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

- Was gibt es bisher, nur kurz anreissen
- wird im Grundlagen Kapitel tiefgehender behandelt...

Google, Spotify oder Netflix: Immer mehr Unternehmen setzen heutzutage auf Virtualisierungsplattformen. Aber warum sind Technologien wie zum Beispiel Docker als Containerisierungsplattform, oder Kubernetes als Orchestrator, so beliebt? In der folgenden Thesis möchte ich hierauf eingehen. Ich werden einen Überblick über die verfügbaren Technologien geben und auf diverse Eigenarten eingehen (was heißt es serverless zu deployen etc.).

1.2 Motivation

- Problemstellung kurz beschreiben
- DPS möchte untersuchen ...
- portierung der anwendung in die cloud (amazon) Problem ist: spring boot anwendung benötigt viel startup-zeit, heute ist die maximale skalierung in produktion vorbestimmt und muss immer vorgehalten werden, alle server sind 24x7x365 aktive und werden eigentlich nur in 1-2h pro tag in den spitzenzeiten ausgelastet... also nicht nur ineffiziente nutzung von hardware und energie sondern auch von kapital. -> hyperscaler geschäftsmodell verspricht abhilfe...
- hier ruhig auch einen Satz zur neuen Sau, die jetzt durchs dorf getrieben wird „new green economy“.. ressourcen sparen um eisbären und gretha thunberg zu retten... energieffizienz... etc bla bla bla sofort-ready vollautomatische Skalierung etc..das sind dann die gganzen Versprechungen der hyperscaler zu der cloud... kannst du hier alle auflisten. Kunde möchte vorbereitet sein, um ggf. Auflagen der Regulierer zu „green“ erfüllen zu können.
- Welche Erkenntnisse sollen gewonnen werden?
- Kriterienkatalog nennen

Mit den neuen Möglichkeiten die diese neuartigen Technologien mit sich bringen, möchte sich die DPS Gmbh in Zukunft ebenfalls in diese Richtung orientieren, denn die Ansprüche an eine moderne IT-Infrastruktur ändern sich. Man möchte effizienter arbeiten. Nicht nur bezogen auf verbrauchte Energie, sondern zum Beispiel auch auf nötiges Fachpersonal zur Wartung der Artifacts in Produktion. Denn diese Technologien ermöglichen ein Abstraktionslevel, welches es dem Anwendungsentwickler ermöglicht direkt Arbeitspakete für das Deployment zu generieren. Die Grenze zwischen der Operation- und der Entwicklungsabteilung verschwimmen immer mehr und es kommt zu weniger Tradeoffs zwischen mehreren Teams. Auch ermöglicht diese Abstraktion, falls gewünscht, ein komplettes Auslagern der Infrastruktur, wie es zum Beispiel mit Cloud Technologien der Fall ist.

Diese Abstraktion ermöglicht den Entwicklern sich mehr auf das Schreiben der Businesslogik zu konzentieren und können durch das Nutzen von Standartisierten Schnittstellen ein vollautomatisiertes Deployment gewährleisten. Diese Prinzip wird je nach Abstraktionstiefe auch als *Infrastructure as Code* (IaC) beziehungsweise *Infrastructure as a Service* (IaaS) bezeichnet. Gerade IaC erlaubt den Entwicklern ihre Programmierkenntnisse auf die Konfiguration von Infrastruktur zu übertragen. Auch die DPS plant in geraumer Zukunft auf Cloud Technologie zu setzen, langfristig ist hierbei eine Portierung in die Cloud von Amazon (AWS) geplant.

Ein Fokus, welchen ich mir in dieser Thesis im Detail anschauen möchte betrifft das Startup-Verhalten einer Anwendung in genau solch einer Virtualisierten Umgebung. Denn unter den genannten Effizienzgesichtspunkten möchte man heutzutage auch nötige Resourcen vorallem auf Anfrage verwenden und nicht mehr rund um die Uhr laufen lassen, selbst wenn dies zu einem gegebenen Zeitpunkt eigentlich gar nicht nötig wäre. Gerade im Banking Bereich gibt es Zeiträume in denen ein relativ geringer Geldfluss festzustellen ist, während es zu anderen Zeitpunkten zu regelrechten Burst kommen kann, wenn zum Beispiel zu Feiertagen relativ viel Geld den Besitzer wechselt. Um dieses Prinzip der Resourcennutzung 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 untersuchen 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.

2.1 Problemstellung

- Vergleich der Startup-Zeit einer containerbasierten Cloudanwendung bei steigender Last. Es soll auf Unterschiede zwischen Spring und Node eingegangen werden.
- insbesondere auf Durchsatz der Instant Payments eingehen
- da anwendung in spring boot ist nicht cloud-fähig weil zu lange für startup, dadurch riskieren wir timeouts also rejects – das ist die besonderheit der anwendung. . . - ..neuer container muss sofort verfügbar sein! Eine alternative wird gesucht: serverless vespricht minimale startup zeiten!....

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 ... beschrieben werden. Wichtig an dieser Stelle, es soll vorallem untersucht werden, inwiefern eine Java-Anwendung als cloudfähig bezeichnet werden kann. Denn wenn sie zu lange zum Starten braucht, 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. Um einen konkreten Vergleich durchführen zu können, wird im zu implementierenden Prototypen jeweils eine Komponente mit dem Java Backend Framework „*Spring Boot*“ und eine weitere Komponente mit dem Javascript Framework „*Node.js*“ implementiert.

Kapitel Referenz einbinde

2.2 Lösungsweg

- Prinzip / Ablauf erklären
- Grundlegende Struktur des Prototypen erklären
- vergleich der startupzeiten zwischen serverless und aktueller springboot variante anhand eines jeweils protypen und fiktiven workflows und fiktiven lastszenarios in einer cloud umgebung und messung des startup verhaltens bei plötzlich auftetenen spitzen oder so. . . .

Um festzustellen wie gut eine Spring- beziehungsweise eine Node.js-Komponente tatsächlich geeignet ist, wird ein fiktiver Workflow mittels eines Komponentenstacks implementiert, der in der Lage ist verschiedene Lastszenarien abarbeiten zu können. Hierbei ist es dem Benutzer möglich dynamisch zwischen der Spring- und Node-Komponente 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 Elemente zur Visualisierung greifen dabei nicht in den Ablauf des restlichen Stacks ein, sondern greifen lediglich auf bereits bestehende Schnittstellen zurück um einen möglichst ungestörten Ablauf und damit unverfälschte Datensätze zu garantieren.

