwendung, die sich aus einer Ansammlung von Komponenten zusammensetzen. Diese einzelnen Komponenten laufen typischerweise jeweils in einem Container, der wiederum von einer Containerisierungsplattform verwaltet wird.

Orchestrierung Dieser Begriff bezieht sich auf das Verwalten von Komponenten eines Systems. Hierbei ist der komplette Lebenszyklus einer Anwendung zu betrachten. Es wird sowohl die Initialisierung, die Ressourcennutzung zur Laufzeit als auch das kontrollierte Herunterfahren der Komponente betrachtet. Obwohl dieser Begriff oftmals im Kontext einer containerisierten Anwendung beziehungsweise eines Komponentenstacks verwendet wird, trifft er ebenfalls auf andere Bereiche. Beispielsweise verwaltet / orchestriert ein IoC Container ebenfalls die betreffenden System-Komponenten.

Backend-Technologie Serverseitige Komponenten basieren auf bestimmten Technologien. Dies können einzelne Programmiersprachen sein, in der Regel sind jedoch unterschiedliche Frameworks gemeint, die verwendet werden, können um die geforderte Business-Logik zu implementieren.

Serverless Functions Serverless Umgebungen bilden eine weitere Abstraktionsstufe zu IaaS Plattformen und werden auch als „*Function-as-a-Service*“ Technologien bezeichnet. Genau wie schon bei IaaS Plattformen gibt es eine Aufteilung zwischen der tatsächlichen Infrastruktur und der virtualisierten Umgebung. Zusätzlich wird jedoch die Funktionalität des automatischen Verwalten von benötigten Instanzen ermöglicht, hierbei werden also Instanzen je nach Bedarf hoch oder heruntergefahren [BA18, Seite 70 ff.].

2.0.2 Kernziele ✓✓

Da das Unternehmen im Banking Bereich tätig ist, stellt eine hohe Verfügbarkeit der Komponenten eines der wesentlichen Kernziele dar. Auf die neuen Technologien bezogen, bedeutet dies, dass es zu ermitteln gilt, wie viel Zeit ein Skalierungsprozess in einem orchestrierten Komponentenstack im Detail braucht. Dazu müssen diverse Metriken erhoben werden, welche im Kapitel 4.1 beschrieben werden. Wichtig an dieser Stelle, es soll vor allem untersucht werden, inwiefern eine Java-Anwendung als cloudfähig bezeichnet werden kann. Denn wenn die Initialisierungsphase zu viel Zeit in Anspruch nimmt, werden Timeouts riskiert, was es unter allen Umständen zu vermeiden gilt. Eine Alternative versprechen serverless Technologien, welche sich vorallem durch ihre minimale Startzeit für diesen Usecase qualifizieren.

Zusammengefassung:

- Cloudfähigkeit zweier moderner Technologien ermitteln
- Dabei Verfügbarkeit sowie Skalierungsdauer prüfen

2.1 Lösungsweg ✓✓

Um festzustellen, wie gut die jeweils betrachtete Backend-Technologie im Endeffekt für den Betrieb in einer containerbasierten Anwendung tatsächlich geeignet ist, wird ein fiktiver Workflow mit Hilfe eines Komponentenstacks implementiert, der in der Lage ist, verschiedene Lastszenarien abarbeiten zu können. Hierbei ist es dem Benutzer möglich, testweise zur Laufzeit zwischen den verwendeten Backend-Technologien zu wechseln, um einen entsprechenden Vergleich anstellen zu können. Es werden diverse Skripte bereitgestellt, die es dem Benutzer erlauben, beliebig komplexe Lasten zu simulieren. Um eine Auswertung auch grafisch aufarbeiten zu können, wird der Stack ebenfalls mit Komponenten zur Visualisierung ausgestattet. Diese Visualisierungs-Elemente greifen dabei nicht in den Ablauf des restlichen Stacks ein, sondern kommunizieren lediglich mit bereits bestehende Schnittstellen um einen möglichst ungestörten Ablauf und damit unverfälschte Datensätze zu garantieren.

Der wesentliche Usecase des Systems besteht in der Verarbeitung von generierten Payments, welche nach mehreren fiktiven Arbeitsschritten verschiedenste Datensätze generieren. Der genaue Gegenstand der Bearbeitung ist allerdings eher nebensächlich, da er so gut wie keinen Einfluss auf die für diesen Usecase interessanten Metriken besitzt.

3

Ist-Analyse ✓✓

Nachdem im letzten Kapitel ein kurzer Überblick über die bestehende Problemstellung, sowie den geplanten Lösungsweg gegeben wurde, folgt ein Einblick in die bestehenden Technologien um die Vorteile modernerer Technologien zu beleuchten.

3.1 JBoss (Microprofile) ✓✓

Die aktuelle Verarbeitung von Payments innerhalb der Anwendung läuft in Produktion auf vier Instanzen des kommerziellen Applikation-Servers „*JBoss*“. Im Development wird hierbei eine Open-Source Variante namens „*Wildfly*“ verwendet. In diesen Application Servern werden .war Dateien deployed, welche den ausführbaren Code der Anwendung beinhalten. Der Application Server bietet dem Backend der Anwendung eine Laufzeitumgebung, in der die Anwendung ausgeführt werden kann. JBoss bietet nun standardisierte Schnittstellen nach dem jeweils festgelegten Java-Enterprise-Standard, um zum Beispiel die Kommunikation mit der Außenwelt zu ermöglichen oder der Anwendung eine Persistenzschicht zur Verfügung zu stellen. Außerdem laufen die Instanzen im *Microprofile* Modus, der es möglich macht, dass bestimmte Konfigurationsparameter ausgelagert werden. So können Applikationen auf unterschiedlichen Systemen deployed werden, ohne komplett neu gebaut zu werden (siehe [mic]). Um eine gleichmäßige Aufteilung der Last zu gewährleisten, teilt ein sogenannter „*Load Balancer*“ die eingehenden Nachrichten den entsprechenden Instanzen zu (siehe Abbildung 3.1). Jede der vorhandenen Instanzen besitzt eine minimale sowie maximale Anzahl an parallel ausführbaren Prozessen. Diese Angaben werden auch „*max. / min. Poolsize*“ genannt. Eine minimale Poolsize muss gegeben sein, um sicherzustellen, dass eine gewisse Grundlast, falls nötig, sofort bearbeitet werden kann, daher darf diese Anzahl auch nicht Null betragen. Die maximale Poolsize stellt sicher, dass es zu keiner Überlastung des Systems kommt. Wenn eine

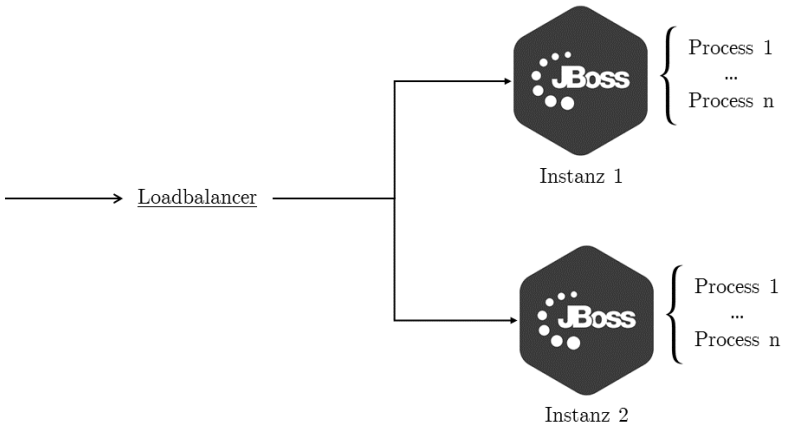


Abbildung 3.1: JBoss Systemaufbau

Instanz bereits mit der maximale Anzahl an Prozessen arbeitet, wird dies dem Loadbalancer signalisiert und welcher wiederum der entsprechenden Komponente in diesem Zeitraum keine weiteren Nachrichten mehr zuteilt. Um zu gewährleisten, dass die Nachrichten nicht verloren gehen, werden sie in eine Warteschlange („*Request Queue*“) geschrieben, welche lediglich dazu gedacht ist, den Overhead abzuspeichern. Wie die Daten im Detail verarbeitet werden, ist für die weitere Betrachtung irrelevant und wird daher nicht weiter erläutert. Eine vereinfachte Implementierung wird im späteren Prototypen dennoch vorhanden sein. Die dazugehörige Beschreibung befindet sich in Abschnitt 4.2.1.

3.2 Probleme ✓✓

Bezüglich der Systemarchitektur einer klassischen Java-Enterprise Anwendung handelt es sich in der Regel um eine monolithische Struktur. Die ausführbare Anwendung wird mit Hilfe von einer einzigen Codebase entwickelt, gebaut und deployed. Dem gegenüber steht die *service-basiert* Designstruktur, dabei handelt es sich um eine mittels Containern ermöglichten Umgebung. Weshalb monolithische Systeme den heutigen Anforderungen in manchen Aspekten nicht mehr genügen und inwiefern eine service-basierte Struktur hierbei Abhilfe schafft wird im folgenden näher erläutert [BA18, Seite 42 ff.].

3.2.1 Skalierte Entwicklung ✓✓

Wenn viele Entwickler an derselben Anwendung arbeiten, muss jeder einzelne zumindest ein grundlegendes Verständnis der gesamten Codebase besitzen. Außerdem kommt es während der Entwicklungszeit zwangsweise zu einer Reihe von Mergekonflikten. Mit den richtigen Designprinzipien bezüglich der losen Kopplung der einzelnen Komponenten sowie der Modularisierung des Systems, sollte dies zwar eigentlich umgangen werden können, in der Realität entwickelt sich ein solches System allerdings ständig weiter. Da das System diese Prinzipien allerdings nicht explizit erzwingt, werden diese Prinzipien häufig mit der Zeit vernachlässigt. Funktionalität aus dem Modul zu extrahieren und in eigenständige Services auszulagern führt hierbei zu einem effizienteren Workflow. Insbesondere hinsichtlich der Arbeitsaufteilung ermöglicht der service-basierte Aufbau die Implementierung klarer Schnittstellen, wodurch viele Entwickler parallel arbeiten können.

3.2.2 Unabhängiges Deployment ✓✓

Eine monolithische Applikation wird mit Hilfe eines einzelnen Artifacts deployed. Dies stellt insbesondere dann ein Problem dar, wenn neue Funktionalität implementiert wird oder einzelne Teile der Applikation häufiger verändert werden als andere. Bei allen genannten Szenarien muss stets die gesamte Applikation neu deployed werden obwohl die Veränderungen nur einzelne Komponenten der Anwendung betreffen. Bei einem stark gekoppelten System muss jede neu hinzugefügte Funktionalität verstärkt getestet werden, da es möglicherweise Abhängigkeiten zwischen Komponenten gibt, die auf den ersten Blick als solche eventuell gar nicht erkennbar sind. Der Service-basierte Ansatz bietet hier Abhilfe, da es möglich ist ausschließlich dedizierte Teile des Systems (neu) zu deployen beziehungsweise nur noch diejenigen Komponenten testen zu müssen, welche überhaupt verändert wurden, ohne dabei Risiken einzugehen.

3.2.3 Skalierung in Produktion ✓✓

Eine Skalierung ist mit monolithischen Strukturen nur dadurch möglich, den ausführbare Code auf zusätzlichen Servern zu deployen, dies wird auch *horizontale Skalierung genannt.* Jede dieser Kopien nutzt die gleiche Resourcenanzahl, was es zu einem ineffizienten Design macht, da sie sich nicht dynamisch der gegebenen Last anpassen. Tatsächlich ist ein es jedoch ein viel größeres Problem, wenn ein bereits aufgeteiltes System an seine Kapazitätsgrenze stößt, denn Java-Anwendungen besitzen eine relativ lange Initialisierungsphase. Um den heutigen Anforderungen der dynamischen Skalierung gerecht zu werden, werden Resourcen auf Abruf gebraucht. Die Systeme sollen so schnell wie möglich verfügbar sein. Außerdem ist es bei monolithischen Systemen ausgeschlossen nur die betreffenden Komponenten des Systems zu skalieren, die durch eine erhöhte Last überhaupt betroffen sind.

4

Vorgehensmodell

Nachdem im letzten Kapitel der aktuelle Aufbau der Payment-Anwendungen, sowie die durch die monolithische Struktur hervorgebrachten Probleme, erläutert wurden, wird im Folgenden auf die Untersuchung des Problemfokuses der Container-Startzeit eingegangen.

4.1 Anforderungen an Daten zur Messung des Startup-Verhaltens von Containern ✓

Der Fokus des Projekts liegt auf der Ermittlung der Cloudfähigkeit der unterschiedlichen System-Komponenten. Um dies festzustellen, muss im Vorwege eine Klassifizierung der Anforderungen stattfinden. Anhand dieser Klassifizierung werden die relevanten Messdaten beziehungsweise Metriken ermittelt. Die internationale Organisation für Normung (kurz *ISO*) verfasste für die generelle Klassifizierung von qualitativen Anforderungen an Softwareprodukte bereits einen ausführlichen Kriterienkatalog, welcher im Folgenden als Grundlage für die beschriebene Auswahl verwendet wird (siehe ISO/IEC 25010:2011 [iso11], fig. 4.1).

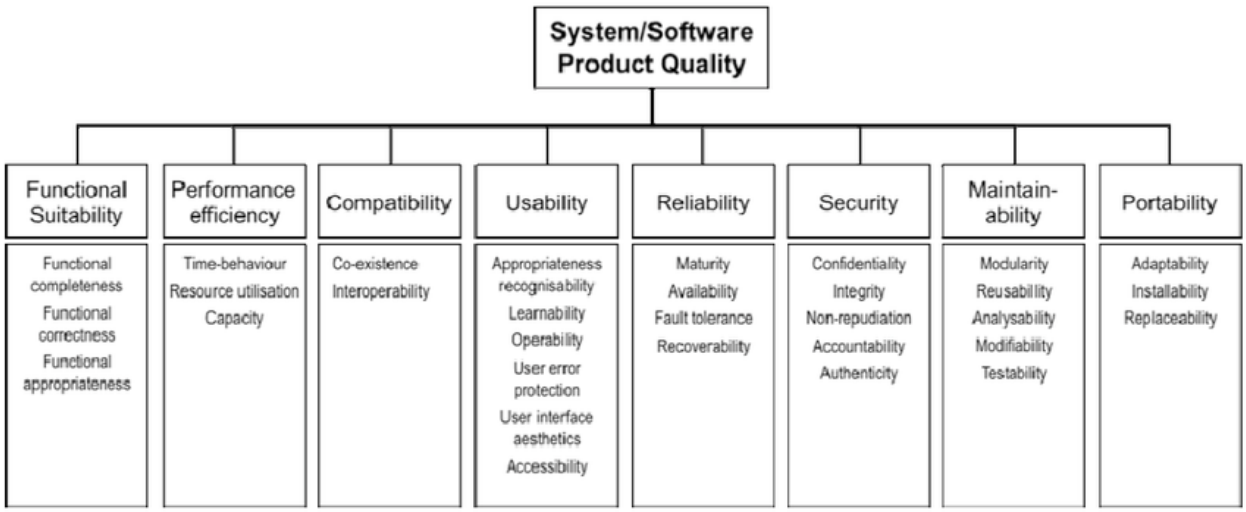


Abbildung 4.1: ISO 25010

Die ISO-Norm 25010 besteht aus acht Hauptkategorien und 31 Unterkriterien, wobei lediglich die erste Hauptkategorie der „*Funktionalität*“ dabei als *funktionale Qualitätseigenschaft* (engl. „*functional requirement*“) auftritt. Die restlichen Hauptkriterien lassen sich der Kategorie der *Non-Functional requirements* zuordnen. Diese Unterscheidung dient als eine erste Abstraktionsschicht. Functional requirements beschreiben hierbei die Korrektheit des Produktes [BA18, Seite 335]. „*Functional requirements specifies a function that a system or system component must be able to*

perform“ [Eid05, Seite 19]. So lässt sich feststellen ob die Applikation den inhaltlichen Ansprüchen, sowohl aus technischer als auch aus Unternehmenssicht, genügen. Hierbei wird auf eine Vielzahl unterschiedlicher Tests zurückgegriffen, welche sich auf breit gefächerte Abstraktionstiefen etc. beziehen lassen (siehe 4.2). Je nach Teilbereich lassen sich hierbei einzelne Gebiete automatisiert oder mittels anderweitiger Werkzeuge testen. Diese Tests sind in einem System in Produktion unabdingbar, für die Messdatenerhebung zur Beurteilung der Cloudfähigkeit können diese allerdings vernachlässigt werden.

