

Performancevergleich einer NodeJS-API Bereitstellung auf IBM Cloud Code
Engine mit der Bereitstellung auf IBM Virtual Server for VPC anhand von
HTTP-Requests

2. Projektarbeit

vorgelegt am 22. November 2021

Fakultät Wirtschaft
Studiengang Wirtschaftsinformatik
Kurs WWI2019E

Von
Luca Alexander Weissbeck

Betreuer in der Ausbildungsstätte:

IBM Deutschland GmbH

Josip Ledic

Frontend Entwickler Code Engine



DHBW Stuttgart:

Dr. Robert Henjes

Inhaltsverzeichnis

Abkürzungsverzeichnis	IV
Abbildungsverzeichnis	V
1 Einleitung	1
1.1 Motivation	1
1.2 Problemdefinition und Problemabgrenzung	2
1.3 Aufbau und Methodik	3
2 Theoretische Grundlagen	4
2.1 Funktionsweise Virtual Server	4
2.2 Funktionsweise Serverless Computing	5
2.3 Application Programming Interface	8
2.3.1 Definition und Verwendungszweck	8
2.3.2 Hyper Text Transfer Protocol	8
2.3.3 Representation State Transfer	10
2.3.4 Funktionsweise NodeJS	11
2.4 API Performancetesting	14
2.4.1 JMeter als Software für Performancetests	14
2.4.2 Performanceteststrategien	16
3 Versuchsaufbau	18
3.1 Konfigurationserläuterung der Bereitstellungsalternativen	18
3.2 API & API Endpunkt	21
3.3 Vorstellung der Testinfrastruktur	23
4 Versuchsdurchführung	24
4.1 Baselinetest	24
4.2 Loadtest	28
4.3 Stresstest	30
4.4 Verursachte Kosten	34
5 Schlusskapitel	35
5.1 Fazit	35

5.2 Ausblick	35
Literaturverzeichnis.....	44

Abkürzungsverzeichnis

API	=	Application Programming Interface
CRUD	=	Create, Read, Update, Delete
FaaS	=	Function as a Service
HTTP	=	Hyper Text Transfer Protocol
IaaS	=	Infrastructure as a Service
JSON	=	Java Script Object Notation
OS	=	Operating System
PaaS	=	Platform as a Service
RAM	=	Random Access Memory
REST	=	Representational State Transfer
URI	=	Uniform Resource Identifier
URL	=	Uniform Resource Locator
vCPU	=	virtual CPU

Abbildungsverzeichnis

Abb. 1: Suchverlauf Trend "Serverless".....	1
Abb. 2: Aufbau eines Virtual Server.....	4
Abb. 3: Virtual Machine vs. Serverless	6
Abb. 4: Code Engine Architektur	7
Abb. 5: URL Aufbau	9
Abb. 6: HTTP-Anfrage Aufbau	9
Abb. 7: Beispielhaftes JavaScript Programm	12
Abb. 8: Output des beispielhaften Programms Synchron vs. Asynchron	12
Abb. 9: JMeter Threadgroup Konfiguration.....	15
Abb. 10: JMeter HTTP-Anfrage Konfiguration	15
Abb. 11: JMeter-Listener am Beispiel der Ergebnistabelle.....	16
Abb. 12: Code Implementation Bcrypt-Hashing.....	21
Abb. 13: HTTP-Anfragenzeit in Abhängigkeit der Saltrounds.....	22
Abb. 14: Baselinetest – Erfolgreiche HTTP-Anfragen & Durchschnittliche Antwortzeit.....	25
Abb. 15: Boxplot Verteilung der HTTP-Antwortzeiten im Baselinetest	26
Abb. 16: Baselinetest Localhost Wiederholung – Erfolgreiche HTTP-Anfragen & Durchschnittliche Antwortzeit.....	27
Abb. 17: Loadtest – Erfolgreiche HTTP-Anfragen & Durchschnittliche Antwortzeit.....	29
Abb. 18: Loadtest - Durchschnittlicher Mehraufwand pro zusätzlichen Nutzer	29
Abb. 19: Stresstest - Erfolgreiche HTTP-Anfragen & Durchschnittliche Antwortzeit	32
Abb. 20: Stresstest - Durchschnittlicher Mehraufwand pro zusätzlichen Nutzer	32
Abb. 21: Stresstest - Bestimmtheitsmaß	34

1 Einleitung

1.1 Motivation

Heutzutage stehen viele Entwickler vor der Wahl, wie sie ihre entwickelte Applikation möglichst effizient bereitstellen können. Sie werden vor die Wahl gestellt, entweder einen traditionellen Server zu verwenden oder eine neuartige *serverless* Bereitstellung zu wählen. Serverless Bereitstellungen stellen eine Revolution im *Cloudcomputing* Bereich dar, mit enormen Wachstumspotenzial.¹ Dass *serverless* Bereitstellungen längst keinen Nischenbereich mehr belegen, lässt sich anhand eines Blicks auf die Google Trends des Begriffs *serverless* belegen.

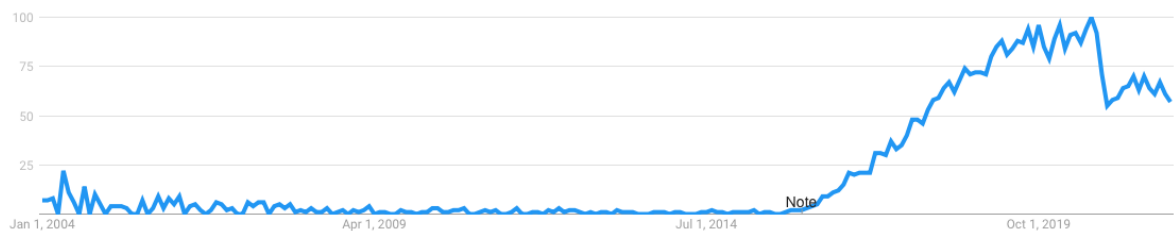


Abb. 1: Suchverlauf Trend "Serverless"²

In Abbildung 1 ist deutlich zu erkennen, dass seit Januar 2016 der Trend nach *serverless* Suchanfragen sprunghaft anstieg und im September 2020 erneut ein wenig abflachte. Erste *serverless* Technologien fanden ihren Anfang bereits 2014 mit der Einführung von AWS Lambda, ein *Function-as-a-Service* (FaaS) Service, während einer AWS re:Invent Vorstellung.³

Einen Grund für diesen Anstieg an Aufmerksamkeit gegenüber *serverless* Bereitstellungen liefert die IT-Applikationsverlagerung vieler Unternehmen zu Containern oder *Microservices*.⁴ Vorteile, wie die Verlagerung von Infrastrukturmanagement, das ehemals in der Verantwortung der Entwickler lag, auf den *Cloudprovider*, ist nur einer der vielen Vorteile dieser neuartigen Technologie.

Bekannte Unternehmen verwenden bereits aktiv *serverless* Technologien im Einsatz. So konnte beispielsweise Coca-Cola 65% ihrer IT-Kosten einsparen mit dem Einsatz einer solchen Technologie für automatisierte Verkaufsautomaten und dem Coca-Cola Treueprogramm.⁵ Andere Beispiele beinhalten die Bank „Santander“ oder die Reisewebsite „Expedia“,

¹ Vgl. Castro u. a. 2019, S. 44

² Enthalten in: Google o.J.a

³ Vgl. Castro u. a. 2019, S. 47

⁴ Vgl. Baldini u. a. 2017, S. 1

⁵ Vgl. Amazon Web Services 2020

die ebenfalls erhebliche Kosteneinsparungspotenziale durch *serverless Computing* erreichen konnten.⁶

1.2 Problemdefinition und Problemabgrenzung

Während sich Vorteile wie Kosteneinsparungen oder das Entlasten von Entwicklern ausschließlich positiv anhören, so müssen dennoch weitere Faktoren berücksichtigt werden, bevor eine Entscheidung über die Auswahl der Bereitstellungsmöglichkeit getroffen wird.

Ein wichtiger Faktor ist die Performance der jeweiligen Alternative. In den häufigsten Fällen besitzt eine Enterpriseapplikation bestimmte Performancekriterien, die erfüllt werden müssen. Die Performancemetrik wirkt sich stark auf die Benutzerfreundlichkeit der Applikation aus. Laut Scott können Wartezeiten über 15 Sekunden bereits dazu führen, dass Nutzer die Applikation oder Website verlassen.⁷

Möchte man beispielsweise eine Applikation von einem Virtual Server auf ein *serverless* Angebot migrieren, so stellt sich die Frage, ob dieselbe Konfiguration im *serverless* Angebot gewählt werden sollte wie im ursprünglichen Virtual Server. Daraus ergibt sich die Forschungsfrage für diese Praxisarbeit „Welche der beiden Bereitstellungsoptionen, Virtual Server und Serverless, gemessen am Beispiel von IBM Code Engine und IBM Virtual Server mittels einer beispielhaften REST-API, erzielen bei gleicher Belastung und derselben Konfiguration die bessere Performance?“

Die Relevanz dieser Forschungsfrage, lässt sich aus relevanter Literatur ableiten. So schreiben unter anderem Castro und Ishakian in einem ihrer Artikel aus Dezember 2019, dass die Performanceevaluierung von *serverless* Angeboten ein aktuell relevantes Thema in der Forschung darstellt.⁸ Darüberhinaus stellen Castro und Baldini als offene Forschungsprobleme ihrer Arbeit „Serverless Computing: Current Trends and Open Problems“ die Fragen: „What are boundaries of *serverless*?“, sowie „How does it relate to other models such as SaaS and MBaaS?“⁹. Speziell die letzte Frage, soll im Rahmen eines Vergleichs des Virtual Servers, als Beispiel für Infrastructure-as-a-Service (IaaS), und *serverless* Computing, als Beispiel für Platform-as-a-Service (PaaS), beantwortet werden.

Es lässt sich die Hypothese aufstellen, dass der Virtual Server möglicherweise eine bessere Performance aufweist als die *serverless* Alternative bei derselben Belastung. Zugrunde liegt dieser Hypothese, dass jegliche IBM Cloud Code Engine Applikation bereits in einem produktionsfertigen Standard bereitgestellt wird. So müssen bei dem Virtual Server noch zusätzliche

⁶ Vgl. Castro u. a. 2019, S. 49

⁷ Vgl. Scott 2010, S. 2ff.

⁸ Vgl. Castro. u. a. 2019, S. 50

⁹ Baldini u.a. 2017, S. 16

Konfigurationen getätigt werden, um eine ausreichende Netzwerksicherheit zu gewährleisten, bevor die Applikation für die Öffentlichkeit bereitgestellt werden kann. Code Engine beinhaltet bereits derartige Sicherheitsmechanismen im Hintergrund, welche möglicherweise die Performance bei derselben Konfiguration entscheidend negativ beeinflussen könnten.

1.3 Aufbau und Methodik

Um die erwähnte Forschungsfrage beantworten zu können, wird zunächst die Basis in der Theorie gelegt. Somit wird zuerst auf den Virtual Server und die *serverless* Bereitstellungsmöglichkeit genauer eingegangen, mit einem speziellen Fokus auf die jeweilige technische Funktionsweise. Im Anschluss wird das Konzept einer *Application-programming-Interface* (API) genauer betrachtet. Zu diesem Kapitel gehört die Erklärung von *Hyper-Text-transfer-Protocol* (HTTP), *Representational-State-Transfer* (REST) und NodeJS. Daraufhin wird der API-Performancetest aus theoretischer Sicht betrachtet, inklusive der verschiedenen Testverfahren, sowie der Performancetestapplikation JMeter. Des Weiteren werden die gewählten Konfigurationsoptionen der beiden Bereitstellungsmöglichkeiten, sowie die konkrete API und der entsprechend getestete Endpunkt, vorgestellt und erklärt. Bevor die Performancetests durchgeführt werden, muss die Testinfrastruktur betrachtet und erklärt werden.

Für den Performancevergleich der beiden Bereitstellungsmöglichkeiten wurde sich in dieser Arbeit, im Rahmen der wissenschaftlichen Methode, für die quantitative Methode des Tests entschieden, wie sie von Bortz und Döring im Buch „Forschungsmethoden und Evaluation“ beschrieben ist.¹⁰ Mit Fokus auf API-Testing gibt es besondere Teststrategien, die sich in der Literatur durchgesetzt haben. Diese werden in einem späteren Kapitel im Detail beschreiben. Mittels der durchgeführten API-Performancetests werden möglichst viele Datenpunkte gesammelt, aus denen im Anschluss ein Resultat in Bezug auf die Forschungsfrage deduziert werden kann. Ausgehend von den Experimentresultaten wird schließlich noch ein Fazit gezogen, sowie ein Ausblick für zukünftige Forschung, und Implikationen für Theorie und Praxis gegeben.

¹⁰ Vgl. Bortz/Döring 2006, S. 189

2 Theoretische Grundlagen

2.1 Funktionsweise Virtual Server

Virtualisierung ist die zugrundeliegende Technik, die für die Entstehung des Virtual Servers verantwortlich ist. Jain und Choudhary definieren die Virtualisierung als „abstraction of computer resources“¹¹. Die grundlegende Aufgabe der Virtualisierung ist, eine Separation des Betriebssystems von der Hardware des Computers herzustellen. Mit Hilfe von Virtualisierung ist es möglich einen physischen Server in viele, kleinere virtuelle Server aufzuteilen. Andererseits ist es ebenso möglich viele virtuelle Server in eine einzelne, stärkere virtuelle Instanz zu kombinieren¹². Bevor explizit auf die Funktionsweise eines Virtual Servers eingegangen wird und welche Rolle dort die Virtualisierung spielt, wird zunächst die Definition eines Virtual Servers betrachtet. Es gibt keine einheitliche Definition, da diese je nach Literatur sich leicht unterscheiden kann. Eine allgemein akzeptierte Definition bietet Google, einer der größten Cloudanbieter. Google definiert den Virtual Server wie folgt: „A virtual server re-creates the functionality of a dedicated physical server. It exists transparently to users as a partitioned space inside a physical server. Virtualizing servers makes it easy to reallocate resources and adapt to dynamic workloads.“¹³. Andere Cloud Anbieter wie Microsoft Azure stimmen mit dieser Definition überein und ergänzen, dass Virtual Server im Grunde nur als Code existieren.¹⁴

Zudem setzt sich der Virtual Server vom Virtual PC insofern ab, dass dieser, anders als der Virtual PC, in der Regel nicht für den Desktopnutzen geeignet ist und häufig spezifische Serverbetriebssysteme verwendet.¹⁵

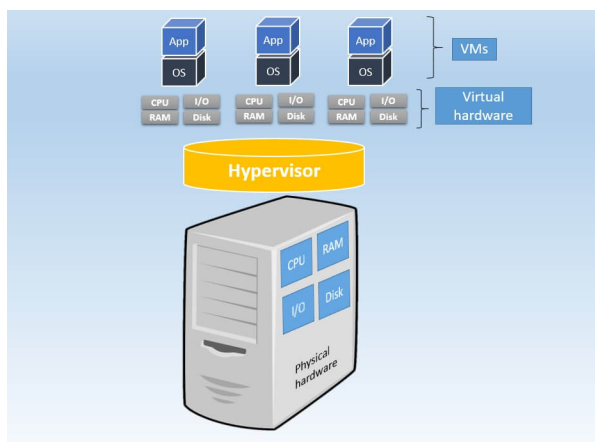


Abb. 2: Aufbau eines Virtual Server¹⁶

Nutzern eines Virtual Servers wird Speicherplatz, sowie Arbeitsspeicher und CPU nach Belieben zugewiesen. Der Aufbau eines Virtual Servers ist in Abbildung 2 dargestellt. Zu sehen

¹¹ Jain/Choudhary 2016, S.1

¹² Vgl. Jain/Choudhary 2016, S.1

¹³ Google o.J.b, S.1

¹⁴ Vgl. Microsoft o.J., S.1

¹⁵ Vgl. Davis 2005, S.5

¹⁶ Enthalten in: Reed 2019

sind unter anderem der sogenannte *Hypervisor*, welche eine Abstraktionsschicht darstellt, um mit der physischen Hardware zu interagieren. Der *Hypervisor* dient somit als Virtualisierungsschicht zwischen Hardware und den individuellen virtuellen Servern und ist zudem dafür verantwortlich, die angeforderten Hardwareressourcen entsprechend zu verteilen.¹⁷ Die zugrundeliegende physische Hardware wird als *Host* bezeichnet. In Abbildung 2 ist ebenfalls deutlich zu sehen, dass jeder Nutzer und somit jede Virtual Machine, ihren eigenen Anteil an CPU, I/O, *Random-access-Memory* (RAM), und Speicherplatz besitzt. Darüber hinaus kann jede Virtual Machine ein unterschiedliches *Operatingssystem* (OS) besitzen.¹⁸ Ein Großteil der *Cloudprovider* bieten dem Nutzer an eben solche Eigenschaft nach Belieben zu konfigurieren.

2.2 Funktionsweise Serverless Computing

Die Bereitstellung von Services auf *serverless Computeressourcen* ist eine neuartige Technologie, die sich ständig weiterentwickelt. Die Technologie bietet Entwicklern die Möglichkeit, Applikationen in der Cloud bereitzustellen, ohne vorher spezifische *Computeressourcen* festgelegt zu haben. Dies reduziert die *Time-to-deploy* enorm, also die Zeit Code bereitzustellen.¹⁹ Zudem kann eine *serverless* Architektur die IT-Infrastrukturkosten erheblich verringern. So konnten Adzic und Chatley die Bereitstellungskosten für eine beispielhafte Applikation mittels AWS Lambda, eine *serverless* Technologie von dem Cloudanbieter Amazon Web Services, um 66% bis 95% reduzieren.²⁰

Die Bereitstellung auf *serverless* Architektur wird zudem durch die zunehmende Containerisierung von Software vorangetrieben. Heutzutage bestehen viele Architekturen aus einer Vielzahl sogenannter *Microservices*. Hasselbring und Steinacker beschreiben *Microservices* als „small services that may be deployed and scaled independently of each other“²¹. Diese werden häufig in einem oder mehreren Containern ausgeführt. Einer der bekanntesten Anbieter solcher Containerisierungstechnologien lautet namentlich „Docker“.²²

Rajan sieht die *serverless* Architektur als nächste Evolution des *Cloudcomputings* und als Nachfolger des Virtual Servers an.²³ Da in einer *serverless* Infrastruktur alle Applikationen innerhalb eines Containers laufen, ist die Ressourcenbereitstellung grundsätzlich schneller als in einer virtuellen Maschine. Innerhalb einer solchen Infrastruktur wird nicht nur die Hardware geteilt, wie in einem Virtual Server, sondern auch das Betriebssystem, sowie die

¹⁷ Vgl. Jain/Choudhary 2016, S. 2

¹⁸ Vgl. Malhotra u. a. 2014, S. 1

¹⁹ Vgl. Baldini/Castro 2018, S. 1f.