Der wesentliche Usecase des Systems wird die Verarbeitung von generierten Payments sein, welche nach mehreren fiktiven Arbeitsschritten 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 hat.

[Knu98]

3 Ist-Analyse

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

3.1 JBoss (Microprofile)

- aktuelle Architektur beschreiben
- Hinweis darauf geben, dass Prototyp in der Thesis vereinfacht mit Spring dargestellt wird
- jetzt ja fiktiv SpringBoot, hier rein technischer Ist-Stand, Systemarchitektur....

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 entsprechende .war Dateien deployed welche den ausführbaren Code der Anwendung beinhalten. Der Application Server bietet dem Server-Teil der Client-Server-Anwendung eine Laufzeitumgebung, in der dieser ausgeführt werden kann. JBoss bietet nun standartisierte Schnittstellen nach dem jeweiligen Java Enterprise Standard um zum Beispiel die Kommunikation mit der Außenwelt zu ermöglichen oder um der Anwendung eine Persistenzschicht zur Verfügung zu stellen. Außerdem laufen die Instanzen im *Microprofile* Modus, der es ermöglicht, bestimmte Konfigurationsparameter auszulagern. So können Applikationen auf unterschiedlichen Systemen deployed werden ohne komplett neu gebaut zu werden ([mic]). Um eine gleichmäßige Aufteilung der Last zu gewährleisten, teilt ein so genannter „*Load Balancer*“ die eingehenden Nachrichten den entsprechenden Instanzen zu. 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 minimale Anzahl auch nicht Null betragen. Die maximale Poolsize stellt sicher, dass es zu keiner Überlastung des Systems kommt. Wenn eine Instanz bereits mit der maximale Anzahl an Prozessen arbeitet, wird dies dem Loadbalancer signalisiert und dieser teilt der entsprechenden Komponente in diesem Zeitraum keine weiteren Nachrichten mehr zu. Um zu gewährleisten, dass die Nachrichten nicht verloren gehen, werden sie in eine „*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.

Simples Schaubil einfuege

3.2 Probleme

Quelle einfügen

- Probleme mit aktuellem System (Stichwort Deployment, Wartbarkeit)

- Prof. hat extra darauf hingewiesen, dass es nicht nur um die Vorteile der Cloud gehen soll
- „Erwartete Probleme in einer Cloud Umgebung“ à aktuell „..starre Hardware und Software-Skalierung...“ unerwartete lastspitzen könnten zu problemen führen, wenn sie die erwartete und verfügbare obergrenze an kapazität übersteigt und wie oben gesagt ineffiziente nutzung von kapital....nochmal mit anderen worten aus
- 1.2 à die auflistung der probleme hier, müssen dann in der zusammenfassung wieder auftauchen und abgehakt werden! Die wollen wir ja auch lösen...

Mit der monolithischen Struktur des aktuellen Systems kommt es im Laufe der Zeit zu unterschiedlichen Problemen. Über die Zeit werden Komponenten immer weiter verstrickt, sodass es in einem solchen System auch nach Personalwechsel nicht einfacher wird den Überblick über die gesamt Applikation zu behalten. Dies führt zu fehleranfälligerem Code, welcher sich wiederum negativ auf das Kapital einer Firma niederschlägt. Diese Fehler führen in einem monolithischen System zu einem großräumigen Ausfall der gesamten Applikation, da es keine klare Abgrenzung der einzelnen Komponenten gibt.

Außerdem laufen die Application Server auf Servern, die (wenn überaupt) eine sehr dünne Abstraktionsschicht bieten („*starrer Hardware*“). Es ist nicht möglich diese schnell und flexibel zu ersetzen, falls es zu Fehlern in der Produktionsphase kommen sollte. Application Server bringen einen großen Konfigurationsaufwand mit sich, der sich schlecht automatisieren lässt. Man braucht geschultes Fachpersonal um ein solches System zum Laufen zu bekommen.

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 wie bereits erwähnt, besitzten Java-Anwendungen 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.

4 Vorgehensmodell

Nachdem im letzten Kapitel der aktuelle Aufbau der Payment-Anwendungen, sowie die durch die monolithische Struktur hervorgebrachten Probleme, erläutert wurden, werde ich im Folgenden beschreiben wie ich vorgehen werde um den Problemfokus der Startzeit zu untersuchen.

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

- Problem reduzieren
- Messkriterien festlegen
- Quellen finden
- hier nur theoretisch schwafeln über einen kriterienkatalog und die möglichen „Messwerte“, die man ggf. braucht -> in Kap. 5 dann den Kriterienkatalog konkret aufstellen

Der Fokus des Projekts liegt allerdings auf der Ermittlung der Cloudfähigkeit der unterschiedlichen System-Komponenten. Um dies festzustellen, muss im Vorwege eine Klassifizierung der Anforderungen stattfinden um festzustellen welche Messdaten für dieses Projekt überhaupt von Relevanz sind. Hierfür hat die ISO bereits einen ausführlichen Kriterienkatalog aufgestellt(siehe ISO/IEC 25010:2011 [iso11], fig. 4.2). Die ISO 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 Unterscheidungen dienen als eine erste Abstraktionsschicht. Functional requirements beschreiben hierbei die Korrektheit des Produktes [BA18]. „Sie spezifizieren die Funktion, die ein System oder Systemkomponente erfüllen muss“ [Eid05]. 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 4.1). 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 allerdings vernachlässigbar.

Zur Eingrenzung der Anforderungen an die Messdaten sind viel mehr 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. Im Folgenden werden die für das Projekt relevanten Kriterien einmal zusammengefasst dargestellt.