Zur Eingrenzung der Anforderungen an die Messdaten sind primär die *Non-Functional Requirements* von Bedeutung. Sie beschreiben nicht *was* vom System geleistet werden muss, sondern *wie* dies geschehen soll, deshalt werden die Kriterien unter anderem auch *system-quality attributes* genannt [BA18, Seite 335]. Im Folgenden werden die für das Projekt relevanten Kriterien zusammengefasst dargestellt.

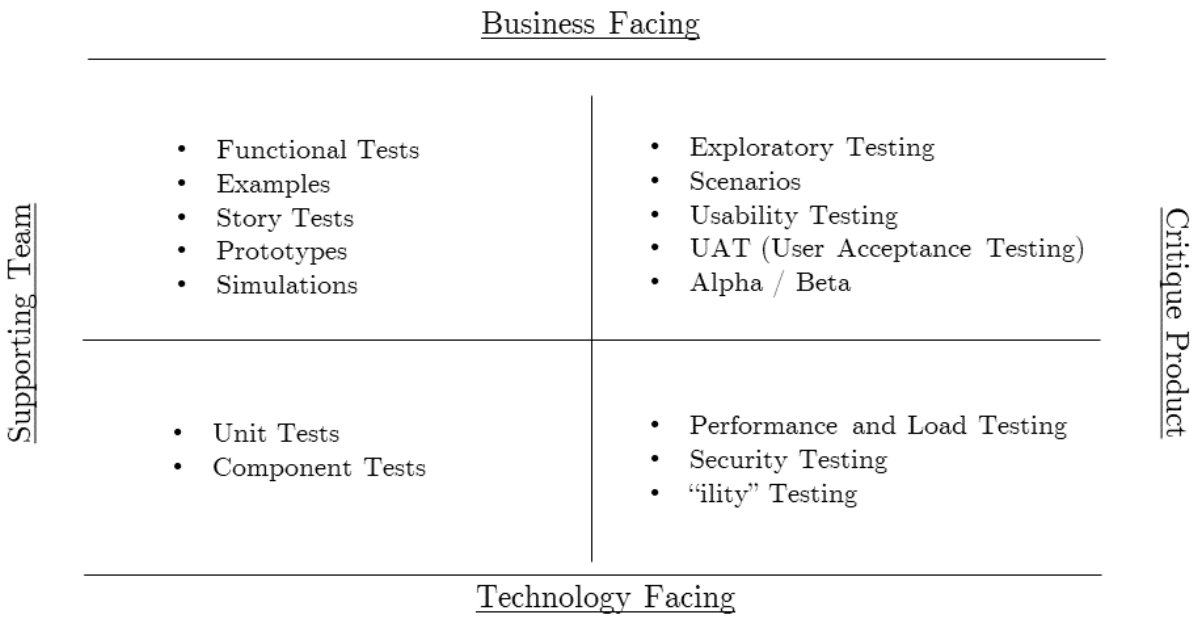


Abbildung 4.2: Agile Testing Quadrants

Leistungsfähigkeit ✓

„Performance of a product or an app defines how a product/app is performing or behaving as compared to its expected behavior“ [nfr]. Die Leistungsfähigkeit einer Anwendung setzt sich aus den Punkten *Zeitverhalten*, *Ressourcennutzung* sowie *Kapazität* zusammen. Das Zeitverhalten wird vor allem durch die Reaktionszeit der Applikation geprägt. Es wird gemessen, wie schnell auf Benutzereingaben eingegangen werden kann. Eng verwand ist dabei auch der Begriff der *Latenzzeit*, da dieser beschreibt, wie lange die Beantwortung einer Anfrage in der Praxis letztendlich dauert. Die Reaktionszeit dient hierbei als ein Threshold der nicht überschritten werden sollte. Typischerweise wird die Reaktionszeit pro Client festgelegt, es ist jedoch auch möglich, dies für eine Menge von mehreren Clients festzulegen. Ein Beispiel wäre, dass ein bestimmter Anteil der Interaktionen eine maximale Reaktionszeit von n Sekunden festgelegt wird. Aber auch das zweite Unterkriterium der *Resourcennutzung* darf nicht vernachlässigt werden, so dürfen beispielsweise bestimmte Kommunikationsschnittstellen nicht überlastet werden und diverse Connectionspools nicht überlaufen, da es sonst zu Timeouts kommen könnte. Die Kapazität beschreibt zum Beispiel die Anzahl von möglichen parallelen Anfragen oder wie viele Nachrichten übermittelt beziehungsweise vom System gespeichert werden können.

Ein weiterer Unterpunkt, welchen man der Leistungsfähigkeit zuschreiben kann, ist der des *Durchsatzes*. Dieser Begriff wird in der ISO Norm zwar nicht erwähnt, stellt hierbei jedoch eine weiteres Kriterium dar, gerade im Hinblick auf den Anwendungsfall der Banking-Anwendung. Die verarbeiteten Daten können sich hierbei auch auf verschiedenes beziehen. Hiermit kann zum Beispiel die zeitliche Aufteilung von Anfragen an eine API sowie deren Rückmeldung gemeint sein. Oder es ist die Rede einer bestimmten Anzahl von Nachrichten bezüglich eines Brokers, die durch das System gereicht werden. Es kann sich aber auch ganz banal auf die Anzahl der Datenbankanfragen etc. beziehen. Diese Art der Performanz wird in der Regel mittels spezieller Tools, wie zum Beispiel „*Gatling*“ oder „*JMeter*“ in sogenannten *Performance* oder *Load Tests* validiert. Alles in allem stellt die Leistungsfähigkeit mit ihren Subkategorien eine der wohl wichtigsten Kriterien dar.

Skalierbarkeit ✓

Ein weiteres wichtiges Kriterium, das jedoch nur indirekt in der ISO-Norm 25010 Norm auftritt, ist die Skalierbarkeit des Systems[1]. Bei der Beurteilung kommt gilt es in erster Linie zwischen dem Skalieren der *Hardware* sowie der *Software* zu unterscheiden [nfr]. Die Hardwareskalierung könnte man grob dem Unterkriterium „*Ressourcennutzung*“ zuordnen. Denn in einer modernen containerisierten Cloud-Umgebung soll es möglich sein, die einzelnen Komponenten nicht nur auf einem Gerät beliebig zu skalieren, sondern, falls die aktuelle Hardware an ihre Kapaziätsgrenzen stößt, auch zusätzliche Hardwarekomponenten hinzuziehen können. Inwiefern dies automatisiert oder manuell geschieht, lässt sich lediglich fallabhängig bestimmen, es sollte jedoch evaluiert werden, ob dies generell ermöglicht wird. Metriken zum Ermitteln der Skalierungszeitpunkte könnten zum Beispiel die Anzahl der parallel arbeiten Benutzer der Applikation (engl. „*concurrent users*“), die Latenzzeit oder die CPU Auslastung der beteiligten Maschinen sein. Relevant ist jedoch ebenfalls, dass eine Applikation nicht nur von aktiv eingehenden Ereignissen skaliert wird, sondern auch ein Zeitverhalten aufweist, wodurch eine selbstständige zeitabhängige Skalierung erfolgen kann. Dies macht den Skalierungsprozess robuster, da die Komponente dadurch etwas von seiner Umwelt abstrahiert werden kann (welche in der Regel durchaus fehlerbehaftet sein kann).

Bei der Hardwareskalierung gibt es zwei wesentliche Arten, die *vertikale* und die *horizontale Skalierung*. Der Begriff der *vertikalen Skalierung* (engl. „*scale up*“ Skalierung) beschreibt das Modifizieren der servereigenen Ressourcen. Dazu können zum Beispiel die Anzahl der CPU-Kerne oder der verfügbare Speicher gehören. Dieses Verfahren wird in monolithischen Strukturen mit älteren Systemen verwendet, die nicht darauf ausgelegt sind, sich auf mehrere Server zu verteilen. Dagegen steht der Begriff der *horizontalen Skalierung* (engl. „*scale out*“ Skalierung), welcher gerade dieses Auslagern bestimmter Komponenten oder Prozessinstanzen auf neue Hardware beschreibt. Letzterer Ansatz wird in der Cloud-Technologie verfolgt und nicht nur auf unternehmensinternen Systemen, sondern auch großen *public cloud* Dienstleistern wie zum Beispiel Amazon (AWS) oder Microsoft (Azure), verwendet.

Dennoch muss die Software auf eine Skalierung der Hardware ausgelegt sein. Manche Probleme lassen sich mit mehr Rechenperformance nicht lösen. Wenn beispielsweise die Übertragungszeiten bestimmter Kommunikationsschnittstellen einen Flaschenhals darstellt, ändert eine Skalierung hieran nichts. Alter Legacy-Code könnte ebenfalls Performanceeinbußen mit sich ziehen, ohne ein angebrachtes Session-Management wird das Skalieren der Hardware dieses Problem nicht lösen können. Diese Grundfunktionalitäten werden heutzutage jedoch von fast allen Frameworks standardmäßig mitgeliefert, dennoch gilt es dies im Vorwege zu validieren.

[1] siehe „*resource utilisatzion*“ der Hauptkategorie „*performance effeciency*“

Kompatabilität ✓

Dieses Hauptkriterium beschreibt, wie ausgeprägt die nebenläufige Verarbeitung des Systems ist. Ist es möglich, dass mehrere Komponenten gleichzeitig arbeiten können, ohne sich gegenseitig zu behindern? Gerade in einer containerisierten Umgebung ist dies zu ermitteln. Der Unterbegriff *Ko-Existenz* definiert dieses Verhalten. Der zweite Unterbegriff der „*Interoperabilität*“ grenzt hierbei die Kommunikation von der Nebenläufigkeit ab und beschreibt inwiefern die verschiedenen Komponenten in der Lage sind untereinander Daten auszutauschen. Wie genau dieser Datenaustausch aussieht, kann sich von System zu System unterscheiden. Zusammenfassend lässt sich sagen, dass diese Kriterien in einer containerbasierten Anwendung zwar gelten müssen, sie allerdings dennoch derartig abstrakt gehalten sind, dass es schwieriger ist hier messbare Metriken zu generieren. Entweder das System arbeitet hinsichtlich dieser Kriterien fehlerfrei oder es ist fehlerbehaftet, einen messbaren Mittelweg gibt es nicht direkt.

Zuverlässigkeit ✓

Ein weiteres Hauptkriterium der ISO-Norm 25010 bezieht sich auf die *Zuverlässigkeit* eines Systems. Es beschreibt die garantierte Laufzeit einer Applikation [nfr]. Dies kann auf verschiedene Weisen festgehalten werden. Eine typische Angabe wäre zum Beispiel der Anteil der Tage im Jahr, Tage pro Monat, oder Stunden am Tag die ein Service ansprechbar ist. Allgemein lässt sich diese Angabe als Metrik der *Verfügbarkeit* charakterisieren. In einer containerisierten Umgebung kommt es immer wieder zu Ausfällen einzelner Systeme, was es bei diesem Punkt besonders zu beachten gilt. Das Gesamtsystem soll hiervon jedoch unberührt bleiben. Dieses Prinzip wird in heutigen Systemen auch „*Design for failure*“ genannt [BA18, Seite 267].

Das Hauptkriterium setzt sich unter anderem aus dem Unterkriterium der *Reife* zusammen. Hierbei sind diejenigen Anforderungen gemeint, die in einer normalen Prozessabarbeitungsphase ohne besondere Ereignisse gegeben sein müssen. Die Anforderungen können sich zum Beispiel auf die Anzahl der zu verarbeitenden Nachrichten oder der Komplexität der zu bewältigenden Aufgaben zur gängigen Betriebszeit handeln. Außerdem wird eine gewisse *Fehlertoleranz* vom System erwartet. Um dies zu testen, genügt es bereits dokumentierte, allerdings Fehlerbehaftete, Benutzereingaben in einer Testumgebung zu tätigen und das Systemverhalten zu analysieren. Eine standardisierte Metrik gibt es auch hier nicht. Das letzte Unterkriterium der *Zuverlässigkeit* stellt die *Wiederherstellbarkeit* nach einem beschriebenen Fehlerfall dar, falls es doch einmal zu einem Ausfall einzelner Komponenten kommen sollte. Generell gilt auch dieses Hauptkriterium gibt Aufschluss über die Eignung gewisser technologischer Ansätze zum Arbeiten als Cloudkomponente, dennoch beschreibt sie die relevanten Umstände in manchen Teilen zu abstrakt, als das es Gegenstand dieser Thesis sein könnte.

Irrelevante Themenfelder ✓

Weitere Hauptkriterien der ISO-Norm betreffen die *Nutzbarkeit* sowie die *Sicherheit* der Anwendung. Da der Fokus des Projekts aber darauf liegt die Skalierbarkeit und Performance des Systems zu testen, können diese Punkte jedoch vernachlässigt werden. Die Themengebiete der *Wartbarkeit* sowie der *Übertragbarkeit* sind in einer containerisierten Plattform zwar ebenfalls bedeutsam, wurden wegen ihrer schwierigen Messbarkeit allerdings im Rahmen dieser Thesis verworfen.

Zusammenfassung ✓

Im letzten Abschnitt wurde auf die verschiedenen Kriterien der ISO-Norm 25010 eingegangen sowie einige weitere relevante Kriterien hinsichtlich der nicht funktionalen Anforderungen hinzugefügt. Hierbei wurde bezüglich der Anwendung der Kriterien auf den zu entwickelnden Prototypen bereits eine minimale Vorauswahl getroffen, wobei erst im nächsten Abschnitt eine letztendliche Festlegung der betrachteten Kriterien sowie deren genau Erfassung erörtert wird. Im Folgenden werden die wesentlichen Kriterien noch einmal zusammengefasst.