²⁰ Vgl. Adzic/Chatley 2017, S. 1

²¹ Hasselbring / Steinacker 2017, S. 1

²² Vgl. Perez u. a. 2016, S. 1

²³ Vgl. Rajan 2018, S. 88f.

Laufzeitumgebung.²⁴ Dies wird besonders ersichtlich mit Blick auf Abbildung 3, in welcher die Kernbestandteile einer Virtual Machine der *serverless* Alternative gegenübergestellt ist.

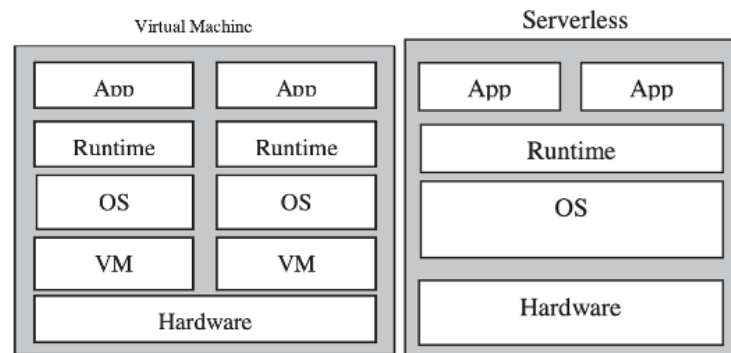


Abb. 3: Virtual Machine vs. Serverless²⁵

Deutlich zu sehen in Abbildung 3 ist, dass in der *serverless* Bereitstellungsmöglichkeit weitaus mehr Ressourcen unter diversen Applikationen geteilt werden können. Dies ermöglicht den Cloudanbietern ihre *Computerressourcen* effizienter einzusetzen. Erwähnenswert ist zudem die spezielle Art des *serverless Computing* namens „Function-as-a-Service“ (FaaS). Diese ermöglicht Entwicklern lediglich einzelne Funktionen auszuführen, unabhängig von der gesamten Applikation.²⁶

Ein wesentlicher Vorteil, den *serverless* Angebote bieten, ist die automatische Skalierung von *Computerressourcen* im Hintergrund. Der Nutzer zahlt nur für die Ressourcen, die er verwendet. Eine sogenannte *Pay-as-you-use* Preisstruktur. Dies hat zur Folge, dass Unternehmen kurzzeitig ihre Infrastruktur enorm skalieren können, ohne die Infrastrukturkosten langfristig zu tragen. Besonders Unternehmen, deren IT-System Last von der Saison abhängig ist, profitieren daher von einer solchen Zahlungsweise.²⁷

Die konkrete zugrundeliegende Architektur, die eine *serverless* Bereitstellung ermöglicht, unterscheidet sich bei den diversen Cloudanbietern und ist häufig nicht einsehbar für den Nutzer. „Code Engine“, das aktuelle *serverless* Angebot des Cloudanbieters IBM, ermöglicht dem Nutzer seine Applikation entweder in Form eines Containers oder eines *Coderepositorys* anzugeben, wobei bei letzterer Möglichkeit ebenfalls ein Container für den Nutzer von Code Engine gebaut wird. Im Hintergrund laufen diese Container auf Knative, einem *open-source* Projekt, welches auf Kubernetes aufbaut und dieses um Funktionen erweitert, die unter anderem das Bereitstellen und Verwalten von *serverless* Applikationen auf Kubernetes ermöglichen.²⁸ Grundsätzlich lassen sich zwei verschiedene Arten von *Workloads* auf Code Engine

²⁴ Vgl. Rajan 2018, S. 89

²⁵ Mit Änderungen entnommen aus: Rajan 2019, S. 88

²⁶ Vgl. Baldini/Castro 2017, S. 2

²⁷ Vgl. Hellerstein/Faleiro 2019, S. 1

²⁸ Vgl. Kaviani u. a. 2019; S. 2; RedHat 2019

bereitstellen. Die erste Option ermöglicht die Bereitstellung von interaktiven HTTP angetriebenen Applikationen und *Microservices*, wie beispielsweise eine REST-API. Die alternative Option stellt die Bereitstellung von Batchjobs dar.²⁹ Für den weiteren Verlauf dieser Arbeit ist lediglich der erste *Workload* relevant, also HTTP angetriebene Webapplikationen. IBM argumentiert, dass Code Engine die Hürden, eine Applikation bereitzustellen, wesentlich verringert im Vergleich zu einem traditionellen, selbst verwalteten Kubernetescluster. So muss sich der Entwickler in einem alternativen Kubernetescluster unter anderem um die Netzwerkkonfiguration, Infrastruktur und die automatische Skalierung zusätzlich kümmern. Code Engine übernimmt diese Aufgaben vollständig für den Entwickler.³⁰

Neben Knative und Kubernetes als Basis, verwendet Code Engine viele weitere *open-source* Technologien wie zum Beispiel Tekton und Shipwright. Knative erleichtert das Verwalten von Webapplikation in Kubernetes, indem es die bereitgestellte Applikation automatisch skaliert, abhängig von der Anzahl der eingehenden HTTP-Anfragen. Tekton und Shipwright sind für das automatische Erstellen von *Containerimages* anhand von Quellcode zuständig.³¹

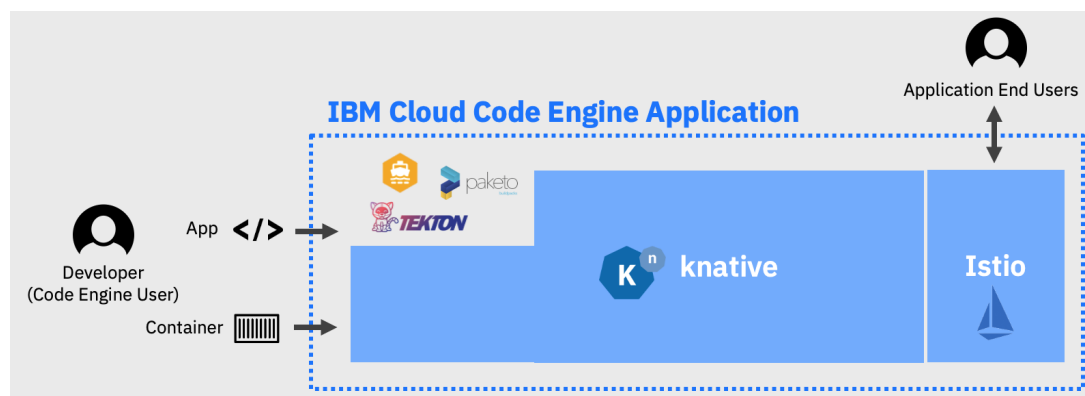


Abb. 4: Code Engine Architektur³²

In Abbildung 4 ist zu sehen, wie der Nutzer mit einer beispielhaften Applikation interagieren kann. Er stellt das *Containerimage* entweder direkt oder in Form von Quellcode bereit und kann anschließend mit seiner Webapplikation interagieren. Neben Tekton, Shipwright und Knative sind ebenfalls weitere Technologien dargestellt, welche Code Engine verwendet. Diese sind jedoch nicht in der Code Engine Dokumentation aufgeführt und werden von dem Kontext dieser Arbeit explizit ausgegrenzt.³³

²⁹ Vgl. Zhuang 2021a

³⁰ Vgl. Zhuang u. a. 2021b

³¹ Vgl. Zhuang u. a. 2021b

³² Enthalten in: Zhuang u. a. 2021b

³³ Vgl. Zhuang u. a. 2021b

2.3 Application Programming Interface

2.3.1 Definition und Verwendungszweck

Meng und Steinhardt definieren den Nutzen einer API darin, *Services* oder Daten einer Softwareapplikation bereitzustellen, mittels vordefinierter Ressourcen. Diese Ressourcen beinhalten beispielsweise Methoden, Objekte oder *Uniform-Resource-Identifiers* (URI).³⁴ Somit wird es anderen Applikationen ermöglicht auf Ressourcen wie Daten oder *Services* zuzugreifen, ohne die zugrundeliegende Codeinfrastruktur dieser Ressourcen selbst implementieren zu müssen. Dies führt zu einer Unabhängigkeit von Applikationen, sodass eine verteilte Softwarearchitektur ermöglicht wird. Daher stellen APIs häufig den Kern einer Softwarearchitektur heutzutage dar.³⁵

Zudem bieten Unternehmen vermehrt eigene APIs an, um beispielsweise eine Dienstleistung des Unternehmens für Entwickler zugänglich zu machen, ohne dabei die eigene Codeinfrastruktur zu veröffentlichen. Zum Beispiel ermöglicht die Google Maps API dem Entwickler in seiner eigenen Applikation einen Ort auf einer Karte anzuzeigen. Entwickler können somit diese Dienstleistungen eines anderen Unternehmens in ihre eigene Applikation einbauen. Der Autor des Buches „API Management“, Da, vergleicht APIs daher mit „windows to the code base“.^{36, 37}

2.3.2 Hyper Text Transfer Protocol

HTTP stellt das meistverwendete Netzwerkprotokoll, für die Gewährleistung der Kommunikation innerhalb des Internets, dar. Es handelt sich um ein *Client-Server* Protokoll, das auf dem TCP-Protokoll aufbaut, um Daten auszutauschen. Ein *Client* kann mittels HTTP, Daten jeglicher Art von einem Webserver abfragen. HTTP unterstützt Nutzer aus einem vielfältigen Anwendungsbereich, sei dies ein Internetbrowser, eine Handyapplikation oder sogar ein Navigationssystem im Auto. Diese Vielfalt an Einsatzmöglichkeiten wird dadurch ermöglicht, dass HTTP lediglich die Reihenfolge und das Format der zu übermittelnden Nachricht bestimmt, nicht jedoch das dafür nötige Programmierumfeld. Dem Nutzer und dem Server sind daher die Wahl des Programmierungsumfelds freigestellt.³⁸

Der *Client* kann bei dem Server Ressourcen mittels konkreter URIs beziehungsweise *Uniform-Resource-Locators* (URLs) anfragen. Hinter jeder URL / URI steht eine konkrete Ressource, etwa eine HTML Datei oder ein JPEG Bild. URIs sind breiter gefasst und müssen ungleich

³⁴ Vgl. Meng u. a. 2018, S. 296

³⁵ Vgl. Meng u. a. 2018, S. 296

³⁶ De 2017, S. 1

³⁷ Vgl. De 2017, S. 1

³⁸ Vgl. Bressoud/White 2020, S. 609 f.

Die Operation DELETE löscht eine Ressource. Darüber hinaus existieren weitere HTTP-Operationen, welche das Ausmaß dieser Arbeit jedoch überschreiten.⁴¹

2.3.3 Representation State Transfer

Bei der Architektur einer API gilt es bestimmte architektonische Grundlagen zu beachten. Ein solches architektonisches Regelwerk bietet unter anderem REST. Andere Regelwerke stellen beispielsweise „SOAP“ und „WSDL“ dar, welche von REST über die letzten Jahre vermehrt abgelöst wurden. Ein deutliches Beispiel für diese Ablösung ist die Umstellung auf REST Prinzipien von Firmen wie Yahoo, Google und Facebook.⁴²

Dabei stellt REST keinen spezifischen Standard, sondern lediglich einen architektonischen Stil, welcher auf weitverbreiteten Technologien sowie Protokollen basiert.⁴³ Mit Hilfe von REST können *Webservices* erstellt werden, die auf Ressourcen aufbauen. So wird mittels REST klar definiert, wie Ressourcenzustände adressiert werden und wie diese mittels HTTP transferiert werden können.⁴⁴ Die Ressource ist ein Schlüsselbegriff im REST-Design und beschreibt alles was namentlich benannt werden kann. *Representation* steht in REST für die Repräsentation einer Ressource, bei welcher die Repräsentation den derzeitigen oder gewollten Zustand der Ressource beschreibt.⁴⁵ Damit ein *Webservice* dem REST-Protokoll gerecht wird, muss dieser vier verschiedene Designprinzipien befolgen, auf welche im Folgenden eingegangen wird.

Zunächst dürfen ausschließlich HTTP-Methoden verwendet werden. Der Nutzer verwendet HTTP, um mit den Ressourcenzuständen zu interagieren. Die konkreten Interaktionswege folgen dem CRUD Konzept, welches bereits im vorherigen Teil erläutert wurde.⁴⁶

Heutzutage ist die Zustandslosigkeit, welche das zweite Designprinzip von REST darstellt, die Grundlage für eine skalierbare Applikation, um auch mit einer hohen Belastung umgehen zu können. Diese Last wird in einer beispielhaften Applikation mittels eines *Loadbalancers* auf ein *Cluster* von Servern verteilt. Die Anzahl der Server im *Cluster* kann auf der Cloud je nach Nachfrage hoch- oder runterskaliert werden. Diese Architektur gelingt allerdings nur dann, wenn die Anfragen der Nutzer vollständig sowie unabhängig voneinander, oder auch zustandslos sind. Nur wenn diese Voraussetzung erfüllt ist, kann der *Loadbalancer* die Nutzeranfragen auf verschiedene Server aufteilen. Die HTTP-Anfrage muss folglich alle nötigen Informationen für ein erfolgreiches Prozessieren der Anfrage auf dem Server enthalten.⁴⁷

⁴¹ Vgl. Rodriguez 2010, S. 2

⁴² Vgl. Rodriguez 2010, S. 1

⁴³ Vgl. Richards 2006, S. 633

⁴⁴ Vgl. Rodriguez 2010, S. 1

⁴⁵ Vgl. Jakl o.J., S. 4

⁴⁶ Vgl. Rodriguez 2010, S. 2

⁴⁷ Vgl. Rodriguez 2010, S. 4ff.

Das nächste Designprinzip von REST stellt die Adressierbarkeit. Dies bedeutet, dass jede angelegte Ressource einen eindeutigen, sogenannten *Resourceidentifier* besitzen muss, so dass zwischen Ressourcen unterschieden werden kann. Diese Aufgabe übernimmt innerhalb von REST die URI.⁴⁸ URIs ermöglichen zusammen mit den bereits definierten HTTP-Methoden auf die zugrundeliegenden Ressourcen einer API zuzugreifen.⁴⁹

Der Nutzer und der Server tauschen Daten mittels des *Body*s der HTTP-Anfrage aus. Es gilt, im Rahmen des letzten Designprinzips von REST darauf zu achten, dass das Format dieses *Body*s möglichst simpel und einfach zu lesen ist. Mögliche Datenformate sind beispielsweise XML oder das vermehrt an Popularität gewinnende JSON.⁵⁰

2.3.4 Funktionsweise NodeJS

NodeJS wurde 2009 erstmalig präsentiert und stellt eine Laufzeitumgebung für die, besonders für Webprogrammierung populäre Programmiersprache, JavaScript dar. Es basiert auf Googles *V8 Engine* und ermöglicht die Ausführung von JavaScript auf jeder Plattform, auf welcher auch NodeJS installiert werden kann. Aufgrund der Tatsache, dass JavaScript eine interpretierte Programmiersprache darstellt, ist sie im Vergleich zu einer Programmiersprache, bei welcher der Quelltext zuerst kompiliert wird, grundsätzlich ineffizienter. Googles *V8 Engine* kompiliert jedoch JavaScript zuerst zu Code auf Maschinenebene. Bei jeder Ausführung wird also Maschinencode ausgeführt, anstatt JavaScript interpretiert. Dies dient einer höheren Effizienz.⁵¹

NodeJS differenziert sich zudem von anderen *Webframeworks* oder Programmiersprachen durch die verwendete *single-threaded* Funktionsweise. Bei derartigen Infrastrukturen wird jeder eingehender HTTP-Anfrage ein neuer *Thread* zugeordnet. Dieser *Thread* wird so lange nicht freigegeben, bis die Anfrage bearbeitet wurde. Einen neuen *Thread* für jede Anfrage zu erstellen, kostet viele Ressourcen. Besonders da in Webapplikationen häufig die Bearbeitung einer HTTP-Anfrage viel Zeit kostet, aufgrund der Notwendigkeit beispielsweise auf eine Datenbank im Hintergrund zugreifen zu müssen. Stattdessen verwendet NodeJS nur einen einzelnen *Thread*, welcher die Anfragen asynchron bearbeitet. Das bedeutet, dass der *Thread* nicht darauf wartet bis die Anfrage bearbeitet wurde, sondern währenddessen bereits andere Anfragen bedient. Diese Vorgehensweise ist deutlich effizienter für Applikationen, die einen hohen *Webtraffic* besitzen, da somit auch Webapplikationen bedient werden können, die Millionen von gleichzeitigen Anfragen erwartet. Ein Nachteil, den der NodeJS Ansatz mit sich führt ist, dass NodeJS standardmäßig nur einen einzelnen CPU-Kern verwenden kann. Dies

⁴⁸ Vgl. Jalk o.J., S.4

⁴⁹ Vgl. Ong u. a. 2015, S. 211

⁵⁰ Vgl. Rodriguez 2010, S. 8

⁵¹ Vgl. Krol u. a. 2014, S. 2

bedeutet, dass die Anzahl der Kerne eines Servers keinen Einfluss auf die Performance der NodeJS Applikation haben. Es gibt jedoch Möglichkeiten, beziehungsweise *Libraries*, welche diesen Flaschenhals umgehen können.⁵²

Wie bereits im vorherigen Teil angesprochen ist NodeJS in der Lage Anfragen asynchron zu bearbeiten. Dies ist auch ein wesentlicher Bestandteil für die erhaltene Popularität von NodeJS in den vergangenen Jahren. Man spricht von *Non-blocking-asynchronous-Execution*. Ermöglicht wird eine solche Asynchronität in NodeJS mittels sogenannter *Callbackfunctions* und dem sogenannten *Eventloop*. Eine Operation bekommt eine solche *Callbackfunction* zugeordnet. Sobald die Operation asynchron bearbeitet wurde, wird dies von der *Eventloop* erfasst und anschließend die *Callbackfunction* aufgerufen. Ein Beispiel erleichtert das Verständnis dieser *Callbackfunctions*.⁵³

```
console.log('One');
console.log('Two');
setTimeout(function() {
  console.log('Three');
}, 2000);
console.log('Four');
console.log('Five');
```

Abb. 7: Beispielhaftes JavaScript Programm⁵⁴

Das in Abbildung 7 dargestellte Programm gibt in einer synchronen Programmiersprache eine unterschiedliche Ausgabe zurück, als in einer Programmiersprache mit einem asynchronen Ansatz. Grundsätzlich ist der Zweck dieses Programms fünf Konsolenausgaben zu erstellen. Die dritte Ausgabe hat den Unterschied, dass sie in einer *setTimeout* Funktion steht. Es wird also erst zwei Sekunden gewartet, bevor der Inhalt der *setTimeout* Funktion ausgeführt wird.