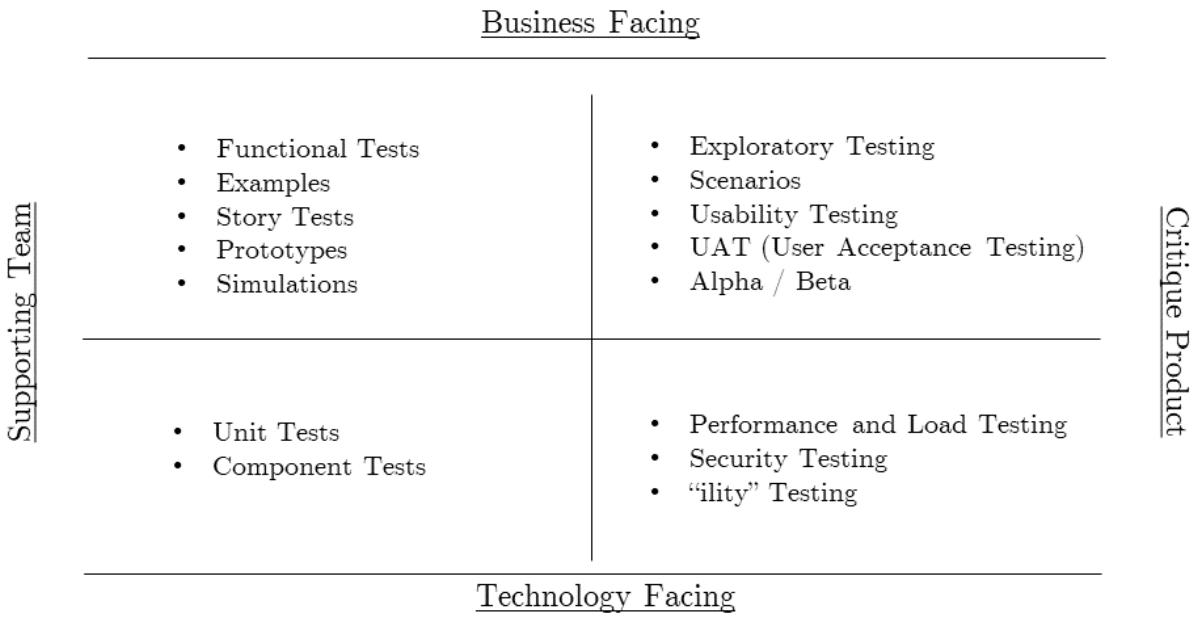


Abbildung 4.1: Agile Testing Quadrants

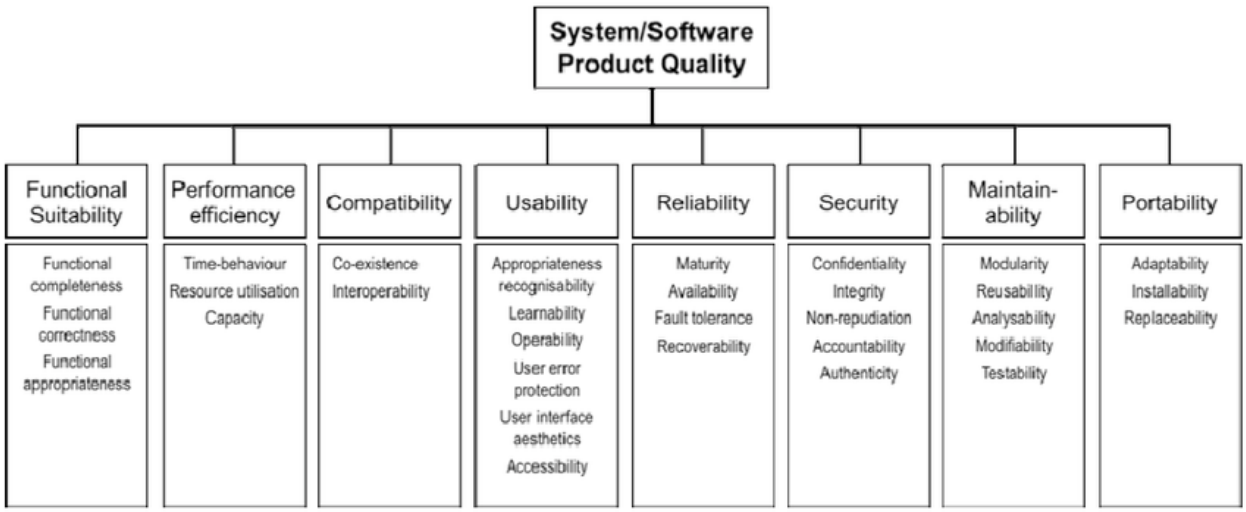


Abbildung 4.2: ISO 25010

Leistungsfähigkeit

- „Performance of a product or an app defines how a product/app is performing or behaving as compared to its expected behavior“
- Reaktionszeiten, wie schnell ist app wenn User interagiert
  - eng verwand *Latenz / Delay*, wie lange darf ein User max. warten müssen (Threshold ist die Reaktionszeit)
  - typischerweise pro Screen gesetzt, kann aber auch auf die Masse bezogen werden (bspw. 80 Prozent der Interaktionen haben eine Reaktionszeit von max. 3 Sekunden)

- Durchsatz: Anzahl der Daten, welche die Applikation handlen kann
  - kann sich sowohl auf Benutzeranfragen, db Anfragen, API calls etc. beziehen
  - Durchsatzziele als Teil der Anforderungsanalyse
  - werden mittels Performance / Load Testing validiert
- tools wie gatling / Jmeter fuer diesen Usecase unpassend, da nicht die Api Anfrage sondern nur die Hochfahrzeit gemessen werden soll
-

„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 vorallem durch die Reaktionszeit der Applikation geprägt. Es wird gemessen wie schnell auf Benutzereingaben eingegangen werden kann. Eng verwand ist hierbei auch der Begriff der *Latenzzeit*, da dieser beschreibt wie lange eine Beantwortung der Anfrage in 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 festzulegen, dass ein bestimmter Anteil der Interaktionen eine maximale Reaktionszeit von n Sekunden besitzt. Aber auch das zweite Unterkriterium der *Resourcennutzung* darf nicht vernachlässigt werden, so dürfen zum Beispiel bestimmte Kommunikationsschnittstellen nicht überlastet werden oder 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 allerdings nicht erwähnt. Im Idealfall möchte man nicht nur feststellen, wie lange Nachrichten verarbeitet werden, sondern auch wie viele im Schnitt verarbeitet werden können. Der Datendurchsatz ist also durchaus auch von größerem Interesse und sollte definitiv gemessen werden. Die verarbeiteten Daten können sich hierbei auch auf verschiedenes beziehen. Es können zum Beispiel die Anzahl von Anfragen an eine API sowie deren Rückmeldung sein, oder diverse Messages an Broker, welche 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 auch *Load Tests* validiert. Alles in allem stellt die Leistungsfähigkeit mit ihren Subkategorien eine der wohl wichtigsten Kriterien dar.