- Lediglich *Nicht funktionale Anforderungskriterien* für dieses Projekt von Bedeutung
- Leistungsfähigkeit
  - Zeitverhalten (Reaktions- / Latenzzeit)
  - Ermittlung von Bottlenecks
  - Leistungsfähigkeit (Durchsatz / Latenzzeit)
- Skalierbarkeit
  - Evaluierung ob automatisiert oder manuell gesteuert werden kann
  - Reaktion auf externe sowie interne Ereignisse
  - Ermitteln welche Skalierungsart angebracht (*scale out* gegenüber *scale up*)
  - Ermitteln ob Skalierung überhaupt zielführend
  - Zeitverhalten relevant
- Kompatabilität
  - Ko-Existenz (Nebenläufigkeit überhaupt möglich)
  - Interoperabilität (Kommunikation zwischen parallelen Komponenten)
- Zuverlässigkeit
  - Garantierte Laufzeit
  - *Design for failure* Prinzip erfüllt?
  - Wiederherstellbarkeit
- Irrelevante sowie nicht betrachtete Themenfelder
  - Irrelevant
    - Nutzbarkeit
    - Sicherheit
  - out of scope
    - Wartbarkeit
    - Übertragbarkeit

4.2 Anforderungen an Prototypen ✓

Bei diesem Projekt soll sowohl eine Orchestrierungsplattformen evaluiert als auch ein entsprechendes Framework hinsichtlich der Eignung im Kontext der Kernziele der Thesis verglichen werden (siehe Abschnitt *Kernziele* 2.0.2).

Hinsichtlich der Skalierung soll es beispielsweise möglich sein, anhand festgelegter Regeln eine automatische *horizontale* Skalierung bestimmter Komponenten vorzunehmen. Dieses Verhalten soll sowohl in hinterlegten Datensätzen als sowie mithilfe eines eigens dafür eingerichteten Dashboards nachvollziehbar dargestellt werden. Auf welche Technologien sowohl bei der Skalierung als auch der Darstellung beziehungsweise bei der Berechnung der Metriken zurückgegriffen wird, wird vom Arbeitgeber nicht vorgegeben. Lediglich hinsichtlich der Komponenten zur Verarbeitung der Businesslogik wurde eine Vorauswahl getroffen.

4.2.1 Festlegung fiktiver Workflow ✓

Der Prototyp soll eine bereits bestehende Applikation nachbilden, welche die Zahlungsabwicklung sogenannter Echtzeitüberweisungen durchführt. „*Die Echtzeitüberweisung (englisch Instant Payment) ist eine Zahlungsart, bei der Guthaben innerhalb weniger Sekunden dem Empfänger final gutgeschrieben werden. Inzwischen gibt es in zahlreichen Ländern weltweit Echtzeitzahlungssysteme. Im einheitlichen Euro-Zahlungsverkehrsraum, dem sogenannten Sepa-Raum, gibt es seit 21. November 2017 einen entsprechenden Standard: Die Sepa Instant Payments (SCTInst) sind 24 Stunden an 365 Tagen im Jahr verfügbar. Die Gutschrift muss in maximal 10 Sekunden erfolgen, faktisch werden die meisten Transaktionen allerdings in weniger als 3 Sekunden durchgeführt*“ [ins].

Da der Fokus auf der Untersuchung der Start-up-Zeit der Komponenten liegt, wird lediglich eine minimale beispielhafte Implementierung erfolgen, welche die Arbeitsschritte der eigentlichen Applikation vereinfacht darstellen soll. Es soll im Rahmen dieser Arbeit eher als ein *proof of concept* gelten. Allerdings werden im System konkrete Nachrichten im XML Format vermittelt, welche einer XSD-Spezifikation folgen wie sie im realen Umfeld ebenfalls genutzt wird (siehe Auszug 4.1).

```xml
<?xml version="1.0" encoding="UTF-8"?>
<Document xmlns="urn:iso:std:iso:20022:tech:xsd:pain.001.001.09">

  ...

  <DbtrAcct>
    <Id>
        <IBAN>AT331200000696200104</IBAN>
    </Id>
  </DbtrAcct>
  <Cdtr>
    <Nm>DE NAME</Nm>
    <PstlAdr>
        <Ctry>DE</Ctry>
        <AdrLine>Point Courrier 201</AdrLine>
        <AdrLine>IT 91191 Gif sur Yvette</AdrLine>
    </PstlAdr>
  </Cdtr>

  ...

</Document>
```

Listing 4.1: Payment Format

Sobald eine neue Nachricht eingetroffen ist, sollen drei Arbeitsschritte ausgeführt werden:

1. Es soll geprüft werden, ob das eingegangene XML der in den Metainformationen angegebenen XSD-Spezifikation folgt oder nicht. Wenn dies nicht der Fall sein sollte, wird die Nachricht zwar acknowledged, sodass sie aus der Eingangsqueue im Message Broker enfernt wird, allerdings bei der weiteren Verarbeitung ignoriert wird.
2. Falls es sich um valides XML-Format handelt, wird ein Feld aus dem XML-Inhalt ausgelesen.
3. In einem letzten Schritt wird das extrahierte Element in eine Datenbank geschrieben damit auch eine Persistenz-Operation in die Verarbeitungszeit einfließt.

4.2.2 Artefaktbasierte Technologie ✓

Wie bereits angedeutet, wird die aktuell laufende Java-Enterprise Applikation mit dem moderneren Spring Boot Framework ersetzt. Spring Boot bietet im Kern sehr ähnliche Funktion und kann in gewisser Hinsicht als eine modernere Version des JEE Standards angesehen werden. Der Austausch ist vor allem hinsichtlich der zu gewinnenden Messwerte / Ergebnisse sinnvoll um einen aktuelleren Referenzpunkt bezüglich der service-basierten Struktur zu erlagen.

4.2.3 Skriptbasierte Technologie ✓

Um bezüglich des Spring Boot Frameworks einen sinnvollen Vergleichspunkt anzusetzen, wird dem eine Skript-Technologie gegenübergestellt, da diese im *Cloud Native* Bereich ebenfalls sehr häufig verwendet werden. „*Eine native Cloud-Anwendung (NCA) ist ein Programm, das speziell für eine Cloud-Computing-Architektur entwickelt wurde. NCA nutzen die Vorteile von Cloud-Computing-Frameworks, die aus lose gekoppelten Cloud-Services zusammengesetzt sind.*“ [def]. Mit dieser doch abstrakt gehaltenen Definition, sind hinsichtlich der verarbeitenden Komponenten in erster Linie die modernen Skript-Frameworks gemeint. Hierbei existieren mehrere Alternativen, wo es seitens des Arbeitgebers ebenfalls keinerlei genaue Vorgaben hinsichtlich des spezifischen Frameworks beziehungsweise der zu nutzenden Technologie gab, um allerdings ein möglichst repräsentatives Bild zu schaffen, wurde hierbei auf das Node.js Backend-Framework ausgewählt, da dies als eines der am weitest verbreiteten Frameworks in diesem Bereich gilt. Sowohl die Node.js als auch Spring Boot Komponenten sollen jeweils die beschriebenen Arbeitsschritte implementieren.

4.3 Anfoderungen an Containerplattform / Orchestrierungsplattform ✓

Unter dem Begriff des „emphContainers“ wird in der Entwicklercommunity meist ein „emphDocker-Container“ verstanden. Dies ist allerdings nicht zwangsweise korrekt. Allgemein beschreibt dieser Begriff in der Softwareentwicklung ein abstraktes Konzept auf Betriebssystem-Level zur Virtualisierung oder Prozessisolation. Es gibt zahlreiche Implementierungen wie beispielsweise CoreOS's „*rkt*“ oder Microsoft's „*Hyper-V*“. Docker stellt hierbei lediglich eine weitere Implementierung dieser Technologie dar [**?**, Seite 63 ff.]. Dennoch hat sich Docker gegenüber anderen Implementierungen vor allem über die einfache Benutzerführung sowie die zentralisierten Mechanismen zum *Teilen*, *pullen* und *pushen* durchgesetzt. Die generelle Funktionsweise sowie Vorteile von Docker wurden bereits relativ ausführlich erläutert. Da die Docker Technologie von allen vorgestellten Implementierungen am weitesten verbreitet ist, soll diese auch im Prototypen Verwendung finden.

Eine Containerplattform zeichnet sich allerdings neben der genutzten Implentierung über weitere Komponenten aus. Eine der Wichtigsten stellt hierbei der Orchestrator dar. „*This component*

is responsible for starting, stopping, and managing the container processes. This technology is often referred to as container infrastructure as a service (CIaaS), and in the Docker ecostystem this is typically provided by Docker Swarm or Kubernetes“ [BA18, Seite 64 ff.]. Hierbei wird auch die Portablität sowie die Reproduzierbarkeit eines Systems als eine wesentliche Anforderung vorausgesetzt [doc, Kapitel Orchestration]. Da Kubernetes im Unternehmen bereits Anwendung findet, wurde sich hierbei auf Docker Swarm festgelegt, um ein Demo-Projekt auszuarbeiten, an dem sich in Zukunft andere Projekte orientieren können. Swarm grenzt sich gegenüber anderen Technologien vor allem durch den nativen Support innerhalb des Docker Ökosystems ab. Man benötigt keine weiteren externen Tools, sofern eine Docker Installation vorliegt [Sop16, Seite 10 ff.]. Desweiteren ist es möglich Images in einem skalierten System zu deployen ohne weitere Anpassungen am Image selbst vornehmen zu müssen. Diese Images können außerdem in Clustern organisiert werden, welche direkt über ein alternatives Tool namens „*Docker Compose*“ konfiguriert werden können.

„*Docker Compose allows the declarative specification of applications/services and associated dependencies and data stores. This pattern does allow for the flexible execution of a collection of dependent services on a local development machine*“ [BA18, Seite 173]. Um diese Spezifikation auch in einer produktiven Umgebung auszuführen, wird auf einen Orchestrator wie zum Beispiel *Docker Swarm* zurückgegriffen. Das Swarm Tool wird hierbei massiv in Produktions-Umgebungen eingesetzt, da es in der Lage ist Tausende von Containern zeitgleich zu verwalten und über einen dezentralen Discovery Service anzusteuern.

Da es im Cluster vermehrt zur Ausführung mehrerer Instanzen des gleichen Images kommen kann, ist es nicht mehr möglich einen Service mit einer spezifischen IP-Adresse zu versehen. Bei untereinander abhängigen Services braucht es einen Kommunikationsmechanismus zwischeneinander. Dieser wird durch einen „*Discovery Service*“ organisiert. Ein solcher Service ist neben dem Vermitteln von Anfragen auch in der Lage Konfigurationsänderungen der einzelnen Services im System zu organisieren sowie die Verfügbarkeit der jeweiligen Instanzen zu prüfen [**?**, Seite 45 ff.].

Zusammenfassung ✓

- Wesentliche Anforderungen Container
  - Portabilität
  - Reproduzierbarkeit
- Wesentliche Anforderungen Orchestrator
  - Automatisierte Wartung
    - Container Updates
    - Austausch fehlerhafter Container
  - ermöglicht Skalierung bestimmter Services

4.4 Anforderungen an Lasttest ✓

Es soll möglich sein, das System in einem einheitlichen Format mit Nachrichten zu versorgen. Hierfür benötigt man eine Benutzerschnittstelle, die es ermöglicht dem Benutzer möglichst aussagekräftige Befehle ohne allzu viel Konfigurationsaufwand zur Verfügung zu stellen. Diese Befehle müssen anschließend intern wiederum in detaillierte Nachrichten umgewandelt werden, welche durch das System gereicht werden können. Hierfür gibt es die Möglichkeit einer Benutzeroberfläche im Browser

oder der Festlegung eines Formats im Body einer Http-Anfrage an eine REST-Api. Eine Skalierung auf mehrere Systeme sollte zwar einmal testweise erfolgen, für die Generierung der Metriken ist dies allerdings eher hinderlich, da es die Ergebnisse verfälschen könnte. Diese Verfälschung könnte auftreten, da es dem Design der Orchestrierungsplattform geschuldet ist, dass der Anwender wenig Einfluss auf die Ressourcenzuteilung hat. Hierbei kann es passieren, dass eine leistungsschwächere Maschine deutlich weniger Anfragen zugewiesen bekommt. Dies könnte man zwar dadurch umgehen, dass man zwei exakt gleichbestückte Server für den Lastentest verwendet. Hierbei würden im Nachhinein allerdings im besten Fall die gleichen Ergebnisse erlangt werden, wie bei Testlauf auf einer einzelnen Maschine.

4.5 Anforderungen Visualisierung und Monitoring zur Unterstützung der Auswertung ✓

Es muss möglich sein, Messwerte automatisch vom System generieren zu lassen. Dazu müssen die Datensätze persistent hinterlegt werden, um auch im Nachhinein nachvollziehen zu können wie die Metriken entstanden sind. Außerdem muss eine grafische Aufarbeitung erfolgen. Diese Visualisierung soll in Echtzeit oder einem zeitlich festgelegten Intervall aktualisiert werden können. Bezüglich des genauen Werkzeugs werden keinerlei Vorgaben gegeben, um dem Benutzer jedoch jeglichen Installationsaufwand zu ersparen wäre eine webbasierte Darstellung von Vorteil.

5 Problemlösung

5.1 Implementierung Prototyp

Nachdem im letzten Abschnitt eine Festlegung auf spezifischen Kriterien zur Erhebung von Messwerten vorgenommen wurde, folgt eine detaillierte Beschreibung der Implementierung des zu erstellenden Prototypen. Der komplette Quellcode sowie alle zugehörigen Konfigurationsdateien befinden sich in einem Github-Repository unter folgender Url `https://github.com/derMacon/stack-scale-benchmark` .

5.1.1 Schichtenmodell ✓

Der Arbeitsfluss der Anwendung wurde in Abbildung 5.1 visuell dargestellt. Die unterschiedlichen Komponenten wurden dabei in vier verschiedene Schichten eingeteilt:

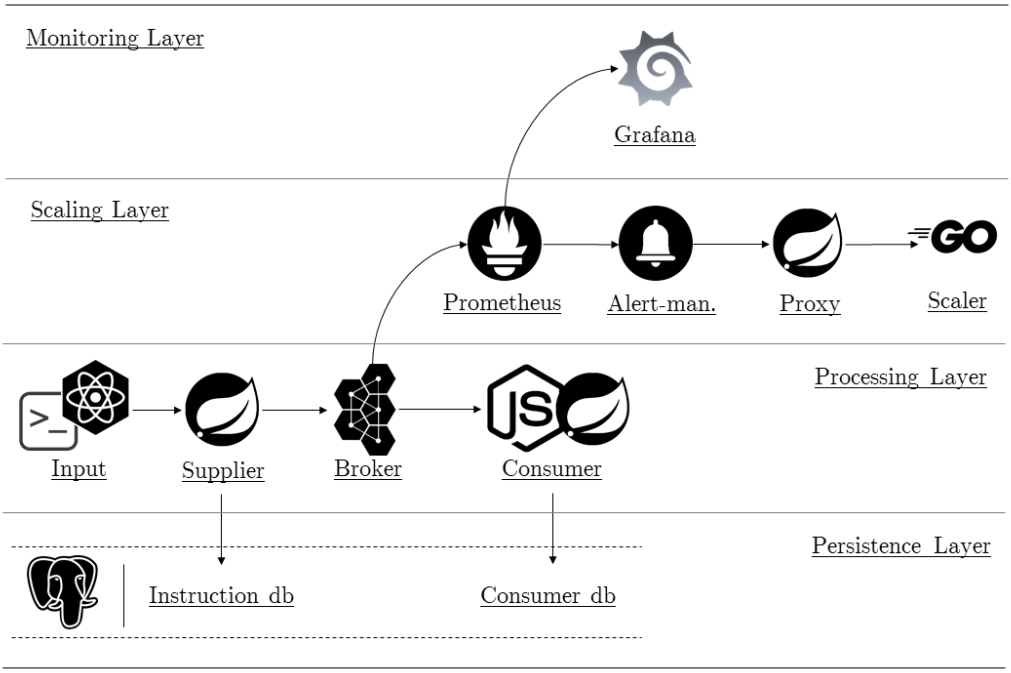


Abbildung 5.1: Komponenten-Stack im Überblick

- Persistenzschicht (engl. *persistence layer*): Beinhaltet sämtliche Komponenten, die für das Abspeichern gegebener Datensätze in dazugehörige Datenbanken zuständig sind. Um Nebenläufigkeit zu ermöglichen wird hier ebenfalls auf Schnittstellen über einen Message-Broker zurückgegriffen. Eine Manipulation der Daten findet auf dieser Ebene nicht statt.