<u>Synchron</u>	<u>Asynchron</u>
One	One
Two	Two
... (2 second delay) ...	Four
Three	Five
Four	... (approx. 2 second delay) ...
Five	Three

Abb. 8: Output des beispielhaften Programms Synchron vs. Asynchron⁵⁵

⁵² Vgl Krol u. a. 2014, S. 3

⁵³ Vgl u.a. Sun/Schiavio 2019, S. 61

⁵⁴ Enthalten in: Krol u. a. 2014, S. 4

⁵⁵ Mit Änderungen entnommen aus: Krol u. a. 2014, S. 4

Zu sehen ist der Unterschied der beiden Ausführungsvarianten in Abbildung 8. In einer asynchronen Programmiersprache beziehungsweise Laufzeitumgebung wie NodeJS, wird der Inhalt der *setTimeout* Funktion zuletzt aufgerufen. Zurückzuführen ist dies auf den Aspekt, dass NodeJS nicht auf die Bearbeitung einer Operation wartet, sondern bereits die nächste Operation startet. In Abbildung 7 ist die Funktion, die als Parameter der *setTimeout* Funktion übergeben wird, als *Callbackfunktion* zu identifizieren. Diese wird allerdings erst nach zwei Sekunden aufgerufen. Innerhalb dieser zwei Sekunden verarbeitet NodeJS bereits weitere Operationen ab. Daher wird „Three“ zuletzt ausgegeben.

2.4 API Performancetesting

2.4.1 JMeter als Software für Performancetests

Für produktive API-Performancetests ist es wichtig die entsprechend richtige Software zu wählen, welche die Durchführung dieser Tests übernimmt. Häufig hängt die richtige Auswahl von der jeweiligen Situation ab. Zunächst einmal muss jedoch herausgestellt werden, dass sich API-Performancetests von herkömmlichen Tests insofern differenzieren, dass bei API-Performancetests nicht das Design oder die Nutzerfreundlichkeit einer Applikation getestet wird. Stattdessen dient ein solcher Test bereits in frühen Entwicklungszyklen einer Applikation, unvorhergesehene Abstürze eines Systems zu verhindern, indem die API unterschiedlich starken Belastungen ausgesetzt wird und das Verhalten der API analysiert wird.⁵⁶

Apache JMeter ist ein *open-source* Framework des Unternehmens „Apache“. Es bietet die Möglichkeit Webapplikationen beziehungsweise APIs unter starker Belastung automatisiert zu testen und diese Tests mithilfe von Graphen zu analysieren. Die Tests werden durchgeführt, indem JMeter HTTP-Anfragen simuliert und diese an die entsprechende API schickt. Zudem bietet JMeter die Möglichkeit HTTP-Anfragen nach Belieben zu konfigurieren. Es kann beispielsweise der Body einer HTTP-Anfrage konfiguriert werden oder Parameter, die für die Authentifizierung notwendig sind. Sogenannte *Listeners* können in einer JMeter-Testkonfiguration eingestellt werden, welche die Funktion haben das Ergebnis des Tests visuell mit Hilfe von Graphen oder Tabellen darzustellen. Ein wesentlicher Aspekt JMeters ist die Erweiterung des Programms durch Community Plug-Ins welche den Funktionsumfang erheblich erweitern können.⁵⁷

Für die Konfiguration eines Tests in JMeter muss unter anderem ein Testplan angelegt werden. Nevedrov beschreibt einen Testplan als „a sequence of operations JMeter will execute“⁵⁸. Ein Testplan muss mindestens eine *Threadgroup* beinhalten, die den Startpunkt des Tests festlegt, sowie alle anderen JMeter Elemente beinhaltet. Denn die *Threadgroup* kontrolliert alle *Threads*, die infolge des Tests von JMeter erstellt werden, um eine Belastung zu simulieren. Ein *Thread* entspricht hierbei einem simulierten Nutzer.⁵⁹

⁵⁶ Vgl. De 2017, S.153; Nevedrov 2006, S. 1

⁵⁷ Vgl. De 2017, S. 160; Nevedrov 2006, S. 1

⁵⁸ Nevedrov 2006, S. 1

⁵⁹ Kurniawan 2003, S. 2

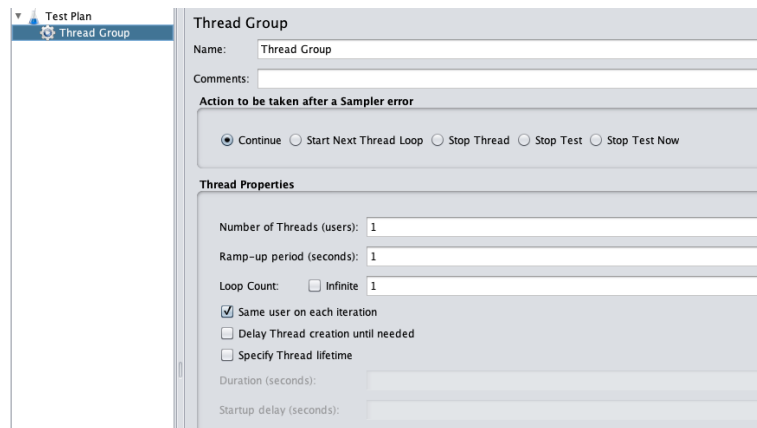


Abb. 9: JMeter Threadgroup Konfiguration

Abbildung 9 sind die Einstellungsmöglichkeiten einer *Threadgroup* zu entnehmen. Es kann der Name festgelegt werden, sowie das Standardverhalten bei Fehlern einer HTTP-Anfrage. Zudem können die Anzahl der *Threads* für den Testplan und die sogenannte *Ramp-up-Period* in Sekunden eingestellt werden. Letzteres beschreibt die Zeit, die JMeter benötigt, für die Erstellung aller eingestellten *Threads*. Sei beispielsweise die Anzahl der *Threads* auf 30 eingestellt und die *Ramp-up-Period* auf 10 Sekunden, so werden innerhalb von 10 Sekunden 30 Nutzer erstellt. Es lässt sich zudem mittels des Felds *Loopcount* einstellen, wie häufig der Test wiederholt werden soll oder ob der Test unendlich häufig wiederholt werden soll mittels der *Infinite* Checkbox.⁶⁰

Im nächsten Schritt lässt sich eine konkrete HTTP-Anfrage konfigurieren im Rahmen der *Threadgroup*.

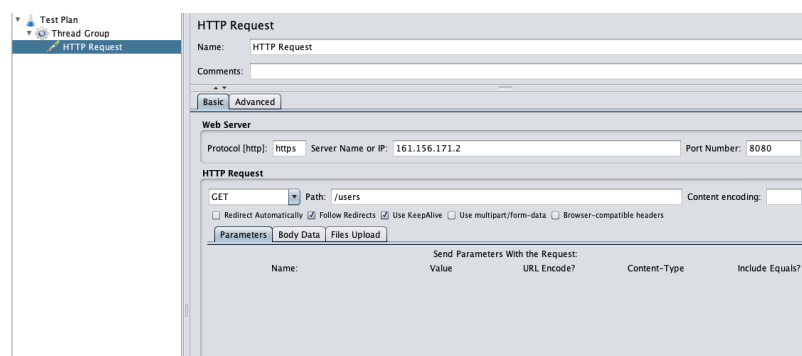


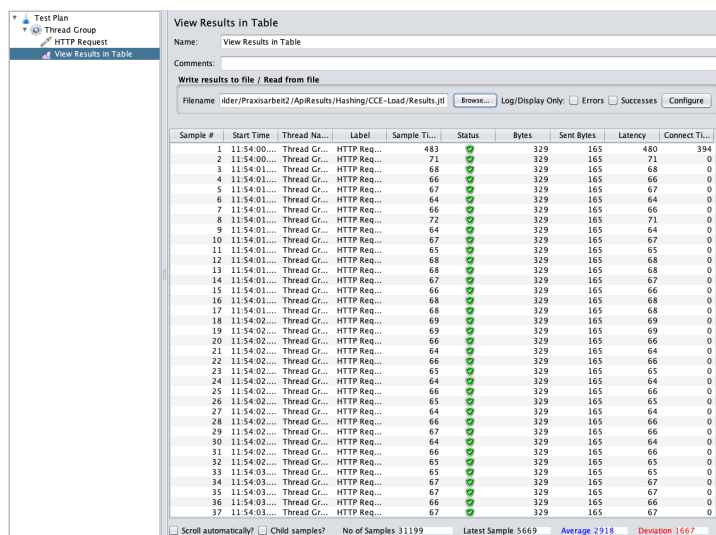
Abb. 10: JMeter HTTP-Anfrage Konfiguration

In Abbildung 10 ist eine solche HTTP-Anfragenkonfiguration zu sehen. Es lassen sich das Protokoll, die IP des Servers, sowie die Portnummer einstellen. Zudem lässt sich die HTTP-Methode und die URI der entsprechenden HTTP-Anfrage konfigurieren. Erwähnenswert sind

⁶⁰ Kurniawan 2003, S. 3

außerdem die zusätzlichen Konfigurationsmöglichkeiten, wie das Hinzufügen eines HTTP-Body oder das Hinzufügen von HTTP-Parametern. Diese sind für den weiteren Verlauf dieser Praxisarbeit jedoch nicht relevant.

Das letzte Element, welches einer *Testgroup* hinzugefügt werden sollte, ist ein sogenannter *Listener*. Kurniawan vergleicht einen JMeter-*Listener* essenziell zu einem Prüfbericht.⁶¹ JMeter bietet eine Vielzahl an *Listener*-Elementen an. Ein Beispiel stellt die Ergebnistabelle dar, welche in Abbildung 11 zu sehen ist.



The screenshot shows the 'View Results in Table' window in JMeter. It displays a table with columns: Sample #, Start Time, Thread Name, Label, Sample Time, Status, Bytes, Sent Bytes, Latency, and Connect Time. The table contains 37 rows of data, representing individual HTTP requests. The status column shows green checkmarks for successful requests. The bottom of the window shows summary statistics: No of Samples 31199, Latest Sample 5669, Average 2918, and Deviation 1667.

Sample #	Start Time	Thread Name	Label	Sample Time	Status	Bytes	Sent Bytes	Latency	Connect Time
1	11:54:00...	Thread Gr...	HTTP Req...	483	✓	329	165	480	394
2	11:54:00...	Thread Gr...	HTTP Req...	71	✓	329	165	71	0
3	11:54:01...	Thread Gr...	HTTP Req...	68	✓	329	165	68	0
4	11:54:01...	Thread Gr...	HTTP Req...	66	✓	329	165	66	0
5	11:54:01...	Thread Gr...	HTTP Req...	67	✓	329	165	67	0
6	11:54:01...	Thread Gr...	HTTP Req...	64	✓	329	165	64	0
7	11:54:01...	Thread Gr...	HTTP Req...	66	✓	329	165	66	0
8	11:54:01...	Thread Gr...	HTTP Req...	72	✓	329	165	71	0
9	11:54:01...	Thread Gr...	HTTP Req...	64	✓	329	165	64	0
10	11:54:01...	Thread Gr...	HTTP Req...	67	✓	329	165	67	0
11	11:54:01...	Thread Gr...	HTTP Req...	65	✓	329	165	65	0
12	11:54:01...	Thread Gr...	HTTP Req...	68	✓	329	165	68	0
13	11:54:01...	Thread Gr...	HTTP Req...	68	✓	329	165	68	0
14	11:54:01...	Thread Gr...	HTTP Req...	67	✓	329	165	67	0
15	11:54:01...	Thread Gr...	HTTP Req...	66	✓	329	165	66	0
16	11:54:01...	Thread Gr...	HTTP Req...	68	✓	329	165	68	0
17	11:54:01...	Thread Gr...	HTTP Req...	68	✓	329	165	68	0
18	11:54:02...	Thread Gr...	HTTP Req...	69	✓	329	165	69	0
19	11:54:02...	Thread Gr...	HTTP Req...	69	✓	329	165	69	0
20	11:54:02...	Thread Gr...	HTTP Req...	66	✓	329	165	66	0
21	11:54:02...	Thread Gr...	HTTP Req...	64	✓	329	165	64	0
22	11:54:02...	Thread Gr...	HTTP Req...	66	✓	329	165	66	0
23	11:54:02...	Thread Gr...	HTTP Req...	65	✓	329	165	65	0
24	11:54:02...	Thread Gr...	HTTP Req...	64	✓	329	165	64	0
25	11:54:02...	Thread Gr...	HTTP Req...	66	✓	329	165	66	0
26	11:54:02...	Thread Gr...	HTTP Req...	65	✓	329	165	65	0
27	11:54:02...	Thread Gr...	HTTP Req...	64	✓	329	165	64	0
28	11:54:02...	Thread Gr...	HTTP Req...	66	✓	329	165	66	0
29	11:54:02...	Thread Gr...	HTTP Req...	67	✓	329	165	66	0
30	11:54:02...	Thread Gr...	HTTP Req...	64	✓	329	165	64	0
31	11:54:02...	Thread Gr...	HTTP Req...	66	✓	329	165	66	0
32	11:54:02...	Thread Gr...	HTTP Req...	65	✓	329	165	65	0
33	11:54:03...	Thread Gr...	HTTP Req...	65	✓	329	165	65	0
34	11:54:03...	Thread Gr...	HTTP Req...	67	✓	329	165	67	0
35	11:54:03...	Thread Gr...	HTTP Req...	67	✓	329	165	67	0
36	11:54:03...	Thread Gr...	HTTP Req...	66	✓	329	165	66	0
37	11:54:03...	Thread Gr...	HTTP Req...	67	✓	329	165	67	0

Abb. 11: JMeter-*Listener* am Beispiel der Ergebnistabelle

Jede einzelne HTTP-Anfrage ist hier aufgelistet, zusammen mit relevanten Informationen, wie der Latenz oder der Anzahl der gesendeten Bytes pro Anfrage. Andere *Listener* beinhalten beispielsweise einen Graphen, der die *Responsetime* der HTTP-Anfragen oder die Anzahl der gestarteten *Threads* über Zeit darstellt.

2.4.2 Performanceteststrategien

Es gibt keine Vorschrift wie ein API-Performancetest auszusehen hat, allerdings beinhaltet eine weitverbreitete Teststrategie, die im Buch „API Management“ des Autors De erläutert wird, die Durchführung von drei verschiedenen, klar definierten Tests. Diese beinhalten einen *Baselinetest*, *Loadtest* und *Stresstest*, die im Folgenden genauer erläutert werden.⁶² Diese Tests lassen sich ebenfalls, wenn auch etwas abgeändert, in anderen Literaturwerken erneut auffinden, etwa wie in einem Artikel des offiziellen RedHat Developer Forums, sowie in der offiziellen SoapUI Dokumentation.⁶³ Rodrigues nennt, neben den bereits genannten Tests,

⁶¹ Vgl. Kurniawan 2003, S. 4

⁶² Vgl. De 2017, S. 158 - 162

⁶³ Vgl. Guerrero 2015, S. 1; Soap UI o.J.

weitere Tests wie den *Soak-/Endurancetest* oder den *Failover-/Resiliencetest*. Zudem erwähnt Molyneaux zusätzlich noch einen *Isolationtest*, sowie einen *Smoketest*. Diese Tests würden jedoch das Ausmaß dieser Arbeit übersteigen.