Skalierbarkeit

Hardware

- je nach Anforderungen hoch oder herunterfahren
- *concurrent users* : Wie viele Benutzer verwenden zeitgleich die Applikation. I.d.R. 20 - 30 Prozent der Userbase. Kann auf 100 Prozent bei besonderem Event hochgehen
- Zeitfaktor nicht vergessen, Applikation Skalierung sollte auch zeitlich gesteuert werden
- *vertikale Skalierung / scaling up* : Mehr CPUs / RAM fuer existierende Infrastruktur
- *hori Skalierung / scaling out* : mehr Server
- vielleicht Bild einfuegen scale out vs scale up

- cloud provider verwenden generell scale out

Software

- Bsp. Ladezeiten einer single page application zu hoch (z.B. Laden von Daten aus DB), scale out bringt hierbei nichts...
- Performanz sollte von Anfang an bedacht werden
- Frameworks sollten auf multithreading etc. ausgelegt sein, vor Verwendung einmal überprüfen
- Bottlenecks früh bestimmen, Legacy Libs eventuell austauschen
- Load Balancer sollte in irgendeiner Art vorhanden sein um Hardware Skalierung mit Software in Verbindung zu bringen
- Ohne Session Management ist die Performance schlecht, gilt es ebenfalls im Vorwege zu prüfen

Ein weiteres wichtiges Kriterium welches allerdings nicht direkt in der ISO 25010 Norm auftritt, ist die Skalierbarkeit des Systems. 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 Unterkreterium „*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 noch weitere Hardwarekomponenten hinzuziehen können. Inwiefern dies automatisiert oder manuell geschieht lässt sich nur fallabhängig bestimmen, es sollte jedoch evaluiert werden ob dies überhaupt generell möglich ist. Metriken zum Ermitteln des Skalierungszeitpunkt könnten zum Beispiel die Anzahl der parallel arbeiten Benutzer der Applikation (engl. „*concurrent users*“), die Latenzzeit oder die CPU Auslastung der beteiligten Maschinen sein. Es ist allerdings auch wichtig, dass eine Applikation nicht nur von aktiv eingehenden Ereignissen skaliert wird, sondern auch ein Zeitverhalten mit sich bringt und sich selbstständig zeitabhängig skalieren kann. Dies macht den Skalierungsprozess robuster, da sich die Komponente dadurch wieder etwas von seiner Umwelt abgrenzen 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* (auch *scale up* Skalierung) beschreibt das modifizieren der Servereigenen Resourcen. 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* (auch *scale out* Skalierung), welcher gerade dieses Auslagern bestimmter Komponenten beziehungsweise Prozessinstanzen auf neue Hardware beschreibt. Dieser Ansatz wird in der Cloud-Technologie verfolgt und nicht nur auf *on premise* , sondern auch von den großen *public cloud vendorn* wie zum Beispiel Amazon AWS oder Microsoft Azure, verwendet.

Footnote
on prem
se

Dennoch muss die Software auf eine Skalierung der Hardware ausgelegt sein. Manche Probleme lassen sich mit mehr Rechenperformance nicht lösen, wenn zum Beispiel die Ladezeiten bestimmter Kommunikationsschnittstellen einen Bottleneck darstellen. Alter Legacy Code könnte ebenfalls Performanceeinbußen darstellen und ohne ein vernünftiges Session Management bringt auch das Skalieren der Hardware nichts. Diese Grundfunktionalitäten werden heutzutage jedoch von fast allen Frameworks standardmäßig mitgeliefert. Dennoch gilt es dies im Vorwege noch einmal zu validieren.

footnote
public
cloud ve
dor

Kompatabilität

Dieses Hauptkriterium beschreibt, wie nebenläufig das System im Endeffekt ist. Ist es möglich, dass mehrere Komponenten gleichzeitig arbeiten können, ohne sich gegenseitig zu behindern? Gerade in einer Containerisierten Umgebung ist dies von fundamentaler Wichtigkeit. Der Unterbegriff *Ko-Existenz* beschreibt gerade dieses Verhalten. Der zweite Unterbegriff der „*Interoperabilität*“ beschreibt inwiefern die verschiedenen Komponenten in der Lage sind Daten untereinander auszutauschen. Wie genau dieser Datenaustausch aussieht kann sich von System zu System unterscheiden. Zusammenfassend lässt sich sagen, dass diese Kriterien in einer cloudbasierten Anwendung zwar gelten müssen, sie allerdings dennoch derartig abstrakt gehalten sind, dass es schwieriger ist hier messbare Metriken zu generieren. Entweder das System funktioniert oder es ist fehlerhaft, einen messbaren Mittelweg gibt es nicht direkt.

Zuverlässigkeit

- High Availability
- Angabe wie lange app garantiert laeuft (bspw. Prozentangabe pro Jahr)
- Design for failure: Komponenten darauf ausgelegt, dass Ausfall einzelner Komponenten zu keinem Systemausfall fuehrt
- accessibility: app ist nicht für die Allgemeinheit konzipiert worden

Das Hauptkriterium der *Zuverlässigkeit* beschreibt die Angabe wie lange eine Applikation garantiert läuft. Dies kann auf verschiedene Weisen festgehalten werden, eine typische wäre zum Beispiel den 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 Systeme auch „*Design for failure*“ genannt. Das Hauptkriterium setzt sich aus unter anderem aus dem Unterkriterium der *Reife* zusammen. Hierbei sind diejenigen Anforderungen gemeint, welche in einer normalen Prozessabarbeitungsphase ohne besondere Ereignisse hinsichtlich zum Beispiel der Anzahl der zu verarbeitenden Nachrichten, oder der Komplexität der zu bewältigenden Aufgaben, gelten müssen. Auch eine gewisse *Fehlertoleranz* wird 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 nicht. Das letzte Unterkriterium der *Zuverlässigkeit* ist die *Wiederherstellbarkeit* nach einem beschriebenen Fehlerfall, 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 man dort genaue Metriken ermitteln könnte. Im nächsten Kapitel wird festgehalten welche Metriken dennoch relevant erscheinen.

Ref einb den bzw schauen ob über- haupt aufgenor men...

irrelevante Themenfelder

- Security (Injection, Broken Authentication, Cross Site Scripting etc.)
- Nutzbarkeit (Benutzerfreundlichkeit, Browser Kompatabilität, Internationalisierung - Eingabesprachen)

Weitere Hauptkriterien der ISO Norm betreffen die *Nutzbarkeit*, sowie die *Sicherheit* der Anwendung. Da der Fokus des Projekts aber darin liegt die Skalierbarkeit sowie die Performance des Systems zu testen, können all diese Punkte jedoch vernachlässigt werden. Die Themengebiete der *Wartbarkeit* sowie der *Übertragbarkeit* sind in einer containerisierten Plattform zwar ebenfalls unabdingbar, sind allerdings derartig abstrakt gehalt, dass es den Rahmen dieser Thesis sprengen würde ebenfalls näher auf diese einzugehen.