- Verarbeitungsschicht (engl. *processing layer*): Beinhaltet sämtliche Komponenten, die für die direkte Verarbeitung der Business-Logik zuständig sind. Die Kommunikation zwischen den Komponenten findet über eine REST-Schnittstelle sowie einen Message-Broker statt. Der Message-Broker stellt hierbei vor allem eine nebenläufige Verarbeitung der konsumierenden Komponenten sicher. Allerdings bietet dieser ebenfalls die Schnittstelle zur Skalierungsschicht (engl. *scaling layer*).
- Skalierungsschicht (engl. *scaling layer*): Beinhaltet sämtliche Komponenten zum Skalieren der konsumierenden Komponenten. Hierbei wird auf eine universelle Schnittstelle des Message Broker zurückgegriffen um entsprechende Metriken abzugreifen, die für die Evaluierung hinterlegter Regeln zur Skalierung verwendet werden. Ansonsten wird in Komponenten dieser Schicht das Zeitverhalten des Initialisierungsprozesses der konsumierenden Komponenten überwacht und an Schnittstellen der Persistenzschicht weitergeleitet.
- Monitoring Ebene: Diese Schicht besteht aus einer einzigen Komponente, deren Aufgabe es ist erhaltene Daten visuell darzustellen. Um einen zeitlichen Überblick zu geben, werden die Daten intern vom Werkzeug gespeichert. Da es sich um lokal verwaltete Datensätze handelt, die für den Rest der Applikation keinerlei Bedeutung darstellen, wurde verzichtet eine Lösung zu finden, in der diese ebenfalls in einer Komponente der Persistenzschicht abzuspeichern.

5.1.2 Komponenten im Überblick ✓

Es folgt eine kurze Zusammenfassung der Funktionalität sowie der Anforderungen an die jeweilien Komponenten.

Input ✓

Um es dem Benutzer zu ermöglichen, gezielte Messwerte zu erfassen, wird eine REST-Schnittstelle vom System bereitgestellt. Es ist zwar möglich, dass der Benutzer selbstständig Anfragen für diese Schnittstelle generiert und absendet, beabsichtigt ist allerdings, dass der Benutzer auf vordefinierte Skripte oder eine entsprechende Benutzeroberfläche zurückgreift. Die Benutzeroberfläche ist sehr funktional gehalten, sodass es zwar möglich ist hierüber Anfragen an das System zu stellen, dennoch empfiehlt sich gerade für komplexere Anfrageszenarien der Gebrauch der bereitgestellten Bash-Skripte. Bezüglich der Skripte gibt es ebenfalls diverse Abstraktionsschichten. So ist es zum Beispiel möglich, mithilfe einer definierten Grammatik Anforderungen zu definieren, die sich beliebig kombinieren lassen. Hierzu wurden mehrere vordefinierte Dateien angelegt, deren Inhalt über ein entsprechendes Skript an das Backend geschickt werden können. Allerdings gibt es weitere Skripte, die auf diesem Prozess aufsetzen und somit eine Abstraktionsschicht aufbauen, sodass sich der Endbenutzer hiermit nicht auseinandersetzen muss.

Supplier ✓

Diese Komponente ist in der Lage, die vereinfachten Anfragen des Benutzers zu interpretieren und in Nachrichten umzuwandeln, die vom System verarbeitet werden können. Hierbei werden bereits an dieser Stelle diverse Informationen an die ursprüngliche Nachricht angeheftet, um im späteren Verlauf entsprechende Metriken zu berechnen. Außerdem erfolgt eine erste Kommunikation mit der Persistenzschicht, in der die übersetzten Benutzeranfragen abgespeichert werden. Hierbei wird direkt auf die Datenbank zugegriffen, da im System lediglich eine Singleton-Instanz des *Suppliers* vorhanden ist. Zwar unterstützt die verwendete Postgres-Datenbank mehrere Klienten zur gleichen Zeit, nativ ist diese Anzahl jedoch begrenzt und muss angemessen konfiguriert werden. Des Weiteren

stellt die Supplier-Komponente zwei Modi hinsichtlich der Geschwindigkeit, in der Nachrichten an den Broker übermittelt werden. So ist es zum Beispiel möglich, Nachrichten über einen gewissen Zeitraum hinweg abzuschicken oder aber eine Transaktion zu bilden, in der alle auf einmal geschickt werden.

Broker ✓

Ein Message-Broker stellt die Funktionalität bereit, Nachrichten über ein gegebenes Protokoll an mehre Konsumenten zu vermitteln. Hierzu wurde eine Implementierung von Apache namens „*Active MQ*[1]“ gewählt. Diese besitzt zwei Betriebsmodi, einmal das Arbeiten mit einer *Topic* sowie mit einer *Queue*. Eine Topic stellt ein *Publisher-Subscriber-Muster* bereit, in dem alle eingehenden Nachrichten an alle registrierten Konsumenten verschickt werden [SBD11, Seite 33 ff.]. Demgegenüber steht eine *Queue* (Warteschlange), die eingehende Nachrichten lediglich an einen einzelnen Konsumenten übermittelt, wobei der Broker hierbei als eine Art Load Balancer[2] agiert. Bevor die Nachrichten jedoch aus einer der beiden Datenstrukturen entfernt werden, muss der betreffende Konsument eine *Acknowledgement-Nachricht* an den Broker schicken, um zu signalisieren, dass die Nachricht nicht nur angenommen, sondern auch korrekt verarbeitet werden konnte. Es gilt ebenfalls noch hervorzuheben, dass die eingeschriebenen Konsumenten bei neuen Nachrichten stets benachrichtigt werden und es clientseitig keine Logik braucht, um zum Beispiel event- oder intervallbasiert eine Abfrage an den Broker zu steuern. Da diese Komponente ebenfalls den Einstiegspunkt für das Monitoring System *Prometheus* (siehe Abschnitt 5.1.2) darstellt, benötigt der Broker eine Schnittstelle über die er diesem Tool Messdaten zur Verfügung stellen kann. Die von ActiveMq bereitgestellt Schnittstelle nennt sich „*Java Management Extension API (JMX)*“ und ist die Standard-API zur Verwaltung von Java Applikationen [SBD11, Seite 331 ff.].

Consumer ✓

Die konsumierenden Komponenten können beliebig skaliert werden und implementieren die in Abschnitt 4.2.1 definierten Arbeitsschritte. Hierbei wird auf diverse Libraries zurückgegriffen, wobei die restliche Logik eher schlicht gehalten wurde. Die Komponenten kommunizieren lediglich über den Message-Broker mit dem Nachrichten-Supplier und über einen weiteren Nachrichten-Broker mit der Persistenzschicht um die extrahierten Elemente abzulegen.

Prometheus ✓

„*Prometheus is an open source systems monitoring and alerting toolkit*“ [Bra18, Seite 400]. Es ist möglich über eine definierte Anfragensprache Daten Dritter zu verarbeiten. Diese Daten können über ein einfaches Textformat von den Komponenten ausgegeben werden. Es ist möglich dieses Textformat händisch zu schreiben, allerdings wird in der Praxis vermehrt auf Libraries, die auf den Client zugeschnitten wurden, gesetzt. Prometheus ist unter der Apache 2.0 Lizenz veröffentlicht [3], und wurde primär in der Programmiersprache Go implementiert. Das intervallbasierte Anfragen (engl. „*scrapen*“) der Metriken wird von der Prometheus-Komponente selbst durchgeführt, die zu überwachenden Komponenten müssen sich selbst nicht darum kümmern Daten an Prometheus zu übermitteln.

checken ob Figur am Ende immer noch nic unter die folgende Absatz auf Seite passt, gg aendern

[1] https://github.com/apache/activemq
[2] Erklärung *Load Balancer* siehe Abschnitt 3.1
[3] https://github.com/prometheus/prometheus

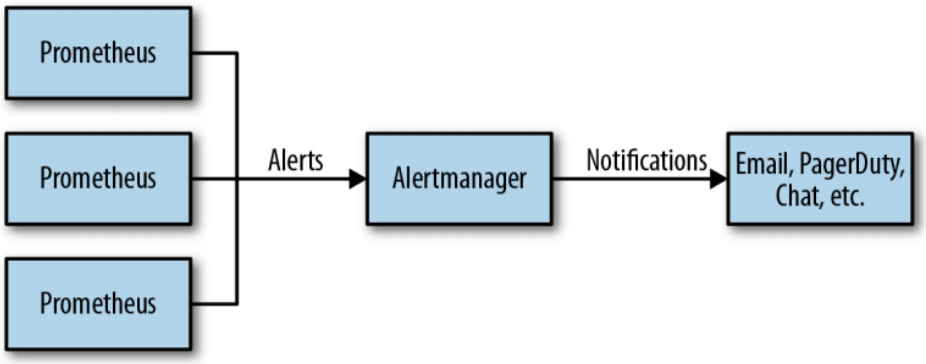


Abbildung 5.2: Alert Manager - Übersicht [Bra18, Seite 291]

Alert Manager ✓

„*Alerting is one of the components of monitoring, allowing you to notify a human when there is a problem*“ [Bra18, Seite 291]. Prometheus bietet hierbei die Möglichkeit mithilfe der funktionalen Anfragesprache *PromQl* diverse Bedingungen zu definieren unter denen dies geschehen soll. Da es in einer produktiven Containerumgebung durchaus möglich ist, dass mehrere Prometheus-Instanzen parallel arbeiten, wurde das Benachrichtigen in eine weitere Komponente (den *Alert Manger*) ausgelagert. Dieser synchronisiert, sammelt und gruppiert die Alerts der verschiedenen Prometheus-Instanzen und sendet Benachrichtigungen an definierte Nachrichtenkanäle, die sich beispielsweise aus einem Emailpostfach, einer Pagernachricht oder Chatnachricht auf Plattformen wie Slack zusammensetzen können (siehe Abbildung 5.2).

Scaler Proxy ✓

Diese Komponente bietet eine REST-Schnittstelle, die im Alert-Manager hinterlegt wird. Sobald eine der Regeln anschlägt, wird der Aufruf an diese Komponente weitergeleitet. Im Endeffekt dient diese Komponente lediglich als Proxy-Service, da ihre primäre Aufgabe darin besteht diese Nachricht an eine weitere Komponente weiterzuleitet. In einer produktiven Umgebung würde diese Komponente komplett enfallen, da es möglich ist den Alert-Manager derartig zu konfigurieren, dass er direkt die öffentlichen Schnittstellen der Scaler-API anspricht. Es wurde sich dennoch für das Zwischenschalten eines solchen Proxy-Services entschieden um genauere Messwerte zu erhalten. Gerade hinsichtlich der Initialisierungsphasen von Containern ist es angebracht so kurz vorher wie möglich einen Timestamp zu setzen. Da es keine Möglichkeit gibt direkt in die Konfiguration der Scaler-Api einzugreifen ist diese Lösung der nächstbeste Ansatz. Sobald eine Skalierungsanfrage weitergeleitet wurde, wird diese noch unbeantwortete Anfrage intern in einer Datenstruktur abgelegt. Sobald ein entsprechender Container komplett initialisiert wurde, ruft dieser eine weitere Schnittstelle des Proxy-Services auf. Daraufhin wird ein *Acknowledgement-Timestamp* in der hinterlegten Anfrage gesetzt und an die Persistenzschicht weitergeleitet.

Außerdem bietet diese Komponente die Möglichkeit für den Benutzer direkt Exemplare eines Consumers hochzufahren. Die verwendete Schnittstelle dient als Umgehung des herkömmlichen Programmflusses und wird zur Generierung der skalierungsschichtenunabhängigen Metrikberechnung verwendet (siehe Abschnitt 6.1.2). Über dedizierte Endpunkte können diese Metriken als .csv Datei ausgelesen werden.

Scaler ✓

„*The goal of the Docker Scaler project is to provide a REST HTTP interface to scale services and nodes*“[Fan18]. Außerdem werden mit jeder Skalierungsanfrage Statusinformationen über die aktuelle sowie zukünftige Anzahl von Instanzen des zu skalierenden Services zurückgegeben, die es dem Proxy-Service ermöglichen davon ausgehend einen Skalierungsstop für neue Anfragen beizubehalten oder aufzuheben. Dieser ist nötig damit das System keine weiteren Skalierungsprozesse starten, wenn bereits aktive Skalierungen vorgenommen werden.

Grafana ✓

Wenn es zum Alert durch den Alert Manager kommt, wird in einer Produktivumgebung der Erste Schritt sein die Performanz des Systems durch dedizierte Dashboard zu überprüfen [Bra18, Seite 97]. Grafana ist ein Werkzeug, das dies über eine Weboberfläche direkt im Browser ermöglicht. Es bietet die Möglichkeit Graphen, Tabellen und weitere Visualisierungskomponenten zu erstellen um zum Beispiel die Latenzzeit oder CPU Auslastung zu überprüfen. Diese Metriken können für das ganze System oder nur einen Teil generiert werden. Es ist das bevorzugte Visualisierungswerkzeug für Prometheus, bietet allerdings ebenfalls Unterstützung für verschiedene weitere Systeme wie zum Beispiel *Graphite*, *Elasticsearch* oder *PostgreSQL*.

Mock scaler api ✓

Diese Komponente ist nicht Kernbestandteil des Komponenten-Stacks. Sie dient lediglich während der Entwicklungszeit dazu die Scaler-Api zu emulieren. So ist es zum Beispiel möglich lokal eine Instanz des Scaler-Proxy Projekts in einer beliebigen IDE als Standard Spring Projekt auszuführen. Sämtliche Skalierungs-Anfragen können von dieser Komponente angenommen werden, da sie auf dem gleichen Port wie der „*echte*“ Scaler läuft, wobei sie dabei ebenfalls in der Lage ist diverse Rückgabenachrichten an den Klienten zu übergeben.

5.1.3 Input ✓

Der Benutzer besitzt die Möglichkeit, sowohl über eine Benutzeroberfläche als auch über Bash-Skripte Benchmark Anfragen an das System zu stellen. Im Folgenden wird beides im Detail erläutert.