Der *Baselinetest* analysiert das Verhalten einer API während einer durchschnittlichen Belastung. Wenn möglich, sollte hier eine Belastung gewählt werden, die im Einsatz der API zu erwarten ist, also die reale Einsatzlast möglichst genau repräsentiert.⁶⁴ Somit kann die durchschnittliche *Responsetime* der API bei zu erwartender Belastung festgestellt werden.⁶⁵ Häufig sind für diesen Test klare Vorgaben gegeben, für unter anderem die durchschnittliche Antwortzeit, die Datenrate, sowie die Verfügbarkeit.⁶⁶

Während des *Loadtests* wird, im Unterschied zu dem *Baselinetest*, die Last während des Tests erhöht. Dies simuliert eine steigende API-Belastung, die durch Stoßzeiten der eingesetzten Applikation ausgelöst werden kann. Es kann somit analysiert werden, wie die API auf zunehmende Belastung reagiert. Es gilt zu beachten, dass im Rahmen des *Loadtests* jedoch nicht der *Breakingpoint* der API gefunden werden soll. Das bedeutet, dass keine *Timeouts* oder Fehler durch die API entstehen sollen.⁶⁷

Im *Stresstest* gilt es nun den *Breakingpoint* der API herauszufinden, indem die Belastung kontinuierlich erhöht wird, bis Fehler von der API zurückgegeben werden. Somit kann festgestellt werden, mit wie vielen HTTP-Anfragen die API maximal, gleichzeitig umgehen kann.⁶⁸ Laut Molyneaux ist ein *Stresstest* aber auch dann erfolgreich, wenn dieser eine vordefinierte Schwelle für beispielsweise die HTTP-Antwortzeit überschreitet. Es muss also nicht zwangsmäßig ein API-Fehler entstehen. Der Sinn des *Stresstests* liegt darin, das obere Limit der Applikation zu kennen. Sollte eine Applikation beispielsweise mit 1000 Nutzern noch umgehen können, bei dem 1005. Nutzer allerdings versagen, so ist dies eine äußerst wichtige Information, besonders in Anbetracht der Verfügbarkeit der Applikation während Stoßzeiten.⁶⁹

⁶⁴ Vgl. Molyneaux S. 39

⁶⁵ Vgl. De 2017, S. 159

⁶⁶ Vgl. Rodrigues u. a., S. 33; Molyneaux S. 39

⁶⁷ Vgl. De 2017, S. 159; Rodrigues u. a., S. 33

⁶⁸ Vgl. De 2017, S. 159

⁶⁹ Vgl. Molyneaux S. 39

3 Versuchsaufbau

3.1 Konfigurationserläuterung der Bereitstellungsalternativen

Bereitstellungsalternative	Virtual Server for VPC	Code Engine
Bereitstellungsart	Multi-tenant	Multi-tenant
virtual CPU (vCPU)	2	2
Arbeitsspeicher	8GB	8GB
Betriebssystem	Cent OS 8.3 - Minimal Install	N/A
Standort	Frankfurt	Frankfurt
IP / URL	161.156.171.2	https://pa-app.e9r2bo5mf3v.eu-de.codeengine.appdomain.cloud

Tab. 1 Konfigurationseinstellung der beiden Bereitstellungsalternativen

Damit ein relevanter Vergleich der Performance einer NodeJS Bereitstellung auf den verschiedenen IBM Cloud Services, Virtual Server for VPC und Code Engine aufgestellt werden kann, bietet sich ein Vergleich, mittels des bereits beschriebenen Verfahren für API-Performancetests, an. Dabei wird ein *Baseline*-, *Load*- und *Stresstest* durchgeführt, deren Bedeutung bereits im vorherigen Teil erläutert wurde. Es muss bei diesen Tests für eine möglichst hohe Vergleichbarkeit der beiden Bereitstellungsmöglichkeiten gesorgt werden, sodass die Ergebnisse ein Vergleichspotenzial besitzen. Daher wird im Folgenden zunächst die gewählte Infrastruktur beschrieben und begründet.

IBM bietet eine große Anzahl an verschiedenen Konfigurationsmöglichkeiten für den Virtual Server for VPC an. So kann der Standort der Infrastruktur ausgewählt werden, sowie das Betriebssystem. Zudem wird unterschieden zwischen einer *public* und *dedicated* Infrastruktur, beziehungsweise *multitenant* oder *singletenant* Infrastruktur. Bei der *public* Variante wird die Infrastruktur mit mehreren anderen Nutzern geteilt, diese sind jedoch nicht sichtbar untereinander. In der *dedicated* Variante hingegen wird eine Infrastruktur garantiert, die ausschließlich einem Nutzer zugeordnet ist. Diese Variante kostet entsprechend Aufpreis bietet allerdings im Gegenzug eine erhöhte Sicherheit. So kann eine Sicherheitslücke eines *Tenants* in einer *multitenant* Architektur möglicherweise andere *Tenants* ebenfalls betreffen.⁷⁰ Des Weiteren lassen sich der zugeordnete Speicher des Virtual Servers einstellen, sowie in welcher *Virtual-private-Cloud* (VPC) dieser bereitgestellt wird und an welches Netzwerkinterface der Server verbunden ist.

Die gewählte Konfiguration des Virtual Servers für die API-Performancetests ist der ersten Spalte der Tabelle 1 zu entnehmen. Alle anderen, nicht in Tabelle 1 erwähnten, Konfigurationsmöglichkeiten wurden auf der Standardeinstellung belassen. So liegt der allokierte, permanente Speicher bei einer Größe von 100GB und besitzt eine maximale Bandbreite von 393 Mbps. Das Netzwerkinterface entspricht ebenfalls dem Standard, und ermöglicht eine

⁷⁰ Vgl. Bezemer/Zaidman 2010, S. 3

maximale Auslastung der Netzwerkbandbreite von 3000 Mbps. Die HTTP-Anfragen besitzen jedoch nicht einmal eine Größe von 500 Bytes. Lai und Baker argumentieren, dass eine Erhöhung der Bandbreite keinen weiteren Einfluss auf die Applikationsperformance hat, solange ausreichend Bandbreite gewährleistet ist.⁷¹ Des Weiteren besitzt die API eine Gesamtgröße von gerade einmal 28MB, dementsprechend gibt es ebenfalls keine Grundlage das Volumen des allokierten, permanenten Speichers zu erhöhen.

Der Arbeitsspeicher und die CPU-Konfiguration entspricht der kleinstmöglichen Konfigurationsoption in der IBM Cloud für den Virtual Server for VPC. Der Hintergrund dieser Auswahl, lässt sich unter anderem auf die Kosten zurückführen. Die Kosten für eine derartige Konfiguration belaufen sich auf 0,104€ pro Stunde. Dieser Betrag setzt sich aus den Kosten für die Virtual Server Instanz, 0,089€ pro Stunde, sowie den Kosten für das Bootvolumen, 0,015€ pro Stunde, zusammen. Eine Erhöhung der vCPUs auf 4 und eine Erhöhung des Arbeitsspeichers auf 16 GB würde einen Aufpreis von insgesamt 0.089€ pro Stunde mit sich tragen, und somit die Kosten auf insgesamt 0.193€ pro Stunde erhöhen.⁷² Zwar bewegen sich diese Preise in den Centbeträgen, lässt man jedoch den Virtual Server den gesamten Monat durchgehend laufen, beträgt der Aufpreis zu der größeren Instanz knapp 60€. Der Sinn der daraus entstehenden wirtschaftlichen Folge lässt zudem Zweifel offen, da eine NodeJS-API standardmäßig nur einen einzelnen Kern verwenden kann. Selbst die gewählte Konfiguration mit zwei vCPUs ist im Grunde nicht wirtschaftlich effizient. IBM bietet zu dem Zeitpunkt der Verfassung dieser Arbeit allerdings keine niedrigere Konfigurationsmöglichkeit an. Außerdem wurde dem Virtual Server eine *floating* IP-Adresse zugeordnet, welche es dem Server ermöglicht über das öffentliche Internet erreichbar zu sein.

Der Virtual Server beinhaltet keine weiteren getätigten Konfigurationen, da die *Services* in einem möglichst originalen Zustand verglichen werden sollen. Die einzige Ausnahme diesbezüglich bildet die zusätzliche Konfiguration von HTTPS, welche für einen verhältnismäßigen Aufwand eine bessere Vergleichsgrundlage schafft. Mittels des CentOS Packets „openssl“ wurde ein selbstsigniertes SSL Zertifikat auf dem Virtual Server erstellt. Leider ist es nicht möglich HTTPS2, die zweite, verbesserte Version des HTTPS Protokolls, mittels des integrierten NodeJS Packets „HTTPS“ zu konfigurieren. Es gibt allerdings Möglichkeiten mittels *Libraries* von Drittanbieter dieses Problem zu umgehen. Die Verwendung dieser könnte jedoch möglicherweise zu einer unkontrollierbaren Abhängigkeit führen, die das Testergebnis beeinflussen könnte. Daher wurde sich gegen eine solche Problemumgehung entschieden.

Code Engine bietet ebenfalls eine Vielzahl an Konfigurationsmöglichkeiten für das Bereitstellen von Webapplikationen an. Der entscheidende Unterschied ist hierbei jedoch, dass sich

⁷¹ Vgl. Lai/Baker 1998, S. 3

⁷² Vgl. IBM o.J.a

kein Betriebssystem einstellen lässt, da es sich um ein *serverless* Angebot handelt. Stattdessen lässt sich ein *Containerimage* oder ein *Coderepository* angeben, welches die Gesamtheit der bereitzustellenden Applikation beinhalten muss. Des Weiteren muss der Port, auf welcher die containerisierte Applikation im Container läuft, angegeben werden. Ähnlich wie bei dem Virtual Server lässt sich auch bei Code Engine eine CPU und Arbeitsspeicher Konfiguration auswählen. Zudem lässt sich einstellen, wie viele Instanzen von der Applikation maximal und minimal erstellt werden können. Zuletzt lassen sich noch netzwerkspezifische Einstellungen vornehmen, wie die maximale Anzahl an gleichzeitigen Anfragen an die Applikation, das *Anfragentimeout* sowie die Schwelle, ab welcher eine neue Instanz hochgefahren werden soll, unter der Voraussetzung, dass dies durch die Einstellung der maximalen Instanzen gestattet ist.

Um eine höchstmögliche Vergleichbarkeit der API-Performancetests zu gewährleisten, wurde die Code Engine Applikation so ähnlich wie möglich zu der Virtual Server Instanz konfiguriert. Die Code Engine Applikation besitzt daher ebenfalls 2 vCPUs und 8GB Arbeitsspeicher. Die konkrete Konfiguration ist der zweiten Spalte der Tabelle 1 zu entnehmen. Zudem wurde die maximale Anzahl der Instanzen auf eine Instanz beschränkt, sodass ein mögliches horizontales Skalieren nicht den Vergleich verzerrt. Der Virtual Server kann in der vorgestellten Konfiguration ebenfalls nicht skalieren. Die Netzwerkkonfiguration wurde zunächst auf den Standardeinstellungen belassen. Diese beschränken die maximalen, gleichzeitigen Anfragen auf 100 Anfragen pro Instanz und schreiben ein *Anfragentimeout* von 300s vor. Die Schwelle, ab welcher neue Instanzen gestartet werden, liegt ebenfalls bei 100 Anfragen pro Instanz. Diese ist jedoch für den weiteren Verlauf dieser Arbeit irrelevant, da es maximal eine Instanz gleichzeitig geben darf. Ebenfalls irrelevant für diese Praxisarbeit sind die weiteren Konfigurationsmöglichkeiten für die Code Engine Applikation, wie das Festlegen von Umgebungsvariablen oder das Überschreiben der Standardkommandos bei Ausführung eines *Containerimages*. Preislich unterscheidet sich Code Engine insofern von dem Virtual Server, dass lediglich die Zeit berechnet wird, welche auch für die Bereitstellung tatsächlich in Anspruch genommen wird. Das bedeutet konkret für die gewählte Konfiguration, dass pro beanspruchte vCPU Sekunde umgerechnet 0.000027€ berechnet werden, sowie 0.0000028€ pro beanspruchten Gigabytesekunde Arbeitsspeicher. Zuletzt werden auch die empfangenen HTTP-Anfragen berechnet mit 0.43€ pro einer Million Anfragen.⁷³

Erwähnenswert ist außerdem, dass Code Engine standardmäßig eine URL bereitstellt anstatt einer IP-Adresse und bereits ein *Secure-Socket-Layer* (SSL) Zertifikat eingerichtet hat. Daher ist die Code Engine Applikation lediglich über den sicheren Standard HTTPS2 erreichbar.

⁷³ Vgl. IBM o.J.b

3.2 API & API Endpunkt

Die beiden Bereitstellungsmöglichkeiten werden anhand einer NodeJS-API auf ihre Performance verglichen. Die NodeJS-API wird mittels eines Express-Webserver bereitgestellt und auf beiden Systemen in einem *Dockercontainer* bereitgestellt. Code Engine benötigt ein *Dockerimage* ausnahmslos. Für einen geeigneten Vergleich bietet es sich demnach an, dasselbe *Dockerimage* auf dem Virtual Server bereitzustellen. Dies hat zur Folge, dass die API auf demselben *Dockerbasisimage* ausgeführt wird, namentlich node-14 alpine, und somit die Wahrscheinlichkeit von Testunregelmäßigkeiten aufgrund des Betriebssystems verringert wird. Des Weiteren wird somit versichert, dass beide Bereitstellungsmöglichkeiten dieselbe Nodeversion verwenden. Das *Dockerbasisimage* beinhaltet die Nodeversion 14 mit einem minimalen Image von Linux Alpine, einem „security-oriented, lightweight Alpine Linux distribution with a complete package index that is no more than 8MB in size.“⁷⁴.

Die zu testende NodeJS-API wird auf dem Port 8080 innerhalb des *Dockercontainers* bereitgestellt und besitzt lediglich einen Endpunkt, welcher den *Resourcepath* „/api/demo“ trägt. Aufgerufen wird dieser Pfad mittels einer HTTP-GET-Anfrage.

Bei Aufruf dieses Pfades wird das willkürlich gewählte Passwort, „This-is-my-test-password-01“ *gehashed*, mittels der NodeJS-Library „Bcrypt“. Dies gewährleistet, dass die Antwort nicht vom Server *gecached* werden kann, da jedes generierte Passwort individuell ist. Bcrypt stellt ein renommiertes Schema für das *Hashing* von Passwörtern dar und existiert seit über 20 Jahren.⁷⁵ Aufgrund dieser langen Lebensdauer gilt Bcrypt als renommierter Standard.⁷⁶ Der Bcrypt-Algorithmus benötigt drei Input Variablen. *Cost*, *Salt* und das Passwort, welches *gehashed* werden soll. *Cost* entscheidet, wie rechenintensiv der Prozess ist, während *Salt* ein zufällig generierter 128-bit Wert ist, der dafür sorgt, dass bei der Eingabe desselben Passworts nicht derselbe Hash entsteht.⁷⁷ Die Rechenintensivität des Prozesses, sorgt für eine ideale Eignung dieses Verfahrens für die Verwendung in einem Performancevergleich. Aufgrund der hohen Last, die auf die CPU fällt, ist es äußerst wichtig die Methoden, die Passwörter mittels Bcrypt verschlüsselt, asynchron auszuführen, sodass der *Eventloop* von NodeJS nicht blockiert wird.⁷⁸ Diese Methoden sind in der NodeJS-Library Bcrypt enthalten und in der zu testenden API asynchron ausgeführt, wie dem Quelltext

```
const password = "This-is-my-test-password-01";
const hashedPassword = await bcrypt.hash(password, 10);
```

Abb. 12: Code Implementation Bcrypt-Hashing

⁷⁴ Zerouali 2018, S. 1

⁷⁵ Vgl. Sriramya/Karthika 2015, S.5553

⁷⁶ Vgl. Sriramya/Karthika 2015, S.5553

⁷⁷ Vgl. Malvoni/Knezovic 2014, S. 1f.

⁷⁸ Vgl. NpmJS o.J.

in Abbildung 12 zu entnehmen ist. Zudem ist in derselben Abbildung zu sehen, dass die Anzahl der *Saltrounds* auf zehn gesetzt wurde. Die *Saltrounds* repräsentierten den bereits erwähnten *Costfaktor*. Bcrypt definiert diese als „the cost of processing the data“⁷⁹. Je höher dieser *Costfaktor*, desto länger dauert es ein Passwort zu verschlüsseln. Die hohen Hardwareanforderungen, welche Bcrypt stellt, bieten Schutz gegen Angreifer aus dem Netz, da unverhältnismäßig viel, kostspielige Hardware gestellt werden müsste, um ein Passwort zu entschlüsseln.⁸⁰

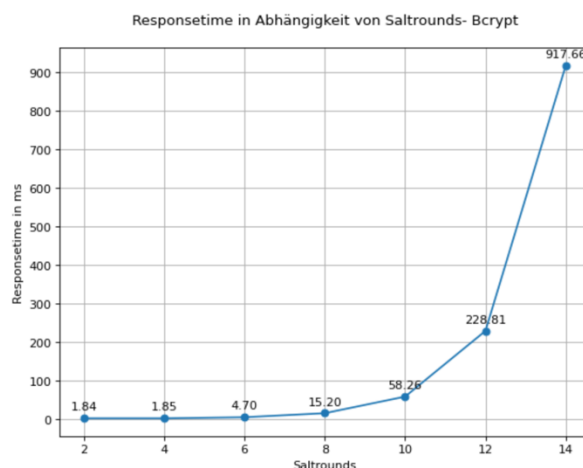


Abb. 13: HTTP-Anfragenzeit in Abhängigkeit der *Saltrounds*

In Abbildung 13 wurde die HTTP-*Reponsetime* der NodeJS-API auf dem Virtual Server, mit der bereits beschriebenen Konfiguration und des beschriebenen Endpunkts, auf verschiedene *Saltrounds* getestet. Es ist die Antwortzeit der HTTP-Anfrage auf der y-Achse in Abhängigkeit der gewählten *Saltrounds* auf der x-Achse zu sehen, welche die Anzahl der *Saltrounds* von zwei bis 14 darstellt. Die Messergebnisse eines Datenpunktes entsprechen dem Durchschnitt von jeweils fünf HTTP-Anfragen. Deutlich zu sehen ist, dass die *Reponsetime* bei steigender Anzahl an *Saltrounds* exponentiell ansteigt, aufgrund der steigenden, geforderten Rechenleistung. Für die weiteren API-Perfomancetests wurde sich auf die Verwendung von zehn *Saltrounds* entschieden, da diese bereits einen erheblichen Rechenaufwand für den Server darstellen, wie Abbildung 13 zeigt. Der benötigte Rechenaufwand lässt dennoch Tests mit einer vermehrten Anzahl an Nutzern zu. Dies kann bei einer höheren Anzahl an *Saltrounds* auf den gewählten Konfigurationen von Code Engine und des Virtual Servers unter Umständen nicht gewährleistet werden. Zudem stellen zehn *Saltrounds* den standardmäßig eingestellten Wert in der Beschreibung der NodeJS-Library Bcrypt dar, sodass von einem vernünftigen und realistisch eingestellten Wert ausgegangen werden kann.⁸¹

⁷⁹ NpmJS o.J.

⁸⁰ Vgl. Harba 2015, S. 15

⁸¹ Vgl. NpmJS o.J.

3.3 Vorstellung der Testinfrastruktur

Die Infrastruktur für die folgenden *Loadtests* ist der Abbildung im Anhang 1 zu entnehmen. Der *JMeter-Client*, von welchem die Tests ausgehen, wird auf einem separaten Ubuntu Virtual Server ausgeführt. Erste Versuche für *Loadtests* wurden von einem privaten MacBook Pro ausgeführt. Jedoch konnten starke Unregelmäßigkeiten bei den Testergebnissen festgestellt werden, die vermutlich durch zusätzliche installierte Programme oder *Services* entstanden, die auf dem MacBook installiert waren und während des Tests wichtige Internetbandbreite belegten. Mittels eines extra Virtual Servers wird eine freie Bandbreite, sowie kein nennenswerter *Overhead* durch andere Programme garantiert, welche die Tests beeinflussen könnten.