4.2 Anforderungen an Prototypen

- einmal als spring boot und einer weiteren variante mit einer cloud-native technologie -> hier wurde vorgegeben mit serverless zu arbeiten.

Bei diesem Projekt sollen sowohl Orchestrierungsplattformen als auch die tatsächlich laufende Runtime Umgebung beziehungsweise ein entsprechendes Framework verglichen werden.

Hinsichtlich der Skalierung soll es beispielsweise möglich sein, anhand festgelegter Regeln eine automatische *horizontale* Skalierung stattfinden. Dieses Verhalten soll sowohl in hinterlegten Datensätzen als auch mittels eines eigens dafür eingerichteten Dashboards nachvollziehbar dargestellt werden. Auf welche Technologien sowohl bei der Skalierung sowie der Darstellung beziehungsweise Berechnung der Metriken zurückgegriffen wird, wird vom Arbeitgeber nicht vorgegeben. Lediglich hinsichtlich der Komponenten zur Verarbeitung der Business Logik wurde eine Vorauswahl getroffen. Wie bereits angedeutet, wird die aktuell laufende Java Enterprise Applikation mit dem moderneren Spring Boot Framework ersetzt um einen aktuelleren Vergleichspunkt in puncto kompilierbare Applikation getroffen. Um einen sinnvollen Vergleichspunkt anzusetzen, wird dem eine cloud-native Technologie gegenübergestellt. „Cloud Native beschreibt einen Software-Entwicklungs-Ansatz, bei dem Applikationen von Anfang an für den Einsatz in der Cloud konzipiert werden. Das Ergebnis sind Native-Cloud-Applikationen (NCAs), die die Stärken der Cloud-Computing-Architektur vollständig zu nutzen wissen.“ [**?**] Mit dieser doch relativ abstrakt gehaltenen Definition, sind hinsichtlich der verarbeitenden Komponenten allerdings in erster Linie die modernen Skript-Frameworks gemeint. Hierbei gibt es mehrere Alternativen, wo es seitens des Arbeitgebers allerdings ebenfalls keinerlei genaue Vorgaben hinsichtlich des spezifischen Frameworks / Technologie gibt.

4.2.1 Festlegung fiktiver Workflow

Da der Fokus auf der Untersuchung der Startupzeit der Komponenten liegt, wird lediglich eine minimale beispielhafte Implementierung erfolgen, welche die Arbeitsschritte der eigentlichen Applikation in 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.

Auszug aus XML einbinden???

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

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

4.2.2 Spring Boot

Wie bereits angedeutet, wird die aktuell laufende Java Enterprise Applikation mit dem moderneren Spring Boot Framework ersetzt um einen aktuelleren Vergleichspunkt in puncto kompilierbare Applikation getroffen. Um einen sinnvollen Vergleichspunkt anzusetzen, wird dem eine cloud-native Technologie gegenübergestellt. Die Komponenten sollen beide die beschriebenen Arbeitsschritte abbilden.

4.2.3 Serverless

- 4.2.2 spring boot 4.2.3 serverless -> so bekommt der prof schon beim draufgucken auf das inhaltsverzeichnis die story mit....

Um der kompilierten Spring Boot Anwendung eine Technologie ein auf skalierbare Komponenten ausgelegt Technologie gegenüberzustellen, wird hierbei auf eine cloud-native Technologie gesetzt. „Cloud Native beschreibt einen Software-Entwicklungs-Ansatz, bei dem Applikationen von Anfang an für den Einsatz in der Cloud konzipiert werden. Das Ergebnis sind Native-Cloud-Applikationen (NCAs), die die Stärken der Cloud-Computing-Architektur vollständig zu nutzen wissen.“ [**?**] Mit dieser doch relativ abstrakt gehaltenen Definition, sind hinsichtlich der verarbeitenden Komponenten allerdings in erster Linie die modernen Skript-Frameworks gemeint. Hierbei gibt es mehrere Alternativen, wo es seitens des Arbeitgebers allerdings ebenfalls keinerlei genaue Vorgaben hinsichtlich des spezifischen Frameworks / Technologie gibt.

4.3 Anfoderungen an Containerisierungsplattform / Orchestrierungsplattform

Es gibt zwar diverse Containerisierungsplattformen dennoch hat sich vorallem die Technologie „*Docker*“ durchgesetzt, die Vorteile und generelle Funktionsweise wurden bereits relativ ausführlich erläutert. Deshalb soll diese auch im Prototypen Verwendung finden. Hinsichtlich der Orchestrierungsplattform (welche auch als *Clustering Werkzeug* dient) werden folgende Anforderungen formuliert.

continuo
delivery
tieren u
aufzaehl

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 detailierte 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 meherere Systeme sollte zwar einmal testweise erfolgen, für die Generierung der Metriken ist dies allerdings eher hinderlich, da es die Ergebnisse verfälschen könnte. Da es dem Design der Orchestrierungsplattform geschuldet ist, dass der Anwender wenig Einfluss auf die Resourcenzuteilung hat, kann es passieren, dass eine leistungsschwächere Maschine deutlich mehr Initialisierungsanfragen zugewiesen bekommt. Dies könnte man zwar dadurch umgehen, dass man zwei exakt gleichbestückte Server für den Lastentest verwendet. Hierbei würden im Nachhinein allerdings die gleichen Ergebnisse erlangt werden, wie bei Testlauf auf einer Maschine.

4.5 Anforderungen Visualiserung und Monitoring zur Unterstützung der Auswertung

Es muss möglich sein, Metriken automatisch vom System generiern zu lassne. 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 Visualierung soll in Echtzeit oder einem zeitlich festgelegten Intervall aktualisiert werden können. Bezüglich des genauen Werkzeugs werden keinerlei Vorgaben gegeben, um dem Benuzter jedoch jeglichen Installationsaufwand zu ersparen wäre eine webbasierte Darstellung sicherlich von Vorteil.

5 Problemlösung

5.1 Implementierung Prototyp

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.