Bash ✓

Damit eine direkte Skalierungsaufforderung an das System gesendet werden kann, wird folgender Endpunkt verwendet.

```
curl "http://$HOST:$DIRECT_SCALING_PORT/manual-scale
            ?additionalCnt=$CNT
            &service=$SERVICE_NAME"
```

Es handelt sich hierbei um eine Get-Anfrage an die *Scaler-Proxy* Komponente des Stacks. Über die beiden Parameter „*additionalCnt*“ und „*service*“ lässt sich steuern wie viele neue Service Instanzen vom Orchestrator bereitgestellt werden sollen. Um dem Benutzer eine schlichtere Einstiegsmöglichkeit zu bieten, wurde das Skript *direct-scaling.sh* implementiert (Inhalt siehe Listing 5.1.3). Dieses ist in der Lage automatisch eine Reihe von Anfragen an das System zu stellen. Vor der ersten Ausführung

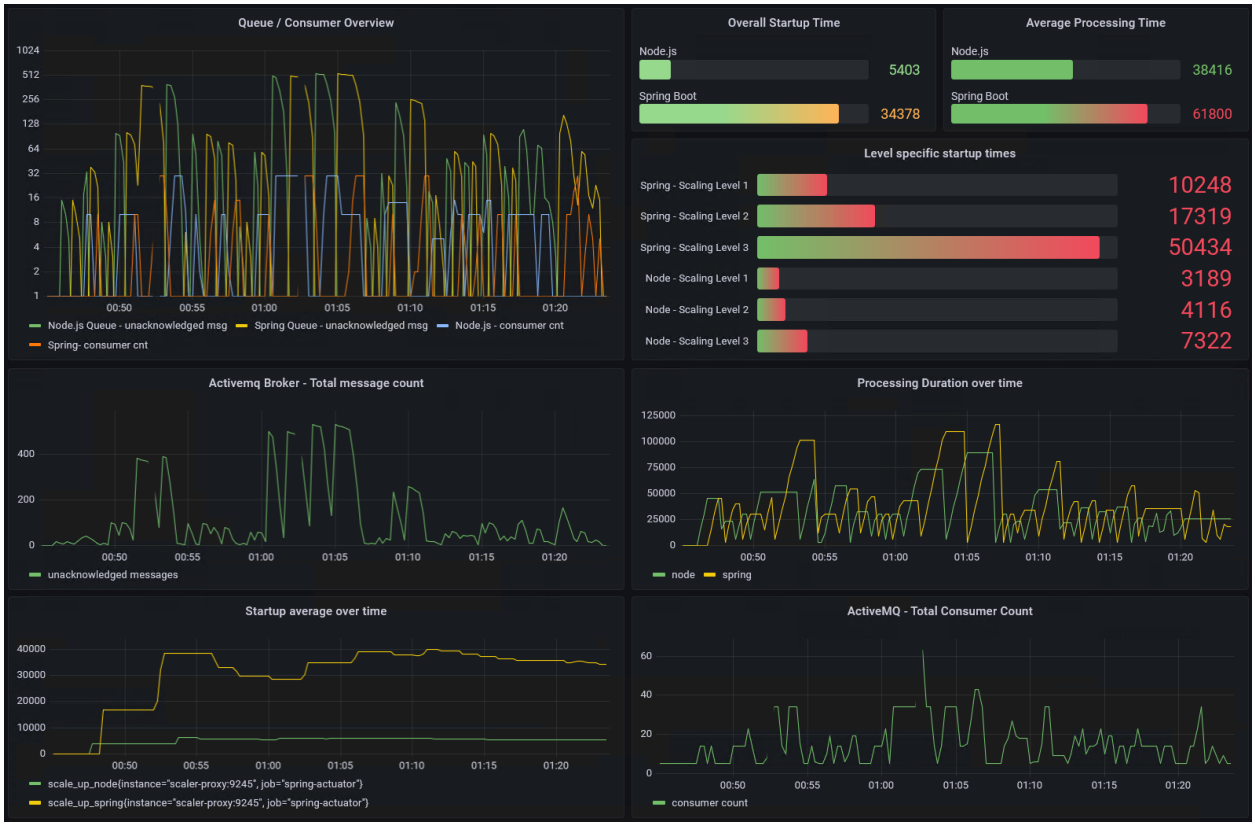


Abbildung 5.3: Grafana Dashboard

muss der Benutzer allerdings die IP-Adresse in der beiligenden .env Datei mit der seines Systems überschreiben.

```
$ tree request-scripts/ -a -L 3 --charset=ascii
request-scripts/
|-- curl-benchmark.sh
|-- direct-scaling.sh
|-- .env
|-- generate-traffic.sh
|-- README.md
`-- requests
    |-- mixed
    |   |-- benchmark_large_mixed.benchmark
    |   |-- benchmark_medium_mixed.benchmark
    |   `-- benchmark_small_mixed.benchmark
    ...
```

Listing 5.1: request-script-structure

Das Skript nimmt zwei Parameter entgegen. Der erste beschreibt die Anzahl der Instanzen, zu der das System skaliert werden soll, wobei hierbei eine inkrementelle Erhöhung von einer Instanz pro Durchlauf durchgeführt wird. Der Zweite Parameter gibt die Wiederholungen dieser testweisen Skalierung an. Der Aufruf `./direct-scaling.sh 10 20` resultiert beispielsweise darin, das sowohl für den Node.js als auch Spring Boot Service jeweils 10 Skalierungsschritte durchgeführt werden. Wenn zum Beispiel bereits ein Node.js Container läuft, wird im ersten Test eine weitere Instanz angefordert. Sobald die beiden Instanzen verfügbar sind, wird der Alert-Manager durch die Prometheus Komponente angewiesen die überflüssige Instanz zu löschen. Sobald wieder ein einziger

Container verfügbar ist, wird dieser Schritt neunzehn weitere Male wiederholt. Nachdem dieser Schritt abgearbeitet wurde, erfolgt die nächste Skalierung bei der nun zwei neue Container kreiert werden sollen. Auch hier folgen 20 Wiederholungen. Diese Schritte werden solange durchgeführt bis zehn Container zwanzig Mal instanziiert wurden.

```
...

request() {
  curl "http://$HOST:$DIRECT_SCALING_PORT/manual-scale?additionalCnt=$1&service=$4"
}

node_request() {request $1 $2 $3 'NODE'}
spring_request() {request $1 $2 $3 'SPRING'}

for scalingOffset in $(seq 1 $1)
do
  for curr_rep in $(seq 1 $2)
        do
    node_request $scalingOffset $curr_rep $2
    spring_request $scalingOffset $curr_rep $2
  done
done
```

Listing 5.2: direct-scaling

Das beschriebene Skript sowie die der zugrunde liegende Http-Endpunkt bieten eine Möglichkeit, das Skalieren als Reaktion auf unbeantwortete Nachrichten zu umgehen. Um jedoch einen herkömmlichen Testdurchlauf zu starten greift der Benutzer auf einen Endpunkt vom *Supplier* zurück:

```
curl
  -X POST
  -H "Content-Type: text/plain"
  --data "$REQUEST"
  "http://$API_HOST:$API_PORT/api/v1/parser/benchmark"
```

Es handelt sich hierbei um eine POST-Anfrage. Im Body wird der Dateiinhalt eines für diesen Zweck verfassten Skripts angeheftet. Im Projekt wurden diverse Beispielskript verfasst. Diese liegen im Ordner *./requests* und tragen die Dateiendung „*.benchmark*“. Der Inhalt korrespondiert zu einer spezifizierten Grammatik (siehe Listing 5.1.3).

```
request     := batch*
batch       := serviceName { instruction | instruction,* };
serviceName := SPRING | NODE
instruction := BENCHMARK ( messageCnt, duration ) | WAIT ( duration )
messageCnt  := [0-9]+
duration    := [0-9]+
```

Es können beliebig viele Batchanfragen den Anfragenbody angeheftet werden. Eine Batch stellt in diesem Zusammenhang eine Gruppe an Skalierungsanfragen an einen bestimmten Service dar, wobei diese mindestens eine Skalierungsinstruktion enthalten muss. In diesem System gibt es zwei wesentliche Skalierungsinstruktionen:

1. BENCHMARK: Nimmt zwei Parameter entgegen. Der erste spezifiziert, wie viele neue Instanzen erstellt werden sollen, während der Zweite angibt, über welchen Zeitraum dies geschehen soll. Alle numerischen Angaben müssen einen positiven Zahlenwert enthalten. Da das System selbstständig in der Lage diese Instanzen wiederum auf ein gesetztes Minimum zu minimieren, wurde sich aktiv dagegen entschieden, das Herunterskalieren in die Grammatik aufzunehmen, um dem Benutzer eine möglichst schlicht gehaltene Schnittstelle zur Verfügung zu stellen.
2. WAIT: Diese Instruktion erwartet lediglich einen Paramter, der angibt wie viele Millisekunden gewartet werden soll, bevor die nächste Instruktion dem System übergeben wird.

Damit der Benutzer allerdings direkt Anfragen ausführen kann, ohne sich mit der Grammatik beschäftigen zu müssen, wurde ein weiterer Skript-Aufsatz für das *curl-benchmark* Skript entwickelt. Dieses trägt den Namen „*generate-traffic.sh*“ und sucht rekursiv absteigend in der eigenen Directory nach Dateien mit der entsprechenden Endung. Anschließend werden diese Skripte an das *curl-benchmark* der Reihe nach als Parameter übergeben.

UI Erklärung einbinden...

Supplier Backend

Bei dieser Komponente handelt es sich um ein Spring Projekt, dessen Aufgabe es ist, Benutzeranfragen in Nachrichten zu übersetzen, die vom System interpretiert und verarbeitet werden können. Die Schnittstelle für den Benutzer besteht hierbei aus einer REST-Api, die über Tools wie zum Beispiel curl angesprochen werden kann.

```
$ tree stack/supplier-backend/ -a -L 7 --charset=ascii
stack/supplier-backend/
|-- Dockerfile
|-- pom.xml
`-- src
    `-- main
        |-- java
        |   `-- dps
        |       `-- hoffmann
        |           `-- producer
        |               |-- config
        |               |-- controller
        |               |-- model
        |               |-- properties
        |               |-- repository
        |               |-- response
        |               |-- service
        |               `-- SupplierBackendApplication.java
        `-- resources
            |-- application-dev.properties
            |-- application-prod.properties
            |-- application.properties
            |-- docs
            |   |-- sample-payment-short2.xml
            |   |-- sample-payment-short3.xml
            |   |-- sample-payment-short.xml
            |   `-- sample-payment.xml
            `-- schema.sql
```

Listing 5.3: Supplier Backend - Struktur

Implementierung

Generelle Spring Projektübersicht Da diese Komponente bezüglich der Struktur eines Spring-Projekts alle wesentlichen Merkmale besitzt, wird die genutzte Projektstruktur an dieser Stelle etwas näher erläutert.

Das Muster, nach dem sämtliche Komponenten des Komponenten-Stacks entwickelt wurden, wird als *Layered Architecture Pattern* oder *n-tier pattern* bezeichnet [ore]. Es stellt einen weitverbreiteten Standard in Java Enterprise Applikationen dar und zeichnet sich durch die klare Unterteilung der verschiedenen Hierarchien aus (siehe Abbildung 5.4). Sämtliche Packages innerhalb der Spring-Projekte finden sich innerhalb einer der Layer wieder. Es sei ebenfalls hervorzuheben, dass sämtliche Layer stets nur mit ihren direkt angrenzenden Schichten kommunizieren. Dies stellt ein weitaus übersichtlicheres Design sicher.

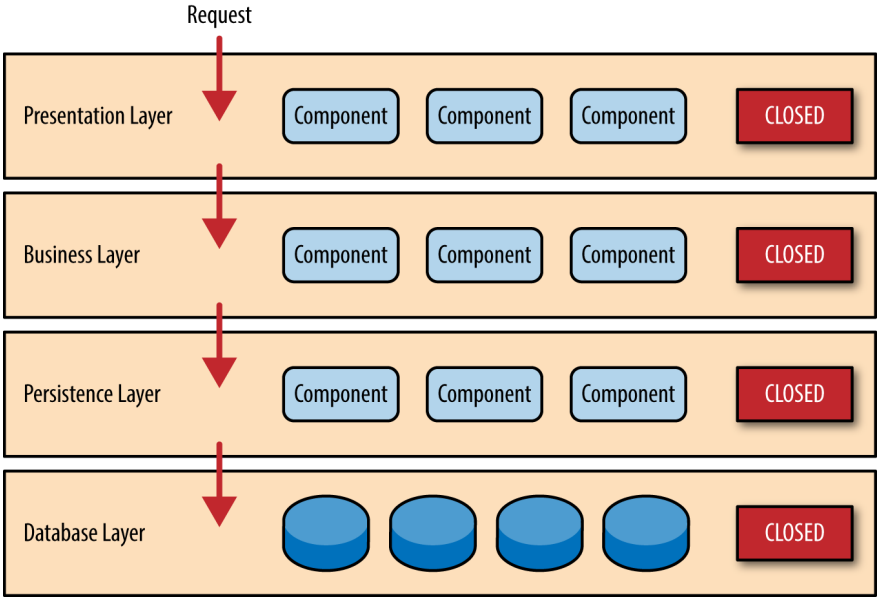


Abbildung 5.4: Layered Architecture [ore]

- *controller*: Dieses Package stellt alle Klassen bereit, welche direkt vom Benutzer angesprochen werden. Typischerweise handelt es sich zumindest im implementierten Stack um Rest-Schnittstellen, aber auch andere Typen (Soap, etc.) bieten entsprechende Implementierung. Klassen dieses Packages werden allgemeinhin als Controller bezeichnet und bilden eine erste Interaktionsschicht, welche keinerlei Business-Logik enthält. Bei diesem Package handelt es sich um einen Teil des Presentation Layers. Die Daten werden angenommen aber nicht direkt verarbeitet. Dies geschieht in sogenannten Services, welche spezifische Funktionalität implementieren. Controller dienen hierbei als ein Verteiler der eingehenden Nachrichten zu diesen Services. Die Service-Instanzen werden den Controllern Über den Spring-Context mittels Dependency Injection bereitgestellt, sodass sich der Entwickler selbst sich nicht mit der Instanziierung etc. zu beschäftigen braucht.
- *service*: Dieses Package enthält ausschließlich Service Implementierungen. Ein Service stellt einen logischen Funktionsbaustein der Anwendung dar und kann in den Spring-Context injected werden. Quelle??
- *repository*: Spring bietet über das Framework die JPA Implementierung *Hibernate* nativen Support für die Anbindung an eine Datenbankverbindung. Dieses Package enthält Interfaces welche die Treiberschnittstelle beerben. Mit Hilfe spezifizierter Namenskonventionen ist es möglich Methodensignaturen Datenbankanfragen zu gestalten. Das Framework generiert hieraus die entsprechenden SQL-Abfragen.

- *config*: Klassen dieses Packages konfiguieren instanziieren beziehungsweise konfigurieren Spring-Beans. Es gibt bei vielen voreingestellten Spring-Komponenten die Möglichkeit Beans über Annotations in der implementierenden Klasse zu erzeugen, in bestimmten Situation wird jedoch Zugriff auf die Instanz selbst zum Initialisierungszeitpunkt gebraucht um entsprechende Konfigurationsschritte zu unternehmen. Dies geschieht standardmäßig in diesem Package.
- *model*: Klassen welche lediglich zum Abbilden bestimmter Datensätze genutzt werden, residieren in diesem Package. Hierbei ist es irrelevant ob es sich um JPA-Entitäten oder herkömmliche POJOs handelt.
- *properties*: Spring bietet die Möglichkeit Daten aus den Konfigurationsdateien innerhalb der Resources direkt in den Application-Context zu laden.
- *utils*: Klassen, die lediglich mit Business-Logik gefüllt sind und nicht vom Spring-Container verwaltet werden, sondern direkt im Programmcode instanziiert beziehungsweise aufgerufen werden, residieren in diesem Package.