Die verschiedenen *Services* sind in demselben VPC in Frankfurt bereitgestellt, dennoch wurde sich für einen Test über das öffentliche Internet entschieden, um ein möglichst realistisches Szenario zu repräsentieren. Ein realer Nutzer würde schließlich auch über das öffentliche Internet auf die API zugreifen.

4 Versuchsdurchführung

4.1 Baselinetest

Der erste Test, der durchgeführt wird, ist der *Baselinetest*. Die Konfiguration für diesen Test ist der Tabelle im Anhang 2 zu entnehmen. Zusätzlich ist zu erwähnen, dass in jedem folgenden Test in dieser Praxisarbeit die Einstellung getroffen wurde, dass bei Aufkommen einer Fehlermeldung der Test sofort stoppt. Zudem ist eingestellt, dass derselbe Nutzer für jede Wiederholung verwendet wird. Es werden weder spezielle Parameter mit der HTTP-Anfrage gesendet noch ein HTTP-Body, da es sich ohnehin bei jeder Anfrage um den HTTP-Typ „GET“ handelt. Diese Konfigurationsoptionen werden im Folgenden nicht erneut erwähnt.

Die *Ramp-up-Period* wurde auf 60 Sekunden gestellt, da somit gewährleistet werden kann, dass ausreichend Zeit besteht um die Nutzer auf die *Services* zu lassen, ohne Fehlermeldung zurückzubekommen. In ersten Versuchen war diese auf lediglich eine Sekunde gestellt und sorgte vereinzelt für Fehler in den ersten HTTP-Anfragen bei beiden Bereitstellungsmöglichkeiten.

Für die Rechtfertigung der *Threadanzahl* und der Dauer des Tests werden im Rahmen des *Baselinetests* für gewöhnlich die Last, welche die API in ihrem Einsatz vermutlich durchschnittlich erwartet, herangezogen. In dem vorliegenden Fall ist dies jedoch nur sehr schwer zu bestimmen, da die getestete API lediglich zu Testzwecken dient und nicht für die Öffentlichkeit bereitgestellt wird. Für den *Baselinetest* wurde sich auf eine Testlänge von 30 Minuten entschieden. Zwar gibt es keine objektiv korrekte Testlänge, dennoch schreiben unter Anderem Rodrigues und Demion in ihrem Buch „Master Apache JMeter – From Load Testing to DevOps“, dass von kurzfristigen Performancetests, also Tests, die kürzer als eine halbe Stunde dauern, abgesehen werden sollte, da sonst nicht garantiert werden könnte, dass die Bereitstellungsmöglichkeit kontinuierlicher Last standhalten kann.⁸² Die Testlänge wurde dementsprechend auf die Mindestlänge von 30 Minuten in Einklang mit Rodrigues und Demion definiert, sodass die verursachten Kosten für beide Bereitstellungsmöglichkeiten möglichst gering gehalten werden konnten, aber dennoch ein Mehrwehrt generiert werden kann.

Die *Threadanzahl* ist dabei ebenfalls schwierig objektiv korrekt einzustellen. Während eines *Baselinetests* sollte der Server keineswegs vollständig ausgelastet sein, daher sprach alles für das Betrachten der CPU-Performancemetrik während eines Tests, sodass festgestellt werden kann, bei wieviel Auslastung der Server liegt. Eine CPU-Messung kann mittels des JMeter-Plugins *Perfmon* vorgenommen werden. Dafür muss der entsprechende Java-Client auf dem zu testenden Server installiert werden. Da Code Engine kein eigenes Betriebssystem für den Nutzer bereitstellt, auf dem Java installiert werden kann, lässt sich diese Untersuchung ausschließlich auf dem Virtual Server durchführen. Jedoch musste festgestellt werden, dass die

⁸² Vgl. Rodrigues u. a. 2019, S. 59

CPU-Messung durchgehend bei 50% lag, unabhängig von der eingestellten Threadanzahl des *Baselintests*. Dies liegt daran, dass NodeJS lediglich einen Kern belegen kann, der Virtual Server jedoch zwei Kerne besitzt. Es war also nicht möglich einen akkuraten CPU-Auslastungstest vorzunehmen, sodass die *Threadanzahl* zunächst auf 150 *Threads* gesetzt wurde und im Nachhinein betrachtet wurde, wie gut die API mit der Last umgehen kann. Braucht die API zu lange für die Bearbeitung der Anfragen, so ist die API zu stark ausgelastet und die *Threadanzahl* muss gesenkt werden.

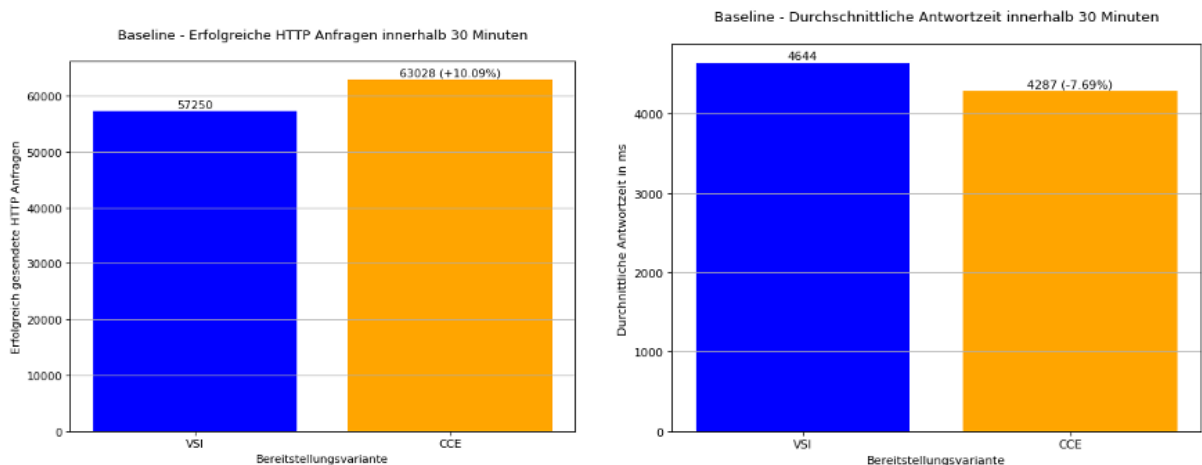


Abb. 14: Baselinetest – Erfolgreiche HTTP-Anfragen & Durchschnittliche Antwortzeit

Abbildung 14 kann jedoch deutlich entnommen werden, dass bei beiden Bereitstellungsvarianten die durchschnittliche Antwortzeit zwischen vier und fünf Sekunden liegt. In der Abbildung ist in der linken Hälfte die Bereitstellungsvariante auf der x-Achse im Verhältnis zu den insgesamt erfolgreichen HTTP-Anfragen während der Testdauer dargestellt. Auf der rechten Hälfte gilt dieselbe Beschriftung, mit der Ausnahme, dass auf der y-Achse die durchschnittliche Antwortzeit in Millisekunden aufgetragen ist. Laut Barber in seinem Artikel „How Fast Does a Website Need To Be?“ ist ein Wert zwischen 3 und 5 Sekunden als typisch zu beurteilen. Somit ist die gewählte *Threadanzahl* von 150 Nutzern als eine durchschnittliche Last für die gewählten Konfigurationen der Bereitstellungsmöglichkeiten anzusehen.⁸³

Den *Baseline*-Testergebnissen in Abbildung 14 ist zu entnehmen, dass Code Engine im *Baselinetest* die besseren Ergebnisse erzielen konnte im Vergleich zu dem Virtual Server. So konnten in derselben Zeit durch Code Engine circa 10% mehr Anfragen bearbeitet werden als mit dem Virtual Server. Diesem Test liegt zugrunde, dass die durchschnittliche Bearbeitungszeit einer Anfrage bei Code Engine vergleichsweise circa 7,6% kürzer war, wie in Abbildung 14 zu sehen ist. Für einen genaueren Vergleich der beiden Messungen ist im Anhang 3 und 4 die Antwortzeit in Verhältnis zur Testdauer für beide Bereitstellungsmöglichkeiten dargestellt.

⁸³ Vgl. Scott 2010, S. 4

Eine interessante Beobachtung, die in dem Graph im Anhang 4 auffällt, ist die kurzzeitig hohe Antwortzeit von Code Engine in den ersten HTTP-Anfragen. Gründe dafür könnten möglicherweise mit den Gründen übereinstimmen, die Scott für eine ausreichende *Ramp-up-Period* nennt. So müssen nämlich beispielsweise *Caches* gefüllt werden oder die Last gleichmäßig mittels *Loadbalancer* verteilt werden.⁸⁴ Dies geschieht im Hintergrund bei Code Engine und ist für den Nutzer nicht sichtbar.

Code Engine scheint also den *Baselinetest* performanter zu absolvieren als der Virtual Server. Allerdings gilt es auch noch die Konsistenz dieser Messungen zu betrachten.

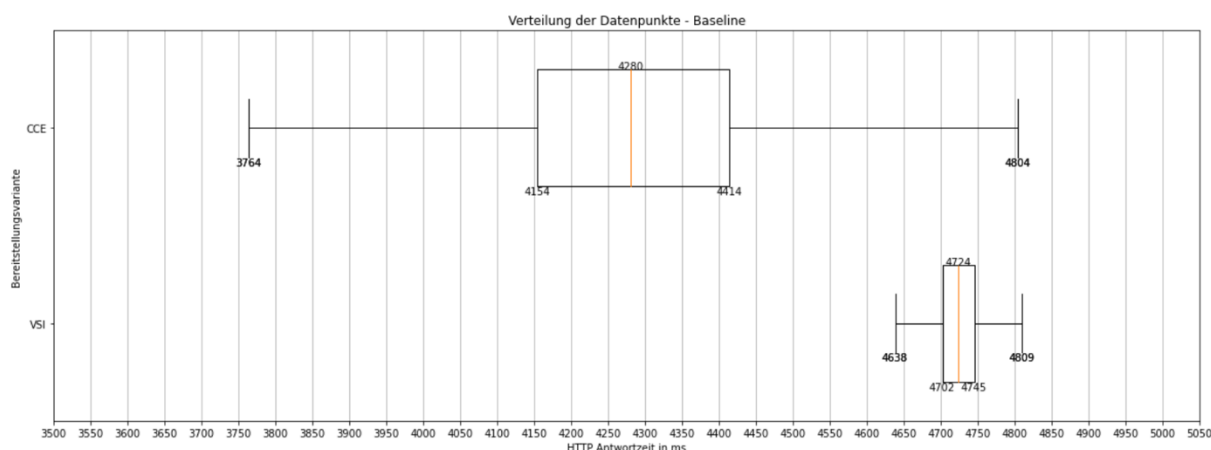


Abb. 15: Boxplot Verteilung der HTTP-Antwortzeiten im Baselinetest

Schaut man sich die Abbildung 15 an, so wird deutlich, dass die einzelnen Messpunkte in Code Engine deutlich stärker variieren als die Messpunkte des Virtual Servers. Dargestellt ist ein Boxplot, der den Median der Messung, sowie das Minimum, Maximum und das untere und obere Quantil des *Baselinetests* beider Bereitstellungsmöglichkeiten beinhaltet. Auf der x-Achse ist die Antwortzeit in Millisekunden aufgetragen, während auf der y-Achse die Bereitstellungsvariante dargestellt ist. Es wurde sich für die Darstellung mittels eines Boxplots entschieden, da somit die Verteilung und Symmetrie der Datenpunkte anschaulich dargestellt werden kann. Dies gestaltet sich schwieriger in den Graphen im Anhang 3 und 4, da dort die y-Achsenbeschriftung zu grob gewählt ist, um eine qualitative Aussage über die Verteilung zu treffen.⁸⁵ So liegt der Interquartilsabstand bei Code Engine bei 260ms, im Vergleich zu lediglich 43ms bei dem Virtual Server. Noch deutlicher wird die breite Ergebnisspanne des *Baseline-tests* von Code Engine bei Betrachtung der Spannweite, welche bei Code Engine über sechsmal so hoch ist, nämlich bei über einer Sekunde (1040ms) im Vergleich zu lediglich 171ms bei dem Virtual Server. Das obere Quantil des Testergebnisses von Code Engine liegt somit nur 5ms unter dem oberen Quantil von dem Virtual Server. Relevant werden diese inkonsistenten

⁸⁴ Vgl. Scott 2010, S. 5

⁸⁵ Vgl. Williamson u. a. 1989, S. 916

Messergebnisse, wenn gewisse Anforderungen an die API gestellt werden. Sei die Anforderungen an die API beispielsweise, dass keine HTTP-Anfrage an die API bei einer Last von 150 Nutzern länger als 5000ms dauern darf, so existieren in der beispielhaften Messung 168 Messpunkte bei welcher Code Engine eine solche Anforderung nicht erfüllen würde. Im Vergleich dazu, existieren lediglich vier solcher Messpunkte für den Virtual Server. Die Messpunkte des Virtual Servers sind also deutlich konsistenter trotz des insgesamt langsameren Durchschnittswertes.

Bevor nun genauer auf den durchgeführten *Loadtest* eingegangen wird, gilt es die Netzwerkkomponente noch einmal genauer zu betrachten. Die Hypothese bestand, dass der Virtual Server im Durchschnitt langsamere Ergebnisse ablieferte aufgrund einer besseren Netzwerkinfrastruktur des Code Engine Services. Code Engine ist schließlich der neuere Service der Beiden und könnte möglicherweise von einer schnelleren Netzwerkinfrastruktur profitieren. Diese Komponente lässt sich einfach eliminieren, indem ein *localhost Baselinetest* durchgeführt wird. In einem solchen Test kommunizieren, ungleich wie in dem beschriebenen Test-szenario in Anhang 1, der zu testende Server und die testende Instanz über eine Lokalhost-adresse. Der JMeter Test wird also auf demselben Server ausgeführt, auf dem auch die API bereitgestellt ist. Die Konfiguration stimmt exakt mit der obigen, beschriebenen Konfiguration für den Virtual Server überein, nur dass die IP nun der *localhost* IP-Adresse entspricht.

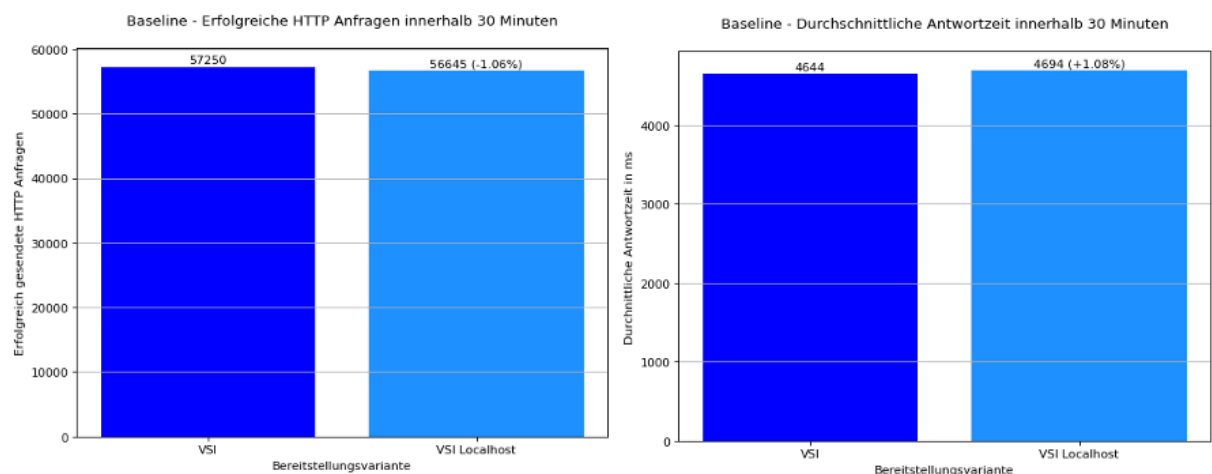


Abb. 16: Baselinetest Localhost Wiederholung – Erfolgreiche HTTP-Anfragen & Durchschnittliche Antwortzeit

Das Testergebnis, das der Abbildungen 16 zu entnehmen ist, überrascht. Die Abbildungsbeschriftung entspricht der Abbildung 14. Tatsächlich wurde bei einem *localhost* Test ein schlechteres Ergebnis für den Virtual Server erzielt. So konnte der Virtual Server circa 1% weniger Anfragen bearbeiten im Vergleich zu einem *Baselinetest* über das Internet. Dementsprechend stieg die durchschnittliche Antwortzeit von 4644ms auf 4694ms an. Es hätte erwartet werden können, dass ein *localhost* Test eigentlich ein besseres Ergebnis erzielen müsste

als ein Test über das Internet, jedoch beweist dies, dass das Internet keinen erheblich großen Einfluss auf die Testergebnisse hat. Andere Faktoren, wie der Zeitpunkt an welchem der Test ausgeführt wird, könnten unter Umständen ein größerer Faktor sein. Eine weitere Erklärung, weshalb der *localhost* Test ein etwas schlechteres Ergebnis erzielt, könnte der zusätzliche *Overhead* liefern, der durch JMeter auf dem Virtual Server entstand. In einem Test über das Internet, ist der Initiator des Tests auf einem separaten System, ungleich bei einem *localhost* Test. Es kann also für die folgenden Tests davon ausgegangen werden, dass das Internet maximal einen sehr geringen Einfluss auf die Testergebnisse hat.

4.2 Loadtest

Der Hauptunterschied des *Loadtests* zum *Baselinetest* besteht in der *Ramp-up-Period*. Diese ist nun nicht mehr auf 60 Sekunden angesetzt, sondern auf 1800 Sekunden wie in der Tabelle im Anhang 5 zu sehen ist. Die *Ramp-up-Period* entspricht somit der Dauer des Tests. Somit kann eine kontinuierlich steigende Nutzerlast simuliert werden. Circa alle 4 Sekunden wird ein zusätzlicher, neuer Nutzer auf die API zugelassen. Die *Threadanzahl* wurde auf 500 Nutzer limitiert. Da, die im *Baselinetest* definierten 150 Nutzer einer Durchschnittslast entsprechen soll, würden 500 Nutzer dementsprechend etwas mehr als der dreifachen Durchschnittslast entsprechen. Dies ist eine realistische Zahl für Stoßzeiten auf einer Website, etwa wegen einer speziellen Aktion oder bestimmten Saison. Würde die verwendete API für eine Webseite verwendet werden, die Aktienwerte anzeigt, könnte täglich mit einem solchen Anstieg bei der Öffnung der verschiedenen Börsen gerechnet werden. Es muss ermittelt werden, wie die API mit einem solchen Lastenanstieg umgeht.