5.1.1 Komponentenmodell

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

- Persistenzschicht (engl. *Persistence layer*): Beinhaltet sämtliche Komponenten, welche für das Abspeichern gegebener Datensätze in dazugehörige Datenbanken zuständig sind. Um Nebenläufigkeit zu ermöglichen wird hier ebenfalls auf Schnittstellen mittels Message-Broker zurückgegriffen. Eine manipulation der Daten findet auf dieser Ebene nicht statt.
- Verarbeitungsschicht (engl. *processing layer*): Beinhaltet sämtliche Komponenten, welche 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 vorallem 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 einzige 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 unterzubringen.

Komponenten im Überblick Es folgt eine kurze Zusammenfassung der Funktionalität sowie der Anforderungen an die jeweligen Komponenten.

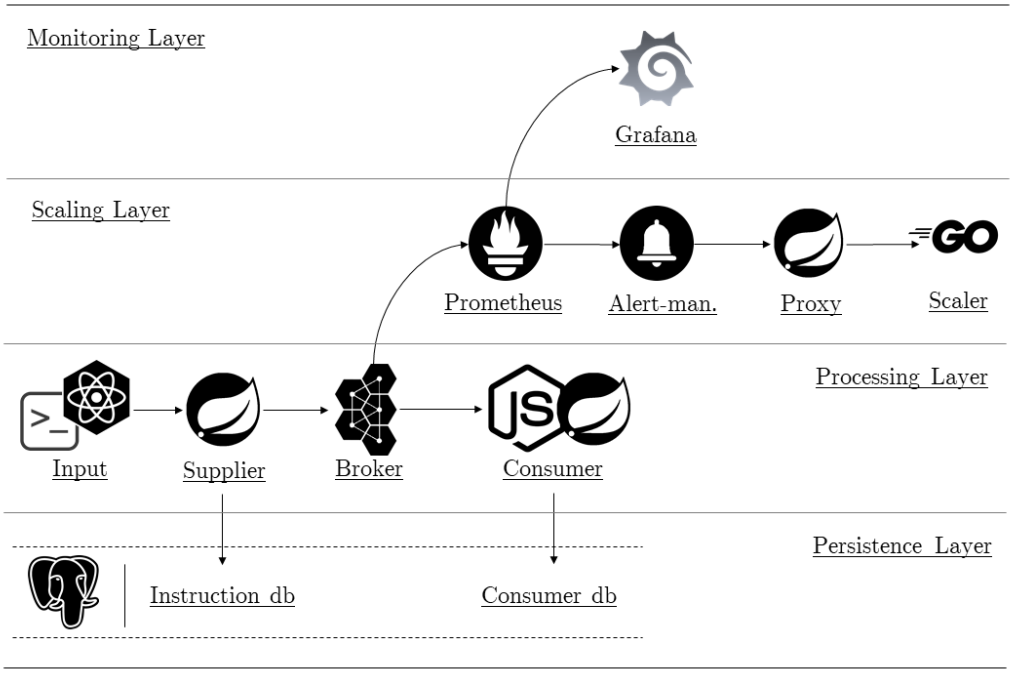


Abbildung 5.1: Komponenten-Stack im Überblick

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, es ist allerdings beabsichtigt, dass der Benutzer auf vordefinierte Skript 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 von bereitgestellten Bash-Skripten. Bezüglich der Skripte gibt es ebenfalls eine diverse Abstraktionsschichten. So ist es zum Beispiel möglich mittels einer definierten Grammatik Anforderungen zu definieren welche sich beliebig kombinieren lassen. Hierzu wurden mehrere vordefinierte Dateien angelegt, welche über ein entsprechendes Skript an das Backend geschickt werden kann. Allerdings gibt es Skripte die auf diesem Prozess aufsetzen, sodass der Endbenutzer sich 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 angehängt 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 es im System stets nur diese eine Instanz des *Suppliers* gibt. Zwar unterstützt die verwendete Postgres-Datenbank mehrere Klienten zur gleichen Zeit, nativ ist diese Anzahl jedoch begrenzt und muss angemessen konfiguriert werden. Desweiteren 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 Dieser Activemq Broker stellt die Funktionalität bereit Nachrichten an mehere Konsumenten zu vermitteln. Dabei gibt es 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 eingeschriebenen Konsumenten verschickt werden . Demgegenüber steht eine *queue* (Warteschlange), welche eingehende Nachrichten lediglich an einen einzelnen Konsumenten übermittelt, wobei er hierbei als eine Art Loadbalancer agiert. Bevor die Nachrichten jedoch aus einer der beiden Datenstrukturen entfernt wird, 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 auch noch hervorzuheben, dass die eingeschriebenen Konsumenten bei neuen Nachrichten stets benachrichtigt werden und es clientseitig keine Logik benötigt um zum Beispiel event- oder intervallbasiert eine Abfrage an den Broker zu steuern. Da diese Komponente ebenfalls vom ... *Prometheus erklären, Links raussuchen.*

Quelle verlinken

Consumer Die konsumierenden Komponenten können beliebig skaliert werden und implementieren den in Abschnitt **??** definierten Workflow. 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

Alert Manager

Quellen raussuchen

Scaler Proxy Diese Komponente bietet eine REST-Schnittstelle, welche im Alert-Manager hinterlegt wird. Sobald eine der Regeln anschlägt wird der Aufruf an diese Komponente weitergeleitet. Im Endeffekt dient diese Kompoonente nur als Proxy-Service, da sie diese Nachricht lediglich an eine weitere Komponente weiterleitet. 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 Messewerte zu erhalten. Gerade hinsichtliche der Initialisierungsphasen von Containern ist es angebracht so kurz vorher wie möglich einen Timestamp zu setzen. Da es nicht möglich ist 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.

Quellen raussuchen

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

fill out

die Zugriff auf den Docker-Daemon besitzt und dort den Swarm steuern und damit skalieren kann.

Grafana

Github Link

Mock scaler API Diese Komponente ist nicht Kernbestandteil des Komponenten-Stacks. Während der Entwicklungszeit dient sie 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.