Programmfluss - Endpunkt Benutzeroberfläche In Listing 5.4 ist die Definition desjenigen Endpunktes zu erkennen, der als primärer Einstiegspunkt für die eingehenden Anfragen der Benutzeroberfläche gilt. Sobald der Benutzer eine Eingabe tätigt und der Endpunkt angesprochen wird, übersetzt eine Objectmapper Instanz den erhaltenen String in eine Datenstruktur, welche alle relevanten Informationen enthält. Die Methode wurde außerdem mit einer *Transactional*-Annotation versehen, die es ermöglicht das Übermitteln mehrere Nachrichten an eine Queue in einer Transaktion durchzuführen (siehe einfügen), dies wird im späteren Verlauf von Bedeutung sein. Im Spring-Framework muss jedoch bereits der Einstiegspunkt mit einer solchen Annotation versehen werden, um zu gewährleisten, dass die aufgerufenen Logikbausteine alle im selben Transaktion-Kontext ausgeführt werden .

Referenz

Quelle?

```
1
2   ...
3
4   @Transactional(propagation = Propagation.REQUIRES_NEW)
5   @RequestMapping(value = "/start",
6           method = POST,
7           consumes = APPLICATION_JSON_VALUE)
8   public void startPaymentBenchmark(@RequestBody String jsonBody) throws JsonProcessingException {
9       log.info("new benchmark request: {}", jsonBody);
10      ParsedInstruction req = mapper.readValue(jsonBody, ParsedInstruction.class);
11      benchmarkService.benchmark(req);
12  }
13
14  ...
```

Listing 5.4: Supplier - Endpunkt

```
1
2   ...
3
4   @SneakyThrows
5   @Transactional
6   public void benchmark(ParsedInstruction parsedInstruction) {
7       parsedInstruction.setReceived(now());
8       // set batch id by saving it to db and reloading instance
9       parsedInstruction = persistenceService.save(parsedInstruction);
```

```
10      int batchId = parsedInstruction.getMessageId();
11      boolean sessionIsTransacted = sessionIsTransacted(parsedInstruction);
12
13      Supplier<String> xPathSupplier = pathGenerator.getSupplier(parsedInstruction);
14      Supplier<String> paymentSupplier = payOptionGenerator.getSupplier(parsedInstruction);
15      Supplier<String> destinationSupplier = destGenerator.getSupplier(parsedInstruction);
16      BiConsumer<PaymentMessage, Supplier<String>> amqConsumer =
17              amqService.getConsumer(sessionIsTransacted);
18
19      int durationMillis = 0;
20      if (!sessionIsTransacted) {
21          durationMillis = parsedInstruction.getDuration()
22                  / (parsedInstruction.getMessageCnt() - 1);
23      }
24
25      for (int i = 0; i < parsedInstruction.getMessageCnt(); i++) {
26
27          PaymentMessage payment = PaymentMessage.builder()
28                  .batchId(batchId)
29                  .xPath(xPathSupplier.get())
30                  .content(paymentSupplier.get())
31                  .sentTimestamp(now())
32                  .build();
33
34          amqConsumer.accept(payment, destinationSupplier);
35
36          Thread.sleep(durationMillis);
37      }
38  }
39
40  ...
```

Listing 5.5: Supplier - Service

In Listing 5.5 ist zu erkennen, wie die Abarbeitung einer übersetzten Nachricht im Detail erfolgt.

- Zeile 7: Zu Beginn wird der aktuelle Timestamp an die übersetzte Nachricht angeheftet, um im Nachhinein diverse Messwerte berechnen zu können.
- Zeile 9: Die übersetzte Nachricht wird in der Datenbank hinterlegt.
- Zeile 11: Da die gesamte Methode mit einer weiteren *Transactional-Annotation* versehen wurde, ist es möglich beim Absenden der Nachrichten einen transaktionbasierten Ansatz zu wählen. Es muss jedoch bereits an dieser Stelle entschieden werden, ob dies tatsächlich nötig ist. Hierfür wird das übersetzte Datenobjekt entsprechend untersucht.
- Zeile 13: Abhängig von der gegebenen Instruktion ist es möglich unterschiedliche Felder eines Payments auszulesen. Es gibt beispielsweise die Möglichkeit eine IBAN, den Betrag der Überweisung, den Empfänger etc. zu extrahieren. An dieser Stelle wird ein entsprechender Supplier erzeugt, der im späteren Verlauf diese Information liefert und das standardisierte Java Interface des Function-Modules implementiert. Der Grund diese Funktionalität in eine Callback Funktion ausgelagert wurde, ist der, dass es möglich sein soll, dass sich die Ausgabe über Zeit ändern kann. So kann der Benutzer beispielsweise in der Nachricht angeben, dass er zufällige Datenfelder pro Nachricht ausgeben möchte. Bei Nachricht X, soll zum Beispiel das IBAN-Feld und bei Nachricht Y der Betragswert bestimmt werden. Diese Maßnahme wurde lediglich ergriffen um etwas Variation der Messwerte hinsichtlich der Verarbeitungsdauer zu generieren.

- Zeile 14: Selbes gilt für die Nachricht selbst. Es soll möglich sein, dem Benutzer die Wahl zu lassen, ob dieser die Benchmark-Tests stets mit der gleichen beispielhaften Payment-Nachricht durchführen möchte oder nicht. Falls ja soll Nachrichten mit zufälligen Werten zum Beispiel für die IBAN etc. zu generiert werden. In diesem Protoypen wurde allerdings lediglich das Design für die Funktionalität entwickelt, da es im Endeffekt keinen Einfluss auf die generierten Metriken hat welche Daten im Detail in dem XML-Inhalt hinterlegt wurden.
- Zeile 15: Diese Callback Funktion gibt den Warteschlangennamen aus zu dem die generierten Nachrichten geschickt werden sollen.
- Zeile 16: Abhängig davon, ob es sich um eine Anfrage, die in einer Transaktion durchgeführt werden soll oder nicht, wird für das Callback eine Spring-Bean zur Datenübermittlung zum Message-Broker verwendet, die mit einem Transaction Flag versehen wurde oder nicht. In Listing 5.6 ist erkenntlich, dass hierbei je nach gesetztem Flag eine Auswahl eines vorkonfigurierten JmsTemplates erfolgt. Dieses Template stellt eine weitere Abstraktionsstufe zwischen Komponente und Broker dar. Sie wird vom Framework gestellt, die Verbindung muss jedoch über Klassen im *config* Package konfiguriert werden. Diese Funktion stellt beim Absenden der Nachrichten die Schnittstelle zum Messagebroker dar.
- Zeile 27: Es werden Nachrichten generiert, die von den weiteren Stack-Komponenten verarbeitet werden können.
- Zeile 34: Die Nachrichten werden an die Broker-Schnittstelle übergeben.
- Zeile 36: Je nachdem ob bei der eingegangenen Nachricht ein Zeitraum angegeben wurde über den die Nachrichten abgeschickt werden soll, wird an dieser Stelle ein entsprechender Sleep-Aufruf durchgeführt.

```
1
2   ...
3
4   public BiConsumer<PaymentMessage, Supplier<String>> getConsumer(boolean sessionIsTransacted) {
5     return sessionIsTransacted
6             ? (message, destGen) -> sendObjPaymentQueueMessage(
7               transactedJmsTemplate, message, destGen)
8             : (message, destGen) -> sendObjPaymentQueueMessage(
9               nonTransactedJmsTemplate, message, destGen);
10  }
11
12  ...
```

Listing 5.6: Supplier - Bi Consumer

Programmfluss - Endpunkt Eingabedatei Da die Lastentests primär mit Hilfe diverser Bash-Skripte ausgeführt werden, gibt es einen weiteren Endpunkt für diesen Anwendungsfall. Nachdem Textnachrichten auf die Grammatik geprüft und in eine Objektstruktur konvertiert wurden, über die iteriert werden kann, wird allerdings die selbe Schnittstelle wie schon beim Gui-Aufruf angesteuert, da sich die darunter liegende Logik in keinster Weise unterscheidet. In Listing 5.7 ist der erwähnte Endpunkt zu erkennen.

```java
@Autowired
private RequestParserService parserService;

@Transactional(propagation = Propagation.REQUIRES_NEW)
@RequestMapping(value = "/benchmark",
        method = POST,
        consumes = TEXT_PLAIN_VALUE)
public void parseBenchmark(@RequestBody String textValue) {
    new Thread(() -> parserService.runRequest(textValue)).start();
}
```

Listing 5.7: Supplier - Bi Consumer

Ähnlich wie bei ersten Endpunkt gibt es wieder ein Transaktion-Context eröffnet. Bezüglich der Http-Anfrage hat sich lediglich geändert, dass nun anstatt eines Json-Wertes eine Textnachricht erwartet wird, da die hinterlegte Grammatik nicht mit einem Json-Format kompatibel ist . Um zu ermöglich, dass mehrere Benchmark Dateiinhalte das System direkt hintereinander mit Werten versorgen kann, wird hier mit jeder eingehenden Anfrage ein neuer Thread gestartet. Dies stellt auch sicher, dass die Anfragen direkt hinterlegt werden und alte Anfragen nicht erst abgearbeitet werden müssen bevor neue angenommen werden können. Der Zwischenschritt des Übersetzens der Grammatik in tatsächlich ausführbare Anweisung erfolgt in einer dedizierten Service-Komponente. Damit es zu keiner Verfälschung der Ergebnisse kommt, ist es von größter Bedeutung, dass lediglich nur ein einziger Thread mit der Ausführung der eigentlichen Tests zur Zeit betraut wird. Wenn es hierbei zu einem nebenläufigen Verhalten kommen sollte, würde dies die verfügbaren Resourcen einschränken und die Werte würden Ungenauigkeiten aufweisen, was es unter allen Umständen zu vermeiden gilt. Um diesem Problem entgegenzuwirken, wurde im *Parser-Service* eine Semaphore eingeführt, die sicherstellt, dass stets nur ein Thread arbeitet. Das Ausführen einer Anfrage wird demnach in einem kritischen Abschnitt ausgeführt.

schauen ob Gram matik ue berhaup vorher erklaert wurde

weitersch

Build Da alle Spring Projekte dieses Stacks mit einem fast identischen Dockerfile versehen wurden, gilt die folgende Beschreibung in gleichem Maß für alle weiteren Spring-Projekte.

Multista Build er klären / Quelle raussu chen

Spring Boot

5.1.4 Node.js

uncomm

Projektübersicht

Das Projekt beinhaltet neben sämtlichen Quellcode-Dateien ein Dockerfile, eine package.json und eine tsconfig Konfigurationsdatei.

Dockerfile Bei dem verwendeten Dockerfile handelt es sich um eine multi-stage Konfiguration.

Quellen raussu chen

Zeilennu

```
# stage 1 building the code
FROM node:10.15.3 AS builder
WORKDIR /usr/app
COPY package*.json ./
RUN npm install
COPY . .
```

```
RUN npm run build

# stage 2
FROM node:10.15.3-alpine
WORKDIR /usr/app
COPY package*.json ./
RUN npm install --production

COPY --from=builder /usr/app/dist ./dist

COPY --from=builder /usr/app/schema.sql .
COPY --from=builder /usr/app/specification.xsd .

CMD node dist/src/index.js
```

Typescript Konfiguration

Quellcode

uncomm

Die Programmlogik wurde in drei wesentliche packages aufgeteilt (siehe Listing).

Listing einkom- mentiere und refe renzieren

- *model*: Dieses Package beinhaltet sämtliche Datenstrukturen, die vom System zur Bearbeitung der eingehenden Nachrichten verwendet werden.
- *model*: Dieses Package beinhaltet sämtliche Komponenten welche logisch klar abgetrennte Aufgabe implementieren und über Schnittstellen miteinander kommunizieren. Hierbei wurde sich an der Architektur einer typischen Spring-Anwendung orientiert.
- *model*: Dieses Package beinhaltet sämtliche Funktionalität der abzuarbeitenden Businesslogik.

5.2 Implementierung mittels Containerisierungsplattform

5.2.1 Container Lifecycle

- Auf verschiedene Schichten eingehen
- Auf Ergebnisse beziehen

5.2.2 Docker Swarm

- Prototypen im Detail erlaeutern

```
cat .env

# qbn: queue bound (level) n
# cbn: container bound (level) n

QB0=15
QB1=30
QB2=100
CB0=1
CB1=5
CB2=10
```

Tabelle 5.1: Server Specs

Prozessor	Intel(R) Xeon(R) Gold 6226R CPU @ 2.90GHz
Kerne	6 Prozessoren á 16 Kerne
RAM	16 GB
Storage	150 GB

`CB3=30`

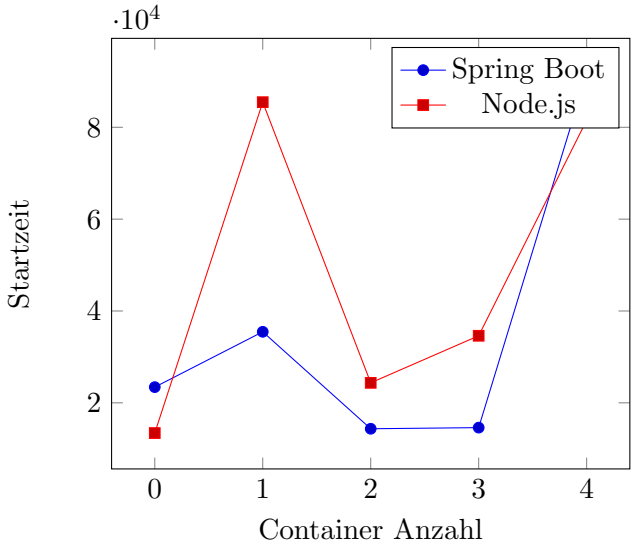
QL3 $QB2 < \overline{MC}$	UP *abs(CB0 – CB3)*	UP *abs(CB1 – CB3)*	UP *abs(CB2 – CB3)*	OK _
QL2 $QB1 < \overline{MC} \leq QB2$	UP *abs(CB0 – CB2)*	UP *abs(CB1 – CB2)*	OK _	DOWN *abs(CB2 – CB3)*
QL1 $QB0 < \overline{MC} \leq QB1$	UP *abs(CB0 – CB1)*	OK _	DOWN *abs(CB1 – CB2)*	DOWN *abs(CB1 – CB3)*
QL1 $QB0 < \overline{MC} \leq QB1$	UP *abs(CB0 – CB1)*	OK _	DOWN *abs(CB1 – CB2)*	DOWN *abs(CB1 – CB3)*
QL0 $QB0 == MC$	OK _	DOWN *abs(CB0 – CB1)*	DOWN *abs(CB0 – CB2)*	DOWN *abs(CB0 – CB3)*
	CL0 $CB0 == \overline{MC}$	CL1 $CB0 < \overline{MC} \leq CB1$	CL1 $CB1 < \overline{MC} \leq CB2$	CL1 $CB2 < \overline{MC} \leq CB3$

5.3 Implementierung Lasttest

5.3.1 Timeline

5.3.2 Testbedingungen

- Kommt in den Anhang
- hat Prof. zwar als eigenes Kapitel erwaehnt, bin mir aber nicht sicher ob das wirklich noetig ist
- auf welcher Hardware werden Tests durchgefuehrt?
- chaos monkey / Stoerfaelle erlaeutern



5.4 Implementierung Visualierung und Monitoring zur Unterstützung der Auswertung

```
1  // standard constructor
2  public Bank(int playerCnt) {
3      this.entries = new Entry[playerCnt];
4      this.bankSize = playerCnt;
5          this.rand = new Random();
6  }
7
8  // testing constructor - no fileIO
9  public Bank(Entry[] entries, Random pseudoRandom) {
10     this.entries = entries;
11     this.rand = pseudoRandom;
12     this.bankSize = entries.length;
13 }
14
15 // testing constructor - with fileIO
16 public Bank(String preallocation, List<Player> players, Random rand) {
17     assert null != preallocation && null != players && null != rand;
18     this.bankSize = players.size();
19     this.entries = new Entry[this.bankSize];
20     if (0 < preallocation.length()) {
21         String[] singleEntries = preallocation.split(SEPERATOR_STRING_REPRESENTATION);
22         int offset = this.bankSize - singleEntries.length;
23         for (int i = singleEntries.length - 1; i >= 0; i--) {
24             this.entries[i + offset] = new Entry(singleEntries[i], players);
25         }
26     }
27 }
```

```
1  #!/bin/bash
2
3  # Add two numeric value
4  ((sum=25+35))
5
6  #Print the result
```

```
7 echo $sum
```

5.5 Bestimmung von Daten zur Messung des Startup-Verhaltens von Containern

Nachdem im letzten Abschnitt auf mögliche Kriterien zur Messung der relevanten Metrikinformationen eingegangen wurde, erfolgt im folgenden eine spezifische Festlegung auf bestimmte Kriterien, welche auf das zu lösende Problem zugeschnitten werden.