Die Testergebnisse lassen sich in den detaillierten Graphen in Anhang 6 und 7 dieser Arbeit auffinden. In beiden Graphen ist die HTTP-Antwortzeit in Abhängigkeit der Anzahl an aktiven Nutzern dargestellt. Auf den ersten Blick sehen die beiden Graphen sehr ähnlich aus. Keiner der beiden Graphen zeigt Anzeichen von Überlastung wie das plötzliche Oszillieren des Graphen. Es wurden auch keine Fehlermeldungen während des Tests festgestellt. Der Graph beinhaltet zudem die durchschnittliche Antwortzeit der beiden Bereitstellungsmöglichkeiten, gemessen über den gesamten Test. Code Engine liegt in dem *Loadtest* bei einer durchschnittlichen Antwortzeit von 7293ms, während der Virtual Server eine Anfrage in 7918ms durchschnittlich bearbeiten konnte.

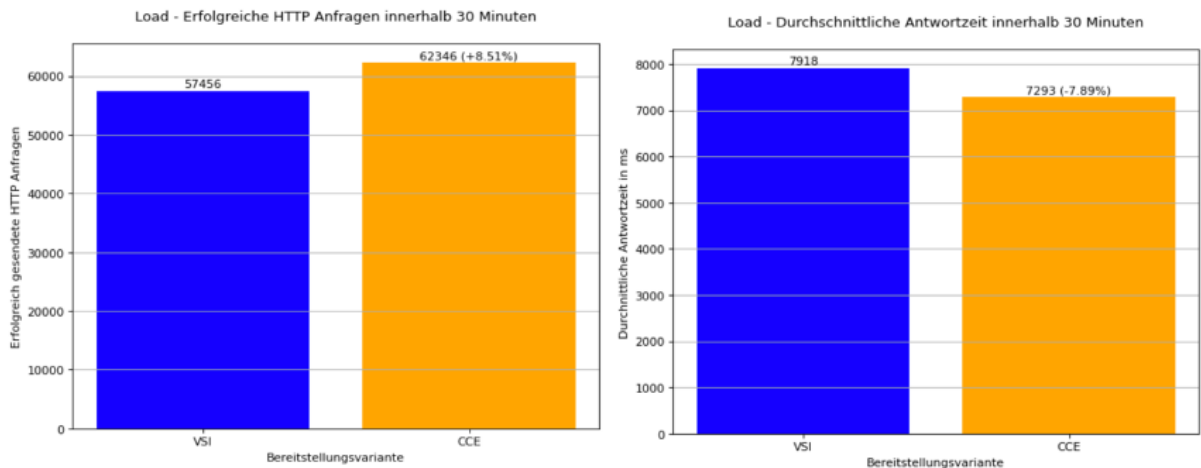


Abb. 17: Loadtest – Erfolgreiche HTTP-Anfragen & Durchschnittliche Antwortzeit

In Abbildung 17 ist die Antwortzeit der beiden Bereitstellungsvarianten noch einmal visualisiert dargestellt. Die Abbildungsbeschriftung entspricht der aus Abbildung 14 im *Baselinetest*. Code Engine ist somit circa 7.89% schneller als der Virtual Server. Dies deckt sich mit den Ergebnissen aus dem *Baselinetest*. Dementsprechend absolviert Code Engine mit 62346 HTTP-Anfragen, circa 8.5% mehr Anfragen als der Virtual Server in derselben Zeit. Anhand der *Loadtests* lassen sich darüber hinaus Aussagen über die Skalierbarkeit des Systems treffen. Die Frage, wieviel zusätzliche Last ein weiterer Nutzer auf das System ausübt, kann beantwortet werden, indem die Steigung der im Anhang 6 und 7 abgebildeten Graphen berechnet wird. Da es sich um annähernd lineares Wachstum in beiden Graphen handelt, wurde für die Berechnung der Steigung die Funktion `polyfit()` aus der Python-Library „Numpy“ verwendet. Mehr über diese Funktion kann auf der offiziellen NumPy-Website erfahren werden.⁸⁶

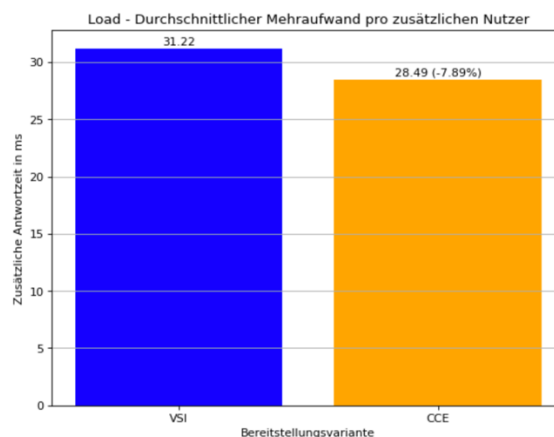


Abb. 18: Loadtest - Durchschnittlicher Mehraufwand pro zusätzlichen Nutzer

⁸⁶ Vgl. dazu ausführlich: NumPy 2021

Abbildung 18 zeigt den durchschnittlichen zusätzlichen Mehraufwand pro Nutzer. Auf der y-Achse ist die zusätzliche Antwortzeit in ms im Verhältnis zu der Bereitstellungsvariante auf der x-Achse dargestellt. Ein zusätzlicher Nutzer auf die Virtual Machine würde eine Erhöhung der Antwortzeit von circa 31,22 ms verantworten, während es bei Code Engine lediglich 28,49ms sind. Dies gilt allerdings nur, solange angenommen werden kann, dass auch mit zusätzlichen Nutzern ein lineares Wachstum garantiert ist. Sollte aus verschiedensten Gründen die Kurve kein lineares Wachstum mehr vornehmen, so muss eine andere Berechnungsgrundlage gewählt werden.

4.3 Stresstest

Das Ziel eines *Stresstests* ist den *Breakingpoint* der API herauszufinden. Dies bedeutet, dass der Test solange durchgeführt werden soll, bis die API Fehlermeldung zurückgibt. Daher wurde die *Threadanzahl* auf 2000 Nutzer gesetzt wie der Tabelle im Anhang 8 zu entnehmen ist. Bei der Durchführung des Tests sind mit dieser Anzahl an *Threads* keine Probleme seitens JMeter aufgetreten. Dennoch empfiehlt Erinle in seinem Buch „Performance Testing with JMeter“ ab 600 Nutzern bereits *distributed Testing* durchzuführen.⁸⁷ Dies bedeutet, dass nicht wie in der Abbildung im Anhang 1 gezeigt, der Test von nur einer Virtual Machine ausgeht sondern einer Vielzahl von Maschinen, die zusammen die gewünschte Anzahl an Nutzern simulieren.

Es wurde sich allerdings gegen das Verwenden von *distributed Testing* entschieden aus den folgenden Gründen. Zunächst muss erwähnt werden, dass selbst mit der hohen Anzahl an Nutzern keine Fehlermeldungen seitens JMeter entstanden, wie etwa Java Memory overflow Fehler, die unter anderem von Jing und Lan in Ihrem Artikel „JMeter-based Aging Simulation of Computing System“ erwähnt werden.⁸⁸ Vermutlich liegt dies an der relativ leistungsstarken Konfiguration des Virtual Servers, von welchem die Tests initiiert werden. Ein weiterer, wichtiger Faktor sind die entstehenden Kosten. So müssen bei dem Ansatz von *distributed Testing* mehrere Server gestartet werden, die neben Fixkosten wie der *floating IP* auch variable Kosten mit sich tragen. Im Rahmen dieser Arbeit standen diese Kosten in keinem Verhältnis.

Um frühzeitige *Timeouts* zu verhindern, wurde für den *Stresstest* die Testlänge von 30 Minuten auf 45 Minuten angehoben. Die Testergebnisse bei der erstmaligen Durchführung dieses Tests lassen sich dem Anhang 9 und 10 entnehmen. Auffällig ist, dass der *Stresstest* bei Code Engine ab dem 1424. Nutzer eine Fehlermeldung zurückgibt. Die HTTP-Anfrage gibt die Nachricht „Socket closed“ zurück. Aufgrund der gewählten Einstellung beendet sich der Test

⁸⁷ Vgl. Erinle 2013, S. 89

⁸⁸ Vgl. Jing u. a. 2010, S. 283

automatisch. Im Vergleich dazu zeigt sich der Virtual Server konsistent und absolviert den Test selbst mit 2000 Nutzern. Es war im Rahmen dieser Praxisarbeit also nicht möglich den *Breakingpoint* des Virtual Servers festzustellen. Der *Breakingpoint* liegt offensichtlich oberhalb 2000 Nutzer und ist ohne *distributed Testing* nicht erreichbar. Eine Ausweitung des aktuellen Tests auf mehr als 2000 Nutzer stellt aus den bereits genannten Gründen keine Option dar. Zudem liegt die Antwortzeit für beispielsweise den Virtual Server, während einer Last von 2000 Nutzern, bei über 60 Sekunden. Eine solche Antwortzeit ist für die allermeisten Applikationen ohnehin nicht tragbar und laut Scott's Klassifikation bei weitem in einem inakzeptablem Bereich.⁸⁹

Es gilt im Rahmen dieses Vergleichs die Konfigurationen beider Bereitstellungsmöglichkeiten möglichst in der Standardeinstellung zu belassen. Allerdings verwundert das aktuelle Ergebniss in Hinblick auf die grundsätzlich schnellere Responsetime von Code Engine in den früheren Tests. Infolgedessen wurde die Einstellung für die maximalen, gleichzeitigen Anfragen pro Instanz bei Code Engine von 100 auf 200 Anfragen angehoben. Dies widerspricht den vorherig beschriebenen Konfigurationseinstellungen von Code Engine, dennoch handelt es sich hierbei um eine Einstellung, die für den Nutzer in keinsten Weise Mehrkosten verursacht und daher lohnenswert zu betrachten ist. Sollte mit einer Erhöhung dieser Einstellung immernoch eine Fehlermeldung bei circa 1424 Nutzern entstehen, so würde dies auf ein Performanceproblem hinweisen und die maximale Last des Services nahe legen.

In Anhang 11 ist das Testergebnis der erneuten Durchführung des Tests mit der aktualisierten Einstellung zu betrachten. Nun ist deutlich erkennbar, dass die getroffene Einstellung einen Unterschied ausübt. So erreicht Code Engine nun auch die 2000 Nutzer ohne Fehlermeldungen. Dies lässt darauf vermuten, dass nicht die Leistung von Code Engine, sondern die zu geringe Einstellung von gleichzeitigen Nutzern den Flaschenhals im vorherigen Experiment darstellte. Für die folgenden Vergleiche wird die verbesserte Messung von Code Engine als Grundlage verwendet.

⁸⁹ Vgl. Scott 2010, S. 4

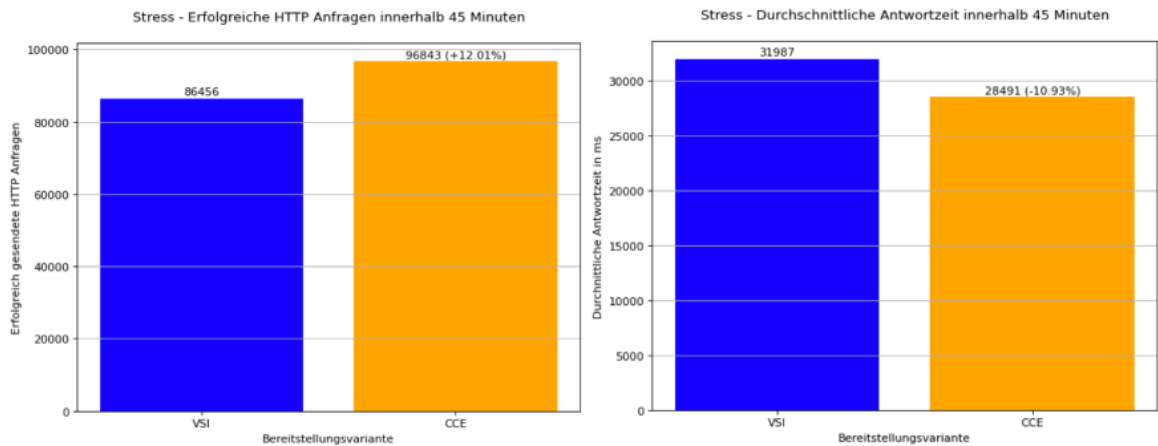


Abb. 19: Stresstest - Erfolgreiche HTTP-Anfragen & Durchschnittliche Antwortzeit

In den beiden Graphen der Abbildung 19 sind erneut die erfolgreichen HTTP-Anfragen innerhalb von 45 Minuten, sowie die durchschnittliche Antwortzeit der beiden Bereitstellungsvarianten dargestellt. Die Graphenbeschriftung stimmt mit den Graphenbeschriftungen aus dem *Baselinetest* und *Loadtest* überein. Besonders interessant werden diese Darstellungen im Vergleich zu den vorherigen Tests. Während im *Baseline-* und *Loadtest* Code Engine circa 8.5% und circa 10% mehr Anfragen behandeln konnte, sind es im *Stresstest* bereits über 12 % mehr. Darüberhinaus verarbeitet Code Engine die Anfragen nahezu 11% schneller im *Stresstest* im Vergleich zu 7.9% schneller im *Loadtest* und 7.7% schneller im *Baselinetest*. Möglicherweise könnte dies darauf hinweisen, dass Code Engine mit einer steigenden Nutzerlast besser umgehen kann als der Virtual Server.

Bei der Betrachtung des Mehraufwands pro Nutzer in Abbildung 20, fällt allerdings auf, dass sich der Virtual Server im Vergleich zum *Loadtest* tatsächlich um circa 0.4 Sekunden verbessert hat. Code Engine hat sich allerdings um mehr als 1.2 Sekunden verbessert. Die Graphenbeschriftung entspricht der Graphenbeschriftung aus Abbildung 18.

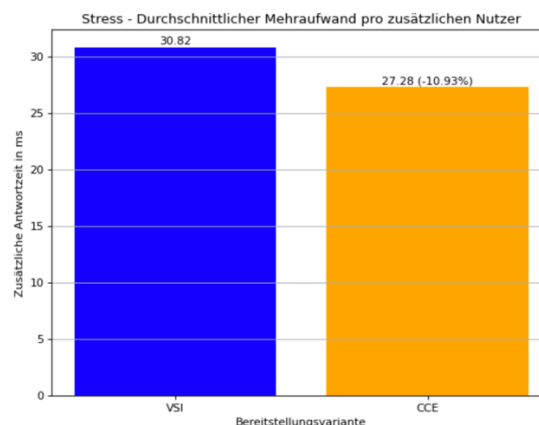


Abb. 20: Stresstest - Durchschnittlicher Mehraufwand pro zusätzlichen Nutzer

Bei keinem der beiden Bereitstellungsvarianten beginnt also die Performance nachzulassen im höheren Lastbereich. Im Gegenteil, es sind Verbesserungen der gesamten Performance festzustellen im Verhältnis zur gegebenen Last, wie in Abbildung 20 erkennbar ist durch den gesunkenen Mehraufwand pro Nutzer in beiden Bereitstellungsvarianten. Die extreme Verbesserung bei Code Engine liegt möglicherweise an der bereits beschriebenen, vergleichbar hohen Antwortzeit bei Code Engine für die ersten HTTP-Anfragen während des Beginns des Experiments, auf welche bereits im *Baselinetest* hingewiesen wurde. Diese sind erneut erkennbar im Anhang 11 am Anfang der Messung. Da der *Stresstest* allerdings eine längere Testdauer besitzt, verlieren diese ersten, langsamen HTTP-Anfragen an Gewicht in der gesamten Messung.

Zuletzt soll erneut auf die Konsistenz der beiden Messungen eingegangen werden, die bereits im *Baselinetest* angesprochen wurde. Anhand der Messergebnisse im Anhang 9 und 11 sind keine eindeutigen Aussagen zu treffen. Dies könnte unter Umständen aber daran liegen, dass die erzeugten Graphen von JMeter aus Performancegründen nicht jeden einzelnen Datenpunkt beinhalten. Im *Baselinetest* wurde bereits festgestellt, dass der virtual Server die konsistenteren Messergebnisse liefert. Mittels einer Regressionsgeraden und dem Bestimmtheitsmaß soll diese Behauptung nun auch im *Stresstest* überprüft werden. Mit Hilfe der Regressionsgerade wird versucht einen linearen Zusammenhang zwischen der Anzahl der aktiven *Threads* und der Antwortzeit herzustellen. Das Bestimmtheitsmaß R^2 gibt an, wie gut die Regressionsgerade an die entsprechenden Datenpunkte angepasst ist. Sie gibt das Verhältnis der erklärbaren Streuung zur gesamten Streuung an. Dementsprechend verschlechtern unerklärte Abweichungen das Bestimmtheitsmaß. Dies bedeutet, dass HTTP-Anfragen, die im Verhältnis sehr lange dauern oder nur sehr kurze Zeit benötigen, das Bestimmtheitsmaß verschlechtern. Daher eignet sich das Bestimmtheitsmaß um eine Aussage über die Konsistenz der HTTP-Anfragen im Verhältnis zur Anzahl der *Threads* zu tätigen. Ein ähnlicher Ansatz wurde von Sereinig in seiner Diplomarbeit gewählt, welche im Rahmen der Firma „Philipps“ durchgeführt wurde. Dort wurde das Bestimmtheitsmaß verwendet um eine Erkenntnis darüber zu erlangen, wie gut eine Regressionsgerade das Verhältnis aus Nutzerbewertungen und verschiedenen psychoakustischen Parametern abdeckt. Eine Streuung der Nutzerbewertungen führt dementsprechend ebenfalls zu einem geringeren Bestimmtheitsmaß.⁹⁰ Die Steigungsgerade der linearen Regression kann für einen Datensatz von x- & y-Werten mittels der Funktion `scipy.stats.linregress` aus der Python-Library „SciPy“ ermittelt werden. Mittels der ermittelten Funktion lassen sich nun für alle x-Werte (Anzahl der *Threads*) die Antwortzeiten prognostizieren. Anhand dieser prognostizierten Werte und den originalen Werten lässt sich das Bestimmtheitsmaß mittels der Funktion `sklearn.metrics.r2_score` aus der Python-Library „scikit-learn“ berechnen.