Quellen raussuchen

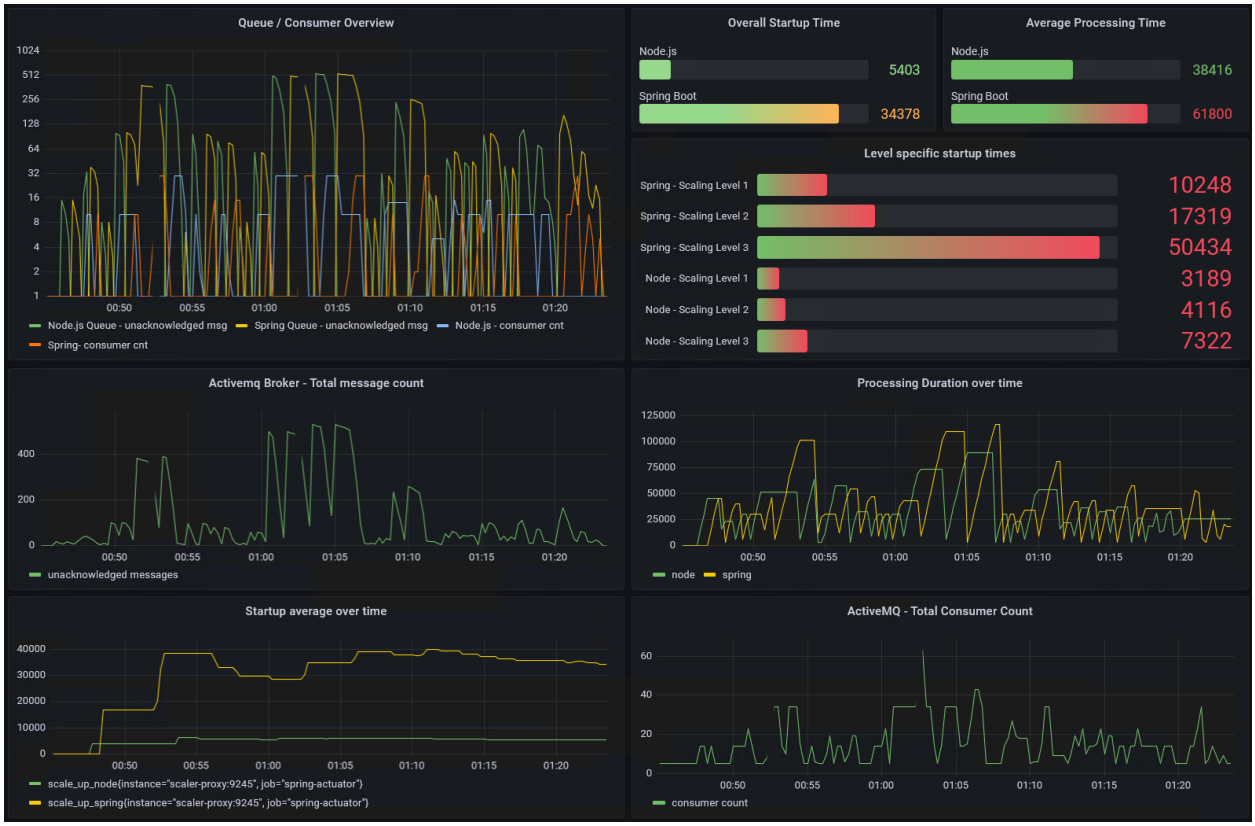


Abbildung 5.2: Grafana Dashboard

5.1.2 Node.js

5.1.3 Spring Boot

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
```

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