5.5.1 Kriterienkatalog

- Vorallem Skalierbarkeit und Performance
- Transaktionen - Durchsatz wichtig
- Deployment egal, da bereits im Container deployed wird
- Generell sagen, warum einige Aspekte egal sind
- Wartbarkeit nicht so wichtig
- Ressourcennutzung
- Störungsfälle (Chaos Monkey)
- Non Functionals - Kai hat PDF geschickt

Leistungsfähigkeit

Die wesentliche Metrik stellt hierbei der Datendurchsatz sowie die Latenzzeit. Die Latenzzeit sollte die komplette Pipeline abdecken, also vom Dateneingang durch den Benutzer, bis hin zur tatsächlichen Abspeicherung der Daten in der Persistenzschicht. Die gesamte Strecke lässt sich allerdings auch in noch kleinere Abschnitte unterteilen um bei unerwarteten Werte festzustellen an welcher Stelle genau es zu einem Engpass kommt. Die Unterteilung der gesamten Pipeline sollte zumindest in zwei Abschnitte erfolgen.

- Hierbei ist vorallem von Interesse ob sich die Zeitenrahmen vom Eingang der Daten bis zum Erhalt der Daten durch die Consumer in irgendeiner Weise verändert. Dies sollte zwar nicht der Fall sein, da beide Arten von Testläufen auf die selbe Eingangsschnittstelle zurückgreifen. Dennoch ist es wichtig festzustellen, falls dies dennoch der Fall sein sollte, denn dann handelt es sich um ein tieferliegendes Problem bei der Orchestrierung des Stacks.
- Außerdem sollte der Zeitrahmen von Erhalt der Daten durch die Consumer bis hin zur Interaktion mit der Persistenz-Schicht gesondert betrachtet werden. Dieser Abschnitt stellt den variablen Teil der Pipeline dar und ist von größtem Interesse.

Auf Werkzeuge wie zum Beispiel *Gatling* oder *JMeter* muss hierbei verzichtet werden, da diese nicht in der Lage sind in die Pipeline einzugreifen um die einzelnen Abschnitte gesondert betrachten / messen zu können. Die Metriken müssen sowohl mittels der kompilierten Arbeitsversion als auch der Skript-Version generiert werden um eine Gegenüberstellung anstellen zu können. Hierbei sollte es außerdem möglich sein, den zeitlichen Verlauf der Metriken darzustellen um nachvollziehen zu können, ob es Phasen gibt in denen die Bearbeitung von dem berechneten Durchschnittswert abweicht. Je nach Stärke der Abweichung gilt es zu evaluieren ob weitere statistische Kenngrößen diesbezüglich berechnet werden sollten oder nicht. Ein weiteres mögliches Kriterium wäre in diesem Zusammenhang zum Beispiel die Standardabweichung. Da die Latenzzeit stark mit dem eigentlichen Datendurchsatz zusammenhängt ist es nicht nötig diese noch einmal gesondert zu berechnen um Redundanz zu vermeiden.

footnote mit link zu gatling und jmeter downloa seite

Skalierbarkeit

Da es sich bei dem Projekt um eine containerisierte Cloudanwendung handelt, muss hierbei eine *Vertikale Skalierung* stattfinden. Um möglichst genaue Werte zu erreichen, müssen die zu skalierenden Container in engem Austausch mit dem System stehen. Es muss festgehalten werden, wann genau der Befehl zum Skalieren getätigt wird, und wann genau ein entsprechender Container gestartet ist. Als vollkommen initialisiert gilt ein Container, wenn neben dem eigenen Filesystem auch sämtliche relevante Kommunikationsschnittstellen gestartet und benötigte Datenverbindungen bestehen. Generell gilt, die erhobenen Metriken sollen jeweils getrennt pro Backend-Technologie erhoben werden um eine Gegenüberstellung zu gewährleisten.

Bezüglich der Skalierung gilt vorallem es soll ein vorausschauendes Regelmodell entworfen werden, welches die Container stufenweise erst dann skaliert, wenn eine Vielzahl von Nachrichten eine gewisse Latenzzeit überschreitet oder der Message Broker eine entsprechende Menge an unbeantworteten Nachrichten hält. Hierbei soll eine sinnvolle Festlegung gefunden werden. Um im Nachhinein genaue Vergleichswerte ermitteln zu können sollen lediglich die Konsumenten Container skaliert werden. Die restlichen Stackkomponenten laufen jeweils mit einer einzigen Instanz. Der Skalierungsprozess soll sowohl mittels eingegangener Nachrichten, als auch durch spezifische Benutzeranfrage, gestartet werden können. Diese Unterteilung soll in den persistierten Datensätzen erkennbar sein.

Im Ausblick erwähnen, dass es möglich ist Datenbanken, Broker etc. zu skalieren Gegenstand de nächsten Arbeit...

Die hierbei generierten Metriken sollen sich bezüglich der vom Nachrichteneingang generierten Skalierungen anhand des festgelegten Stufenmodels ausgegeben werden. Hierbei wird der Durchschnitt aller in dieser Stufe / Gruppe enthaltenen Skalierungen gebildet und dargestellt. Es soll allerdings auch möglich sein die durchschnittlichen Werte sowohl für einzelne Containeranzahlen als auch für alle durchlaufen Skalierungen geben. Ähnlich zu den Metriken der *Leistungsfähigkeit* gilt, sie sollen ebenfalls zeitlich aufbereitet werden. Falls sich hierbei ungewöhnlich Auschweifungen hinsichlich der zeitlichen Entwicklung abzeichnen lassen sollten, können weitere statistische Metriken hinzugezogen werden.

Zusammenfassung

Der Übersichtlichtkeitshalber hier noch einmal eine stichwortartige Zusammenfassung der zu ermittelnden Metriken hinsichtlich der betrachteten Kriterien.

- Latenzzeit im Durchschnitt sowie als zeitliche Historie
  - Abschnitt *vor* Datenaufnahme gesondert betrachten

am Ende schauen ob genügend Zei um noch die Verlässlichk des Systems zu testen. Stichwor Chaos

  - Abschnitt *nach* Datenaufnahme gesondert betrachten
  - Gesamte Pipeline betrachten
- Skalierungsdauer jeweils pro verwendeter Backend-Technologie festzuhalten
  - Skalierung anhand eingehender Nachrichten mittels Stufenmodell
  - Skalierung mittels direkter Benutzeranfrage (ohne eingehende Nachrichten)
  - Metriken als Datensätze in einer Datenbank hinterlegt
  - Durchschnittliche Startzeit pro Containeranzahl
  - Durchschnittliche Startzeit pro Skalierungsstufe
  - Gesamtdurchschnittliche Startzeit als einzelner Messwert
  - Metriken als zeitlich visualisiert darstellen

6 Ergebnisanalyse

Nachdem im letzten Abschnitt ausführlich beschrieben wurde wie der komplette Komponentenstack im Detail implementiert wurde, werden im Folgenden auf die erhaltenen Metriken erläutert. Es wird insbesondere auf die Bedeutung einzelner Messwerte, sowie auf mögliche Begründungen dieser eingangen. Die genauen Messwerte wurden im Anhang aufgelistet und sind ebenfalls unter folgender Url einsehbar: https://github.com/derMacon/serverless-bsc-thesis/tree/main/data

- Latenzzeit im Durchschnitt sowie als zeitliche Historie
  - Abschnitt *vor* Datenaufnahme gesondert betrachten
  - Abschnitt *nach* Datenaufnahme gesondert betrachten
  - Gesamte Pipeline betrachten
- Skalierungsdauer jeweils pro verwendeter Backend-Technologie festzuhalten
  - Skalierung anhand eingehender Nachrichten mittels Stufenmodell
  - Skalierung mittels direkter Benutzeranfrage (ohne eingehende Nachrichten)
  - Metriken als Datensätze in einer Datenbank hinterlegt
  - Durchschnittliche Startzeit pro Containeranzahl
  - Durchschnittliche Startzeit pro Skalierungsstufe
  - Gesamtdurchschnittliche Startzeit als einzelner Messwert
  - Metriken als zeitlicher visualisiert

6.1 Ergebnisse

- Vorstellung der erhaltenen Daten
- Interpretation / Analyse allerdings Teil vom naechsten Kapitel?

nochmal schau- en ob Überlei- tung noc stimmt wenn Im plementi rungstei geschrie- ben wur de...

Verweis auf An- hang mi allen Ta- bellen fü Metrikin formatio nen

am En- de noch- mal nac schauen ob URL weiterhi stimmt

folgende Stichpun te entfer nen

6.1.1 Latenzzeit

Die wichtigsten Messdaten bezüglich der Leistungsfähigkeit des Systems beziehen sich auf den Datendurchsatz beziehungsweise die entsprechende Latenzzeit. In der Tabelle 6.1 wurden diese für die beiden betrachteten Backendtechnologien gegenübergestellt. Alle Angaben wurden in Millisekunden erfasst. Ein im System eingegangene Nachricht verweilt bei Bearbeitung durch einen Node.js Konsumenten im Schnitt 38 Sekunden, während sich dieser Wert bei Spring Boot auf über eine Minute beläuft. Diese Werte bilden lediglich einen Durchschnitt aller erhaltenen Metriken ab, eine genauere Aufteilung bezüglich des Zusammenhangs zu den Skalierungsgrößen erfolgt im nächsten Abschnitt (siehe). Der Gesamtdurchschnitt setzt sich aus zwei Zeitangaben der Pipeline zusammen:

Ref einfügen

1. Den Nachrichteneingang: Diese Metrik beschreibt den Zeitraum zwischen erhaltener Anfrage im System und Acknowledgement durch die Konsumer-Komponente, dass die Nachricht nun bearbeitet wird. Sie beläuft sich bei der Node.js Komponente auf ungefähr 34 Sekunden und 58 Sekunden bei der Spring-Boot-Komponente.
2. Die Verarbeitungsdauer: Diese Metrik beschreibt den Zeitraum zwischen Acknowledgement durch den Konsumer, dass die Nachricht erhalten wurde, sowie dem Abspeichern des extrahierten Wertes durch den Konsumenten in der Datenbank. Dieser Werte beläuft sich bei beiden Implementierung auf etwas über drei Sekunden. Bei dieser Metrik liegt der Fokus allerdings auf dem minimalen Zeitunterschied im Millisekundenbereich, da in beide Implementierungen eine künstliche Verlangsamung (*Sleep-Funktion*) eingebaut wurde um die erhaltenen Messwerte besser nachvollziehen zu können. Bei Node.js liegt die tatsächliche Verarbeitungsdauer bei 388 Millisekunden *(total 3388)*, während sie bei Spring Boot bei 29 Millisekunden liegt *(total 3029)*.

6.1.2 Skalierungsdauer

Diese Gruppe von Metriken bezieht sich auf die genauen Zeiträume, welche benötigt werden, um einen Container mit der entsprechenden Implementierung hochzufahren. Die ermittelten Zeiten stellen im Folgenden die durchschnittlichen Initialisierungszeiten eines einzelnen Containers dar. Sie beziehen sich nicht auf auf die Gesamtdauer der Initialisierungsphase aller beteiligten Container.

Tabelle 6.1: Latenzzeiten

Metrik	Dauer
Node.js	
Gesamtdurchschnitt	38416
Nachrichteneingang	35027
Verarbeitungsdauer	3388
Spring Boot	
Gesamtdurchschnitt	61800
Nachrichteneingang	58771
Verarbeitungsdauer	3029

Aufschlüsselung nach Services Die abstrakteste Metrik bezieht sich auf den durchschnittlichen Zeitraum zum Initialisieren der Container eines bestimmten Services. Hierbei wird zwischen der Node.js sowie Spring-Boot-Komponente zum konsumieren der Eingangsnachrichten unterschieden. Die Container, welche eine Node.js Instanz beinhalten, benötigen im Schnitt 6.6 Sekunden zum Hochfahren, während diejenigen mit einer Spring-Instanz 35.3 Sekunden brauchen. Diese relativ großen Zeiträume lassen sich unter anderem auf die Anzahl der hochgefahrenen Instanzen zurückführen. Eine genauere Aufschlüsselung dieses Prozesses erfolgt in den nächsten Abschnitten.

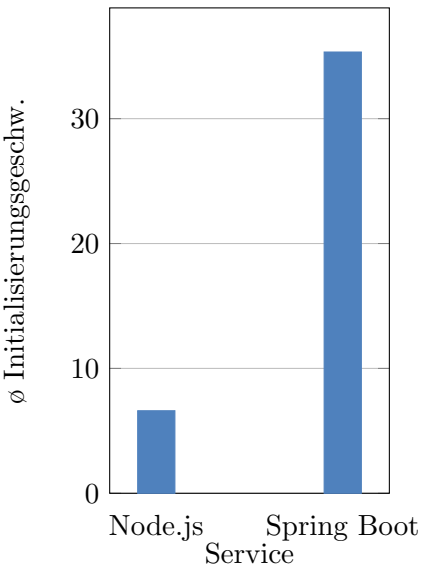


Abbildung 6.1: Serviceübersicht

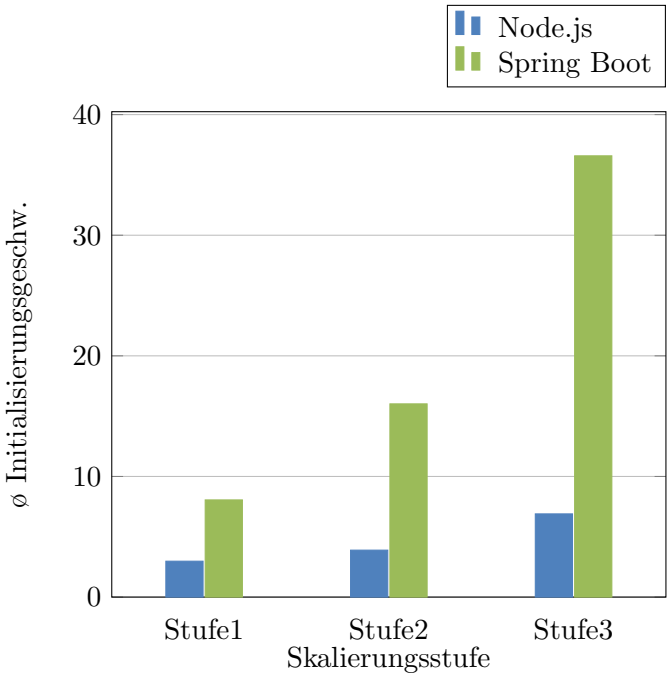


Abbildung 6.2: Stufenübersicht

Aufschlüsselung nach Skalierungsstufe Um einen beispielhaften Skalierngsalgorithmus zu implementieren wurde ein Regelsatz verfasst welcher vom Alert-Manager zur Laufzeit automatisch in einem festgelegten Interval ausgewertet wird. Um diesen Regelsatz zu Vorführungszwecken nicht unnötig ausführlich zu gestalten, wurde das Hinzufügen neuer Instanzen in Stufen organisiert.