⁹⁰ Vgl. Sereinig 2010, S. 53; Sereinig 2010, S. 60

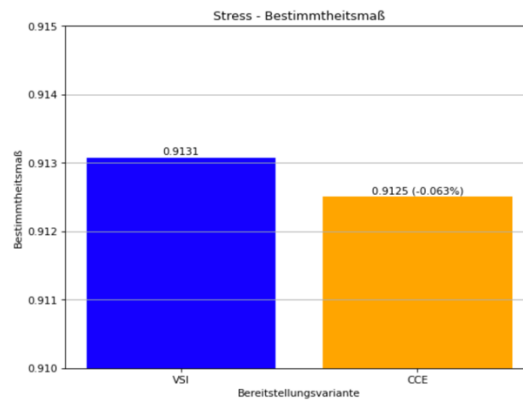


Abb. 21: Stresstest - Bestimmtheitsmaß

In Abbildung 21 ist der Unterschied des Bestimmtheitsmaß gemessen an der HTTP-Antwortzeit auf der y-Achse im Verhältnis zur *Threadanzahl* auf der x-Achse während des *Stresstests* dargestellt. Auf den ersten Blick scheint der Unterschied minimal. In der Tat ist der Unterschied sehr gering, so hat Code Engine ein 0.063% geringeres Bestimmtheitsmaß. Dies bedeutet entsprechend, dass Code Engine 0.063% mehr HTTP-Anfragen besitzt, deren Varianz im Rahmen des Bestimmtheitsmaß als unerklärt gelten. Diese Punkte entsprechen HTTP-Anfragen, die entweder unverhältnismäßig lange oder unverhältnismäßig kurz gedauert haben und weisen auf inkonsistente Antwortzeiten der Bereitstellungsmöglichkeit hin. In dem konkreten *Stresstest* von Code Engine bedeutet dies allerdings immerhin, dass circa 960 mehr solcher unerklärten HTTP-Anfragen existieren als eine vergleichbare Messung des Virtual Servers. Ob sich die bessere Konsistenz der HTTP-Antwortzeiten des Virtual Servers für die höheren durchschnittlichen Antwortzeiten im Vergleich zu Code Engine lohnt, muss für das jeweilige Projekt individuell entschieden werden. Zudem wurde bestätigt, dass in beiden Bereitstellungsmöglichkeiten ein starker linearer Zusammenhang zwischen der Anzahl der *Threads* und der Antwortzeit besteht.

4.4 Verursachte Kosten

Die verursachten Kosten haben zwar keine direkte Verbindung zur Forschungsfrage, sind aber dennoch für die praktische Implikation dieser Arbeit von Bedeutung. So verursachte Code Engine insgesamt 0.38€ über einen Zeitraum von 3 Monaten. Die beiden Virtual Server hingegen verursachten zusammen Kosten in Höhe von 134,13€ in demselben Zeitraum mit derselben Anzahl an ausgeführten Tests. Dies zeigt deutlich die Vorteile eines *Pay-as-you-use* Modells, wie es bei Code Engine implementiert ist, für das Bereitstellen von nicht permanenten *Workloads*. Die Virtual Server hingegen verursachte durchlaufende Kosten, auch wenn keine Tests liefen.

5 Schlusskapitel

5.1 Fazit

In der vorliegenden Arbeit wurden im Rahmen des Praxisteils drei verschiedene Versuche durchgeführt. Die Versuche beinhalten das Testen der zwei Bereitstellungsvarianten Code Engine und Virtual Server for VPC der IBM Cloud. Auf beiden *Services* wurde eine NodeJS-API mittels eines *Baselinetests*, *Loadtests* und *Stresstests* getestet. Die konkreten Konfigurationsoptionen dieser *Services* wurde beschrieben und erklärt.

Es wurde zudem ein Überblick gegeben, inwiefern die Tests mittels des API-Perfomancetestprogramms JMeter durchgeführt werden. Es wurde sowohl die konkrete Konfiguration der JMeter-Tests beschrieben, sowie die allgemeine Infrastruktur, in dessen Rahmen die Tests abliefen. Im Anschluss wurden die beschriebenen Tests durchgeführt und die Ergebnisse visuell dargestellt sowie detailliert erklärt.

In allen Tests war deutlich zu erkennen, dass Code Engine die performantere Bereitstellungsmöglichkeit bietet in Hinblick auf die durchschnittliche Antwortzeit der HTTP-Anfragen, sowie der gesamten Menge der bearbeiteten HTTP-Anfragen. Aufgrund dieser schnelleren Antwortzeit war es Code Engine möglich bis zu 12% mehr Anfragen in derselben Zeit zu bewältigen als der Virtual Server wie am Beispiel des *Stresstests* festzustellen war.

Während Code Engine zwar die HTTP-Anfragen schneller beantworten konnte, so ist dennoch festzuhalten, dass der Virtual Server die konsistenteren Messergebnisse liefern konnte. Dies konnte sowohl im *Baselinetest* mittels eines Box Plots als auch im *Stresstest* mittels linearer Regression und dem Bestimmtheitsmaß nachgewiesen werden.

Für die Beantwortung der anfangs gestellten Forschungsfrage gilt es also, die jeweiligen Anforderungen des Projektes zu beachten. Generell lässt sich behaupten, dass Code Engine die schnelleren Antwortzeiten bei gleicher Konfiguration liefert. Sollte jedoch die Konsistenz der Antwortzeiten als Priorität für die Infrastrukturwahl gelten, so ist der Virtual Server zu bevorzugen, wenn auch mit etwas langsameren HTTP-Antwortzeiten. Zudem gilt es in der Praxis die eventuell höheren Kosten des Virtual Servers in Erwägung zu ziehen.

Die anfangs aufgestellte Hypothese wurde somit widerlegt. Erstaunlicherweise konnte Code Engine trotz bereits bestehender Netzwerksicherheitsschichten grundsätzlich performantere Ergebnisse liefern als der Virtual Server, bei welchem weitere Konfigurationsschritte notwendig wären, um einen produktionsfertigen Standard zu erreichen.

5.2 Ausblick

In der vorliegenden Arbeit wurden die beiden Bereitstellungsmöglichkeiten anhand ihrer Performance verglichen. Allerdings wurden beide Alternativen auf eine einzelne Instanz

beschränkt. Dies ist unter realistischen Bedingungen außerhalb eines Versuchs nur selten der Fall. Wird eine hohe Last der Systeme erwartet, so skalieren moderne Applikationen horizontal, es werden also zusätzliche Instanzen gestartet. Für den Virtual Server könnte dies mittels einer *Autoscalinggroup* und einem *Loadbalancer* umgesetzt werden. In folgenden Arbeiten könnte die Performance der beiden Systeme erneut verglichen werden, diesmal jedoch mit Einbezug mehrerer Instanzen mittels automatischer Skalierung.

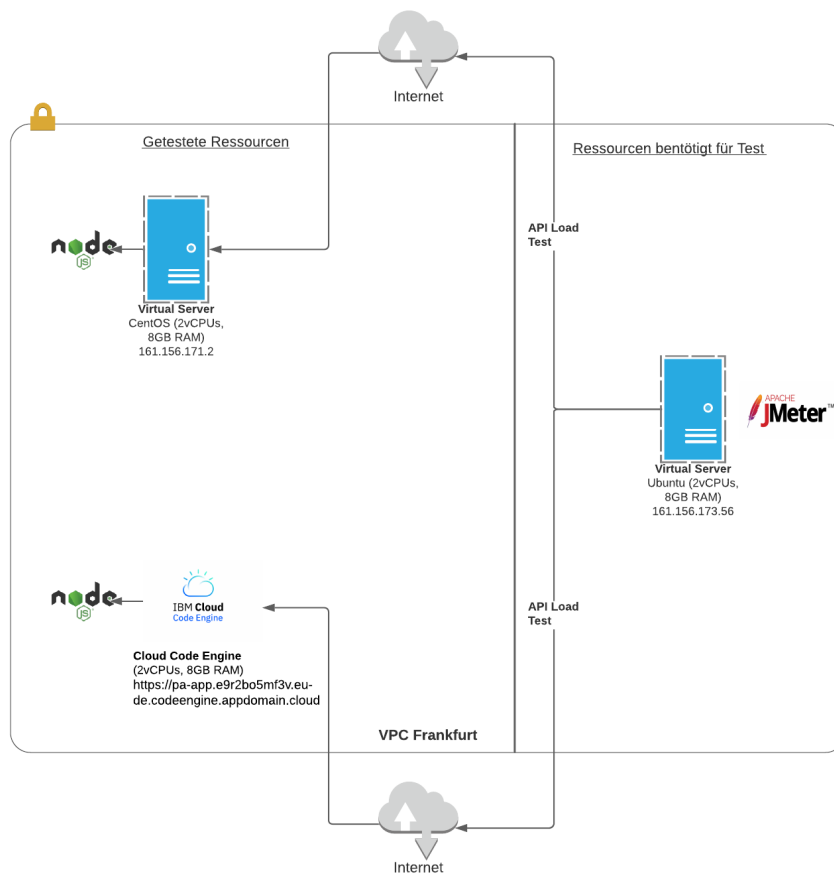
Zudem könnten die vorgestellten Tests wiederholt werden, sollte NodeJS, beziehungsweise Express, eine offizielle Unterstützung für die zweite Version des Protokolls HTTPS veröffentlichen. Unter Verwendung dieser könnte eine noch akkuratere Vergleichsgrundlage geschaffen werden, sowie untersucht werden, inwiefern die Version des HTTPS-Protokolls die Messergebnisse beeinflusst.

Zuletzt könnten in einer fortführenden Arbeit die bereits erwähnten, zusätzlichen API-Perfomancetests, wie beispielsweise der *Soaktest*, im Rahmen der beiden Bereitstellungsmöglichkeiten durchgeführt werden.

Anhang

Anhang 1: Performancetest Infrastruktur	38
Anhang 2: Baselinetest Konfiguration.....	38
Anhang 3: Baselinetest - Virtual Server (HTTP-Antwortzeit im Verhältnis zur Testdauer)	39
Anhang 4: Baselinetest – Code Engine (HTTP-Antwortzeit im Verhältnis zur Testdauer)	39
Anhang 5: Loadtest Konfiguration	40
Anhang 6: Loadtest – Virtual Server (HTTP-Antwortzeit im Verhältnis der aktiven Threads)	40
Anhang 7: Loadtest – Code Engine (HTTP-Antwortzeit im Verhältnis der aktiven Threads)	41
Anhang 8: Stresstest Konfiguration	41
Anhang 9: Stresstest – Virtual Server (HTTP-Antwortzeit im Verhältnis der aktiven Threads)	42
Anhang 10: Stresstest – Code Engine (1. Durchführung) (HTTP-Antwortzeit im Verhältnis der aktiven Threads)	42
Anhang 11: Stresstest – Code Engine (2. Durchführung) (HTTP-Antwortzeit im Verhältnis der aktiven Threads)	43

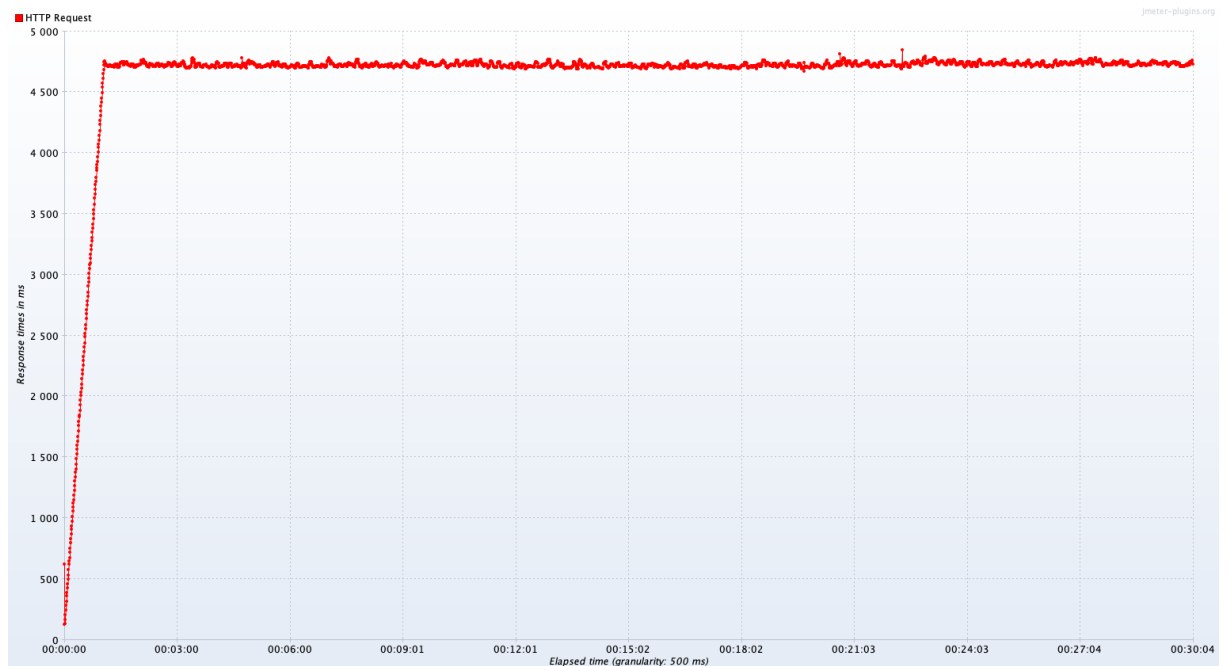
Anhang 1: Performancetest Infrastruktur



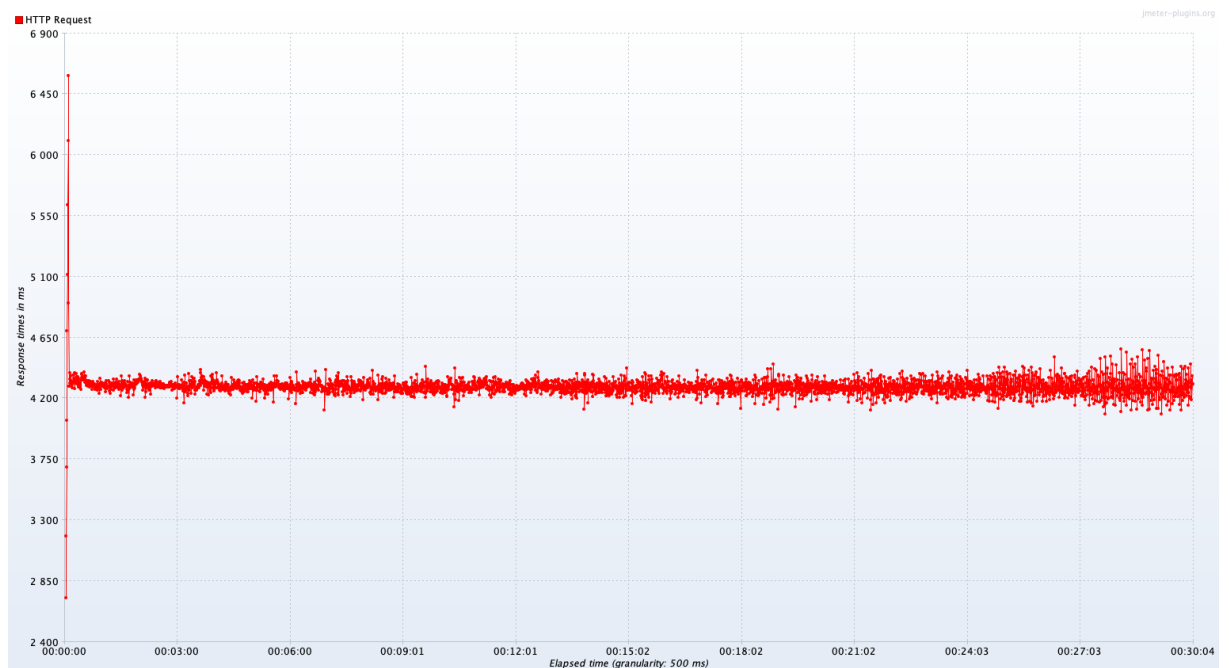
Anhang 2: Baselinetest Konfiguration

Einstellung	Code Engine Wert	VSI Wert
Threadanzahl (Nutzer)	150	150
Ramp-up-Period (Sekunden)	60	60
Loop Count	Infinite	Infinite
Dauer des Tests (Sekunden)	1800	1800
Protokoll	HTTPS2	HTTPS
Server Name / IP	pa-app.e9r2bo5mf3v.eu-de.codeengine.appdomain.cloud	161.156.171.2

Anhang 3: Baselinetest - Virtual Server (HTTP-Antwortzeit im Verhältnis zur Testdauer)