```
CB0=1
CB1=5
CB2=10
CB3=30
```

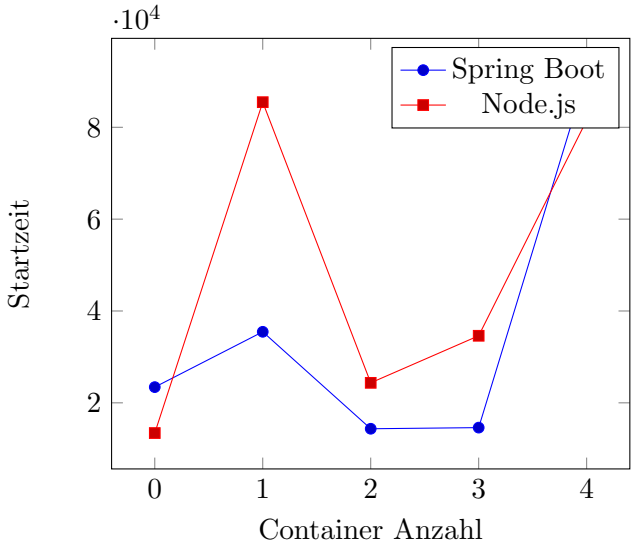
QL3 $QB2 < \overline{MC}$	UP *abs(CB0 – CB3)*	UP *abs(CB1 – CB3)*	UP *abs(CB2 – CB3)*	OK _
QL2 $QB1 < \overline{MC} \leq QB2$	UP *abs(CB0 – CB2)*	UP *abs(CB1 – CB2)*	OK _	DOWN *abs(CB2 – CB3)*
QL1 $QB0 < \overline{MC} \leq QB1$	UP *abs(CB0 – CB1)*	OK _	DOWN *abs(CB1 – CB2)*	DOWN *abs(CB1 – CB3)*
QL1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

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.

footnote mit link zu gatling und jmeter downloa seite

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.

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 der 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
  - 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

am Ende schauen ob genügend Zeit um noch die Verlässlichkeit des Systems zu testen. Stichwort Chaos Monkey

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.

- 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

nochmal schauen ob Überleitung noch stimmt wenn Implementierungsteil geschrieben wurde...

Verweis auf Anhang mit allen Tabellen für Metrikinformationen

folgende Stichpunkte entfernen

6.1 Ergebnisse

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

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)*.

Tabelle 6.1: Gegenüberstellung der Latenzzeiten

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

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.

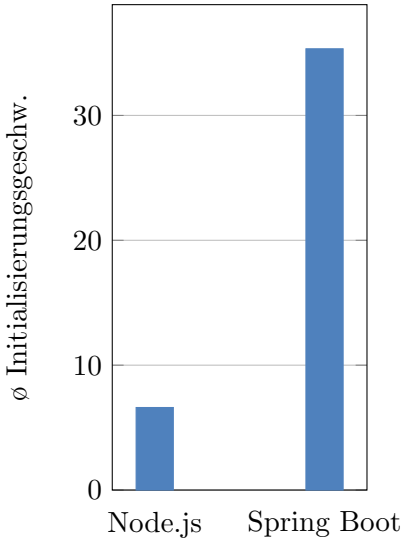


Abbildung 6.1: Startzeit Serviceübersicht

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

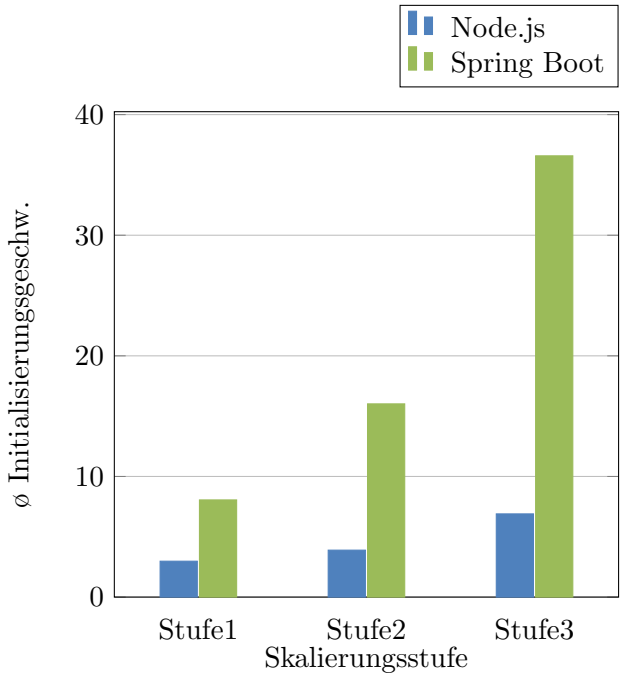
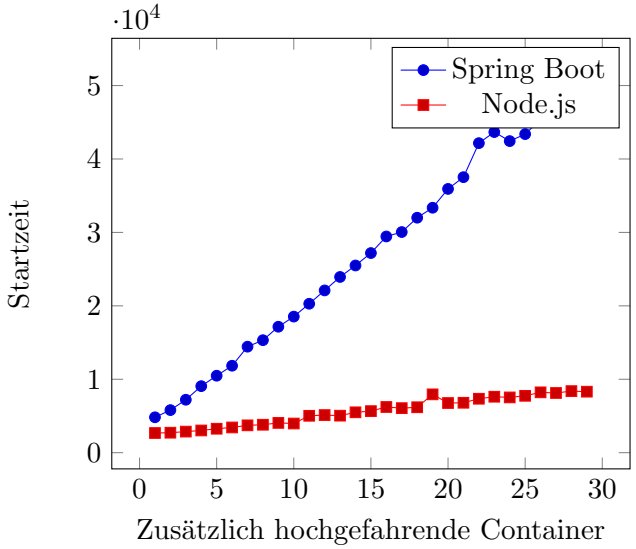


Abbildung 6.2: Startzeit Container - Stufenweise

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: Startzeit Container - Anzahl spezifisch



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.

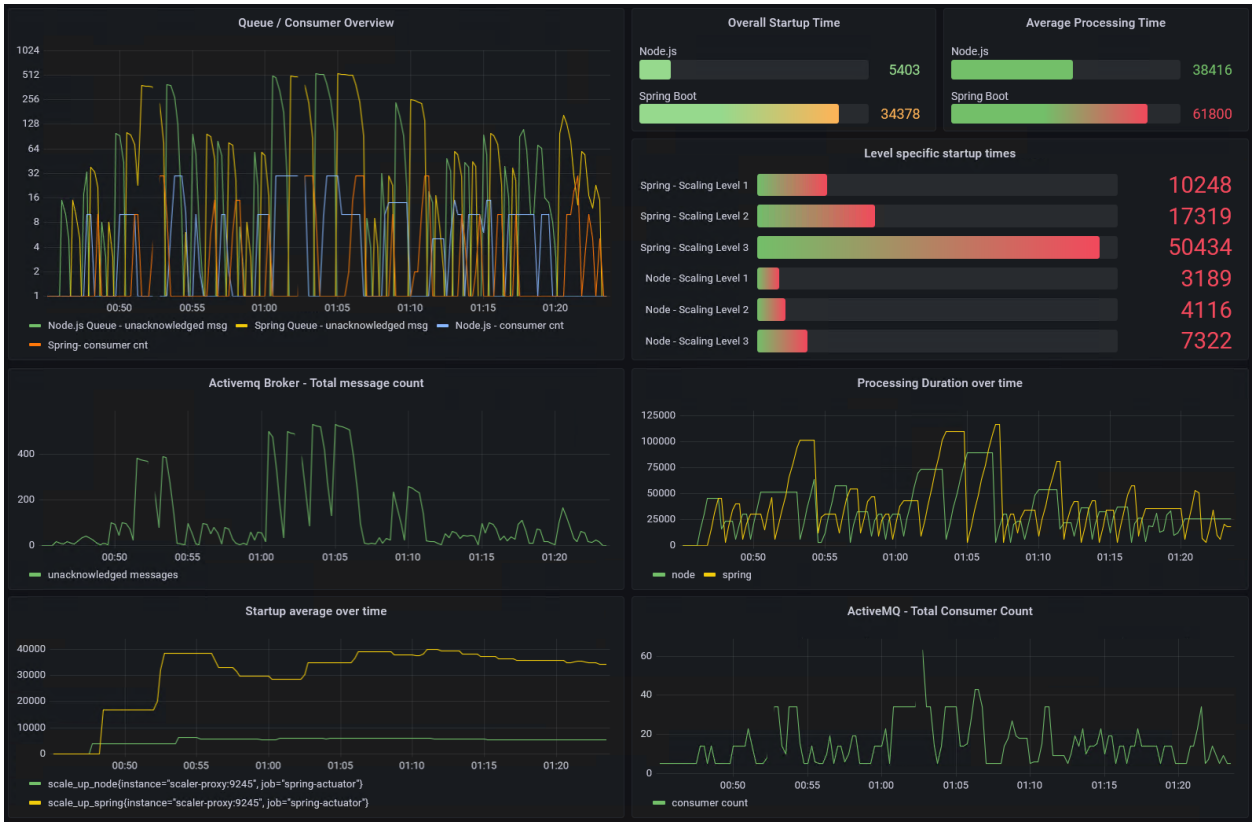


Abbildung 6.4: Grafana Dashboard

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).

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

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.

[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.

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

[Knu98] Donald E. Knuth. *The Art of Computer Programming, Volume 3: (2Nd Ed.) Sorting and Searching.* Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1998.

[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.

[Tur14] J. Turnbull. *The Docker Book: Containerization Is the New Virtualization.* James Turnbull, 2014.

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