Im durchgeführten Testszenario betrug die Grenze zum Überschreiten der ersten Stufe 15, für die zweite Stufe 30 und für die Dritte Stufe 100 unbeantwortete Nachrichten in der jeweiligen Warteschlange. Wenn im Folgenden von einem Burst gesprochen wird, ist hiermit das Überschreiten einer dieser Grenzen gemeint. Bei dem kleinst möglichen Burst bezüglich der Warteschlange der Node.js-Komponente, nimmt das System im Schnitt 2.9 Sekunden zum Starten der Container in Anspruch um der Arbeitslast nach Skalierung mit insgesamt 5 Containern zu begegnen. Bei der Spring-Boot-Komponente wurde hierfür ein Wert von 8 Sekunden gemessen. Im mittleren Grenz-Intervall wurde bezüglich Node.js eine Initialisierungsdauer von 3.8 Sekunden und auf Seiten von Spring 16 Sekunden festgestellt um nach der Skalierung mit 10 Instanzen zu arbeiten. Beim höchsten Burst werden kam es zu einer bei Node.js zu einer Initialisierungsphase von 6.9 Sekunden und bezüglich Spring zu 36.6 Sekunden. Hierbei sollen am Ende 30 Containerinstanzen des Konsumenten arbeitsbereit sein. Da beim Skalierungsalgorithmus mit den beschriebenen Stufen gearbeitet wird, kann hierbei nicht genau gesagt werden, wie viele Container genau im Endeffekt initialisiert wurden, da hierbei stets mit den Differenzen zwischen den Stufen gearbeitet wird. Wenn zum Beispiel bereits n Container laufen wenn neue Nachrichten einen Burst verursachen, sollen nur die benötigten Instanzen kreiert werden, um genau die fehlende Anzahl zu decken. Die genaue Übersicht nach parallelen Instanziierungen wurde im Paragraphen (siehe Abschnitt 6.1.2) genauer beschrieben.

Aufschlüsselung nach Containeranzahl Die grundlegendste Metrik beschreibt die Zeiten aufgeschlüsselt nach Anzahl gleichzeitig hochfahrender Container (siehe 6.3). Wie im Graphen erkennbar, handelt es sich um ein lineares Wachstum. Mit jedem zusätzlichen Container, der zeitgleich erstellt werden soll, dauert der Initialisierungsprozess bei der Spring-Implementierung im Schnitt 1611 Millisekunden länger. Bei der Node Implementierung liegt dieser Wert bei 194 Millisekunden. So dauert das Initialisieren eines einzelnen neuen Containers bei der Spring-Boot-Komponente 4,8

Sekunden während sich dieser Wert bei der Node.js-Koponente lediglich 2,6 Sekunden beläuft. Um diese nach Containeranzahl aufgeschlüsselten Werte zu erhalten, wurde nicht wie bei den anderen Skalierungstests auf die öffentliche Schnittstelle mittels Nachrichtengenerierung zurückgegriffen, es wurde stattdessen eine interne Schnittstelle mittels dediziertem Skript verwendet um möglichst Störungsfrei direkt auf die Komponente welche das Skalieren orchestriert zuzugreifen. Die nachfolgenden Metriken wurden allerdings durch die öffentliche Schnittstelle generiert.

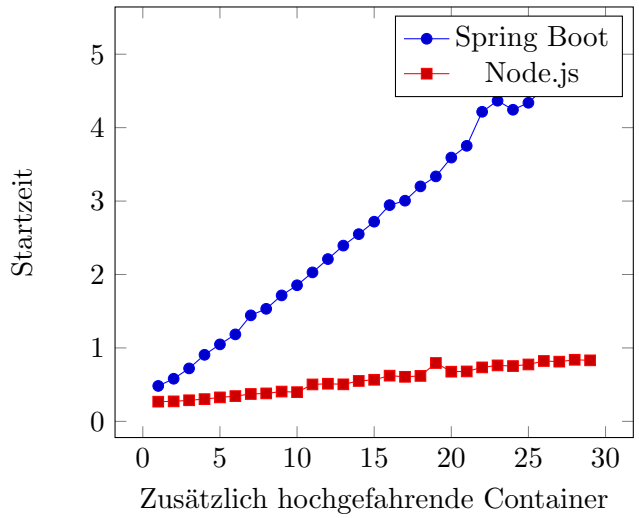


Abbildung 6.3: Anzahlspezifisch

Zeitliche Aufschlüsselung Wie in Bild 6.4 zu sehen, wurde diverse Panels zur Zeitlichen Übersicht bereitgestellt. Diese wurden allerdings vor allem zu Kontrollzwecken implementiert. Diese visuelle Darstellung war während der Entwicklungszeit sehr hilfreich um Fehler frühzeitig zu erkennen und beheben zu können, für die Auswertung der Ergebnisse haben diese allerdings wenig Relevanz. Der Vollständigkeit halber soll an dieser Stelle dennoch eine kurze Zusammenfassung der zeitbasierten Messwerte erfolgen. Für alle bisher behandelten Metriken gibt es Panels im Grafana Dashboard. Da die zeitliche Erfassung in Grafana selbst allerdings intervallbasierte Abfragen stellt, sind diese im Vergleich zu den Werten, welche in der Datenbank durch das System selbst generiert und Abgelegt wurden, sehr ungenau. Selbstverständlich wurde in der bisherigen Analyse ausschließlich auf diese Daten zurückgegriffen. Bezüglich der Konsumenten wurde neben dem insgesamten durchschnittlichen Startverhalten auch der jeweils aktuelle Initialisierungszeitpunkt dargestellt. Hierbei war eine klare Korrelation zwischen der Anzahl der parallel startenden Konsumenten mit der Erhöhten Initialisierungsdauer zu beobachten. Außerdem wurden diverse Metriken aus dem Activemq selbst ausgelesen. So ist es zum Beispiel möglich nachzuvollziehen, wie viele Nachrichten sich zu einem gegebenen Zeitpunkt nicht nur in einer bestimmten Warteschlange sondern auch innerhalb des ganzen Systems befinden. Außerdem wurde die zeitliche Abfolge zwischen einem Burst an unbeantworteten Nachrichten und dem Anstieg der Containerinstanzen in einem Dashboard-Panel dargestellt.

6.2 Analyse

- Interpretation / Analyse der Daten
- Begruendung fuer Verhalten suchen

6.2.1 Latenzzeit

Hinsichtlich der Latenzzeit überrascht vorallem die deutliche Diskrepanz zwischen der Initialisierungsdauer der unterschiedlichen Technologien. Durch die Unterteilung der Pipeline in die zwei Messwerte ist erkennbar, dass diese auf die zeitliche Dauer bis zum Nachrichteneingang zurückzuführen ist. Hierbei läuft die Dauer der Initialisierungsphasen weit auseinander (siehe 6.2.2).

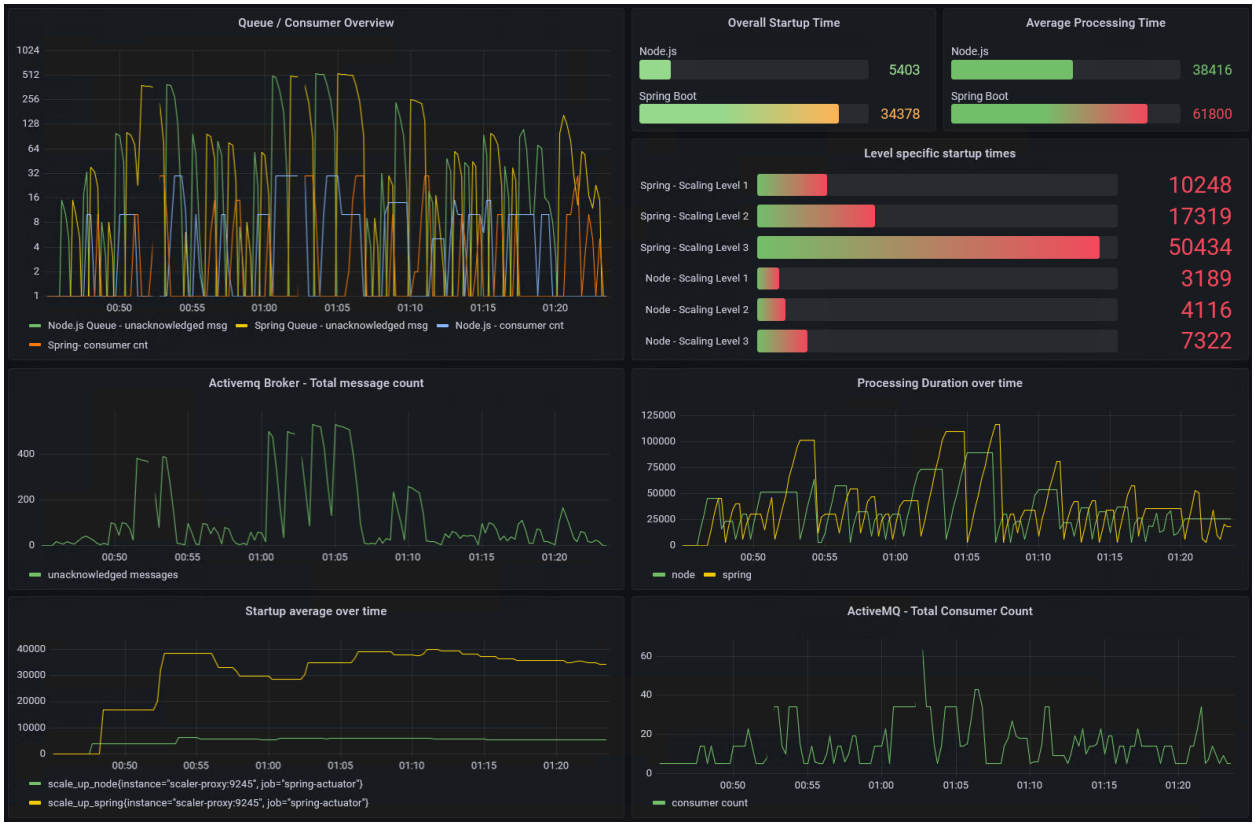


Abbildung 6.4: Grafana Dashboard

Bezüglich der Verarbeitungsgeschwindigkeit ist die Spring-Boot Komponente jedoch im Vergleich schneller. Dies ist auf die Natur einer kompilierten Sprache zurückzuführen. Java Code wird im Vorwege in entsprechenden Bytecode übersetzt, während eine Skriptsprache wie Javascript innerhalb der Node.js-Komponente zur Laufzeit übersetzt wird. Da in den Komponenten nur minimale Logik verbaut wurde, ist der festgestellte zeitliche Unterschied in der Bearbeitungsdauer mit 359 Millisekunden zwar nicht so gravierend wie der Unterschied hinsichtlich der Initialisierungsphase der Container, allerdings drastischer als vorher angenommen. Da der Prototyp eine sehr einfache Ausführungslogik implementiert, ist dieser Messwert doch relativ hoch. In Anbetracht der Komplexität der realen Banking-Anwendung gilt es ein spezifisches Konzept für das Regelwerk zum Starten der Container zu entwerfen. Hierbei muss eine Untersuchung bezüglich der Rentabilität eines neuen startenden Containers gegenüber eines laufenden Containers evaluiert werden.

6.2.2 Skalierungsdauer

Die erheblich abweichende Initialisierungsgeschwindigkeit der beiden unterschiedlichen Komponenten lässt sich im Kern auf auf die Initialisierungs des Spring-Containers sowie des dazugehörgien Application-Contexts zurückführen. Das Framework bietet durch die ... Kern des Spring Frameworks ist der *Spring Container*. Dieser verwaltet fachliche und nichtfachliche Objekte, die eine Anwendung ausmachen. Die verwalteten Objekte werden als *Beans* bezeichnet. „Eine Bean ist ein Objekt, das vom Spring-Container instanziiert und konfiguriert wurde und dessen Lebenszyklus vom Container verwaltet wird. Die Abhängigkeiten zwischen Beans sind als Metadaten im Container verfügbar“ [Sim18, Kapitel 3.1.1].

Weitersch schauen ob es die Erklärun schon irgendwe in der Thesis gibt.

6.3 Diskussion

6.3.1 Begründung Startupzeit

- Warum Node.js schneller ist
- Erlaeutern warum die erhaltenen Ergebnisse in einem real-life Szenario vielleicht nicht aussagekraeftig sein koennten

7

Zusammenfassung

8 Ausblick

9

Literaturverzeichnis

Literaturverzeichnis

[BA18] Daniel Bryant and Marin-Perez Abraham. *Continuous delivery in Java essential tools and best practices for deploying code to production.* O'Reilly, 2018.

[Bra18] Brian Brazil. *Prometheus: Up and Running: Infrastructure and Application Performance Monitoring.* O'Reilly Media, 2018.

[def] Definition Native Cloud-Anwendungen. https://www.computerweekly.com/de/definition/Native-Cloud-Anwendung. Aufgerufen am: 05-07-2021.

[doc] Docker Dokumentation. https://docs.docker.com/. Aufgerufen am: 06-07.2021.

[Eid05] Petter L. H. Eide. Quantification and traceability of requirements. https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.95.6464&rep=rep1&type=pdf, 2005.

[Fan18] Thomas J. Fan. Docker scaler - github pages. https://thomasjpfan.github.io/docker-scaler/, 2018.

[ins] Cloud Native - Definition. https://www.dertreasurer.de/themen/instant-payments-revolution-im-zahlungsverkehr/. Aufgerufen am: 26-06-2021.

[iso11] Iso/iec 25010:2011. https://www.iso.org/obp/ui/#iso:std:iso-iec:25010:ed-1:v1:en, 2011.

[mic] microprofile Spezifikation. https://access.redhat.com/documentation/en-us/jboss_enterprise_application_platform_continuous_delivery/15/html/configuration_guide/eclipse_microprofile. Aufgerufen am: 25-06-2021.

[nfr] non functional requirements - Evaluierung. https://www.dotnetcurry.com/project-management/1462/non-functional-requirements-nfrs. Aufgerufen am: 26-06-2021.

[ore] Layered Architecture - Definition. https://www.oreilly.com/library/view/software-architecture-patterns/9781491971437/ch01.html. Aufgerufen am: 26-06-2021.

[SBD11] Bruce Snyder, Dejan Bosanac, and Rob Davies. *ActiveMQ in Action* -. Manning Publications, Birmingham, 2011.

[Sim18] Michael Simons. *Spring Boot 2 - Moderne Softwareentwicklung mit Spring 5.* dpunkt.verlag, Heidelberg, 2018.

[Sop16] Fabrizio Soppelsa. *Native docker clustering with swarm.* Packt Publishing Limited, 2016.

10 Eidesstattliche Erklärung

Ich erkläre hiermit an Eides statt, dass ich die vorliegende Arbeit selbstständig und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe; die aus fremden Quellen direkt oder indirekt übernommenen Gedanken sind als solche kenntlich gemacht. Die Arbeit wurde bisher in ähnlicher Form keiner anderen Prüfungskommission vorgelegt und auch nicht veröffentlicht.

Ort, Datum Silas Hoffmann