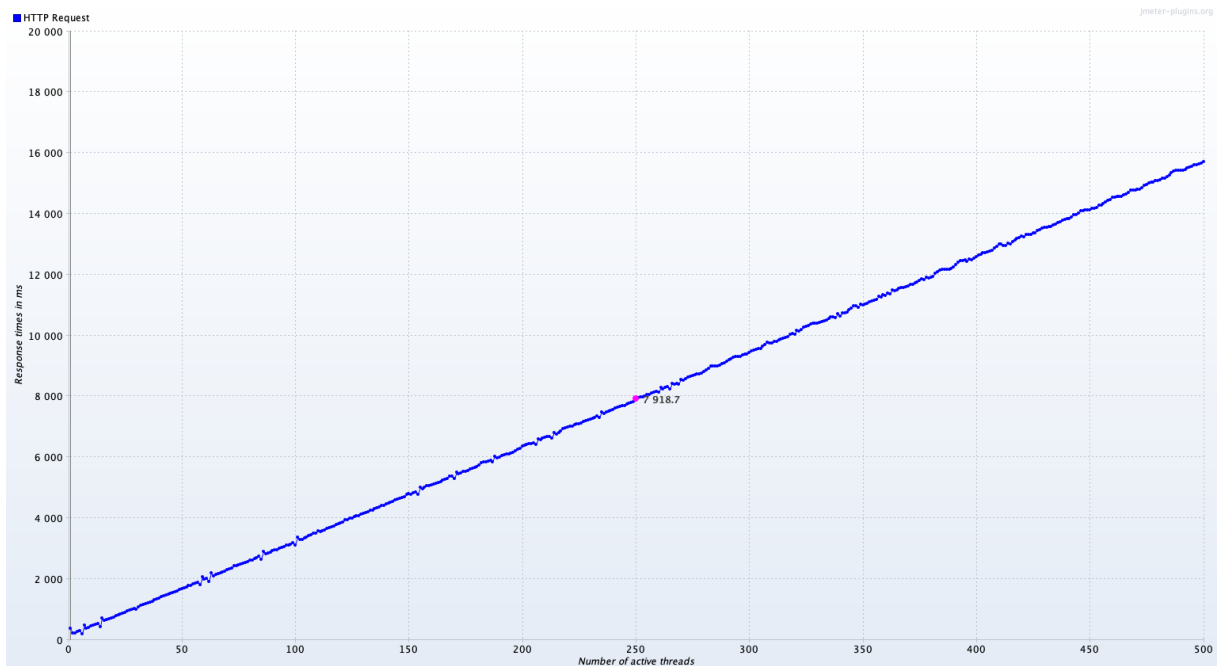
Anhang 4: Baselinetest – Code Engine (HTTP-Antwortzeit im Verhältnis zur Testdauer)



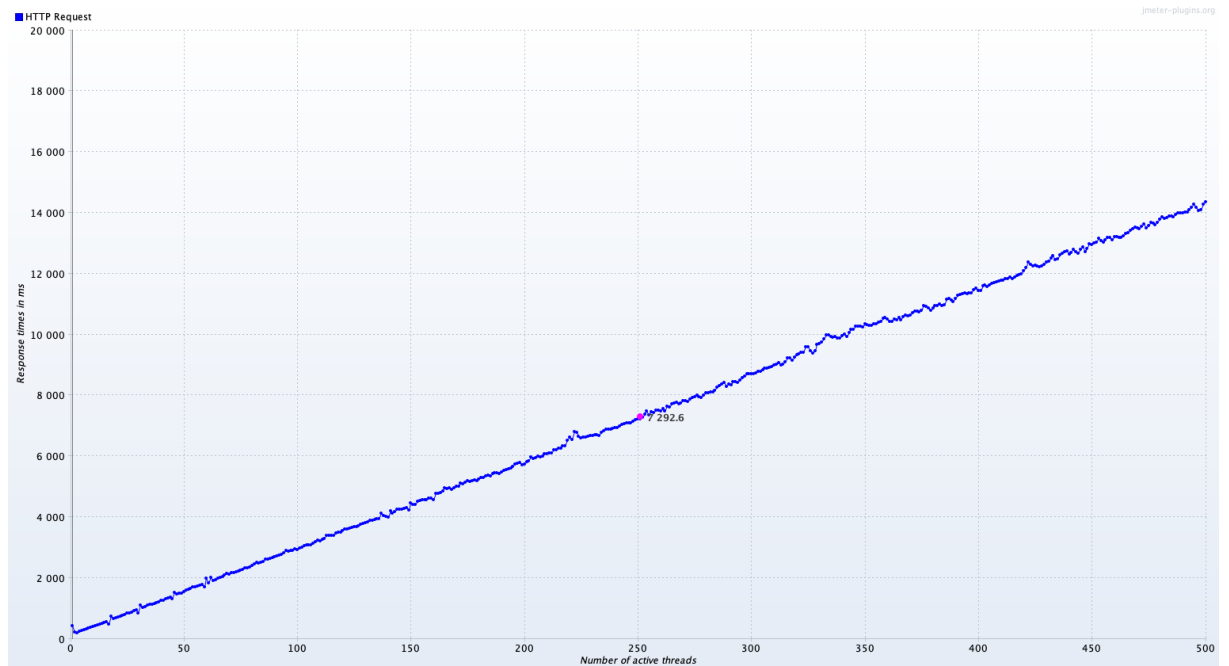
Anhang 5: Loadtest Konfiguration

Einstellung	Code Engine Wert	VSI Wert
Threadanzahl (Nutzer)	500	500
Ramp-up-Period (Sekunden)	1800	1800
Loop Count	Infinite	Infinite
Dauer des Tests (Sekunden)	1800	1800
Protokoll	HTTPS2	HTTPS
Server Name / IP	pa-app.e9r2bo5mf3v.eu-de.codeengine.appdo-main.cloud	161.156.171.2

Anhang 6: Loadtest – Virtual Server (HTTP-Antwortzeit im Verhältnis der aktiven Threads)



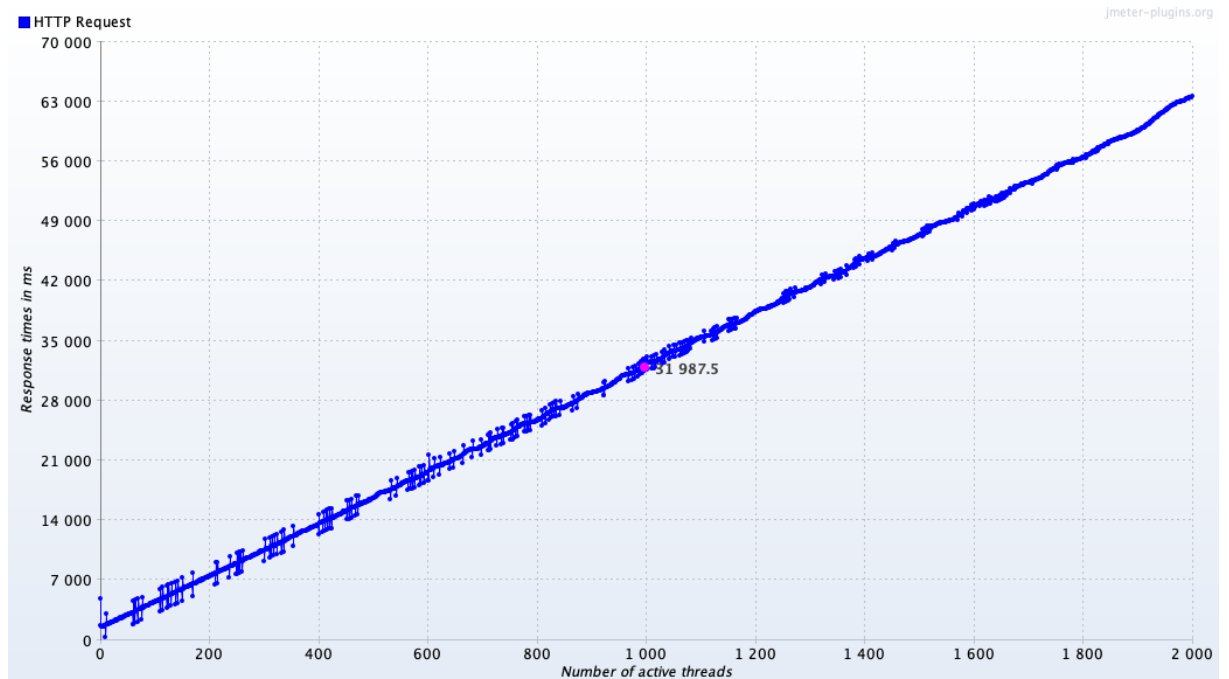
Anhang 7: Loadtest – Code Engine (HTTP-Antwortzeit im Verhältnis der aktiven Threads)



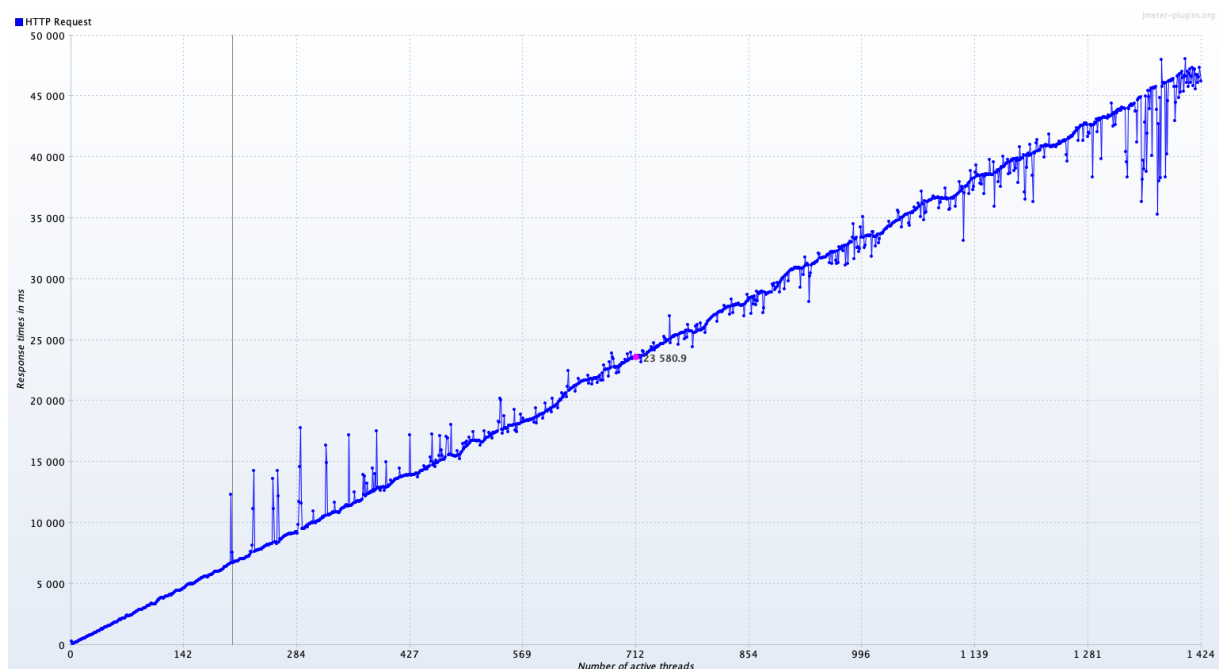
Anhang 8: Stresstest Konfiguration

Einstellung	Code Engine Wert	VSI Wert
Threadanzahl (Nutzer)	2000	2000
Ramp-up-Period (Sekunden)	2700	2700
Loop Count	Infinite	Infinite
Dauer des Tests (Sekunden)	2700	2700
Protokoll	HTTPS2	HTTPS
DNS / IP	pa-app.e9r2bo5mf3v.eu-de.codeengine.appdomain.cloud	161.156.171.2

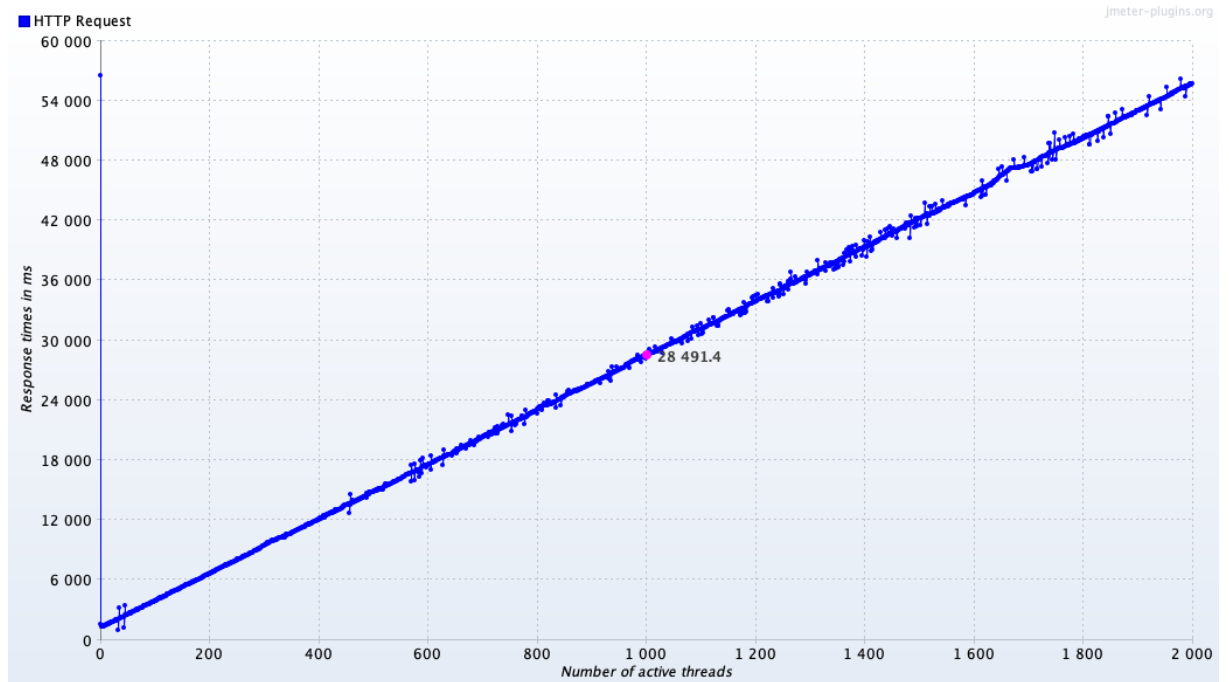
Anhang 9: Stresstest – Virtual Server (HTTP-Antwortzeit im Verhältnis der aktiven Threads)



Anhang 10: Stresstest – Code Engine (1. Durchführung) (HTTP-Antwortzeit im Verhältnis der aktiven Threads)



Anhang 11: Stresstest – Code Engine (2. Durchführung) (HTTP-Antwortzeit im Verhältnis der aktiven Threads)



Literaturverzeichnis

- Adzic, Gojko/Chatley, Robert (2017):** Serverless computing: economic and architectural impact, S. 884–889
- Amazon Web Services (2020):** Coca-Cola Freestyle Launches Touchless Fountain Experience in 100 Days Using AWS Lambda, <https://aws.amazon.com/solutions/case-studies/coca-cola-freestyle/>, Abruf: 01.11.2021
- Baldini, Ioana/Castro, Paul/Chang, Kerry/Cheng, Perry/Fink, Stephen/Ishakian, Vatche/Mitchell, Nick/Muthusamy, Vinod/Rabbah, Rodric/Slominski, Aleksander/Suter, Philippe (2017):** Serverless computing: Current trends and open problems, in: Research Advances in Cloud Computing, S. 1–20
- Barber, Scott (2010):** How Fast Does a Website Need To Be ?
- Bert, Jacob/Brown, Michael/Fukui, Kentaro/Trivedi, Nihar (2005):** Introduction to Grid Computing, in: Handbook of Research on Computational Methodologies in Gene Regulatory Networks, S. 28–55
- Bezemer, Cor Paul/Zaidman, Andy (2010):** Multi-tenant SaaS applications: Maintenance dream or nightmare?, in: ACM International Conference Proceeding Series, S. 88–92
- Bortz, Jürgen/Döring, Nicola (2006):** Forschungsmethoden und Evaluation
- Bressoud, Thomas/White, David (2020):** Introduction to Data Systems
- Castro, Paul/Ishakian, Vatche/Muthusamy, Vinod/Slominski, Aleksander (2019):** The rise of serverless computing, in: Communications of the ACM, 62. Jg., Nr. 12, S. 44–54
- Daniels, Jeff (2009):** Server virtualization architecture and implementation, in: XRDS: Crossroads, The ACM Magazine for Students, 16. Jg., Nr. 1, S. 8–12
- Davis, Megan (2005):** Virtual PC vs. Virtual Server: Comparing Features and Uses
- De, Brajesh (2017):** API Management
- Erinle, Bayo (2013):** Performance Testing with JMeter 2.9
- Google (o. J.a):** Google Trends, <https://trends.google.com/trends/explore?date=all&q=serverless>, Abruf: 20.09.2021
- Google (o. J.b):** What is a virtual server?, <https://cloud.google.com/learn/what-is-a-virtual-server>, Abruf: 03.10.2021
- Harba, Hind S (2015):** Design and Implementation of Multilevel Secure Database in Website, in: Journal of Information Engineering and Applications, 5. Jg., Nr. 12, S. 11–18
- Hasselbring, Wilhelm/Steinacker, Guido (2017):** Microservice architectures for scalability, agility and reliability in e-commerce, in: Proceedings - 2017 IEEE International Conference on Software Architecture Workshops, ICSAW 2017: Side Track Proceedings, S. 243–246

- Hellerstein, Joseph M./Faleiro, Jose/Gonzalez, Joseph E./Schleier-Smith, Johann/Sreekanti, Vikram/Tumanov, Alexey/Wu, Chenggang (2019):** Serverless computing: One step forward, two steps back, in: CIDR 2019 - 9th Biennial Conference on Innovative Data Systems Research, 3. Jg.
- IBM (2021b):** IBM Cloud Code Engine Pricing, <https://www.ibm.com/de-de/cloud/code-engine/pricing>, Abruf: 29.10.2021
- IBM (2021a):** IBM Virtual Server Pricing, <https://www.ibm.com/cloud/vpc/pricing>, Abruf: 29.10.2021
- Jain, Nancy/Choudhary, Sakshi (2016):** Overview of Virtualization in Cloud Computing, S. 283
- Jakl, Michael (2005):** Representational State Transfer (REST), in: Representational State Transfer, S. 633–672
- Jing, You/Zhang, Lan/Hongyuan, Wang/Yuqiang, Sun (2010):** JMeter-based Aging Simulation of Computing System
- Kaviani, Nima/Kalinin, Dmitriy/Maximilien, Michael (2019):** Towards serverless as commodity: A case of Knative, in: WOSC 2019 - Proceedings of the 2019 5th International Workshop on Serverless Computing, Part of Middleware 2019, S. 13–18
- Krol, Jason/Mithun, Satheesh/D'mello, Bruno (2014):** Web Development with MongoDB and Node.js
- Kurniawan, B/Java, JM is a (2003):** Using JMeter, S. 1–8
- Lai, Kevin/Baker, Mary (1998):** Measuring Bandwidth, 74. Jg., Nr. 1934, S. 535–546
- Mah, B A (1997):** An empirical model of HTTP network traffic, in: Proceedings of INFOCOM '97, S. 592–600 Bd.2
- Malhotra, L/Agarwal, D/Jaiswal, A (2014):** Virtualization in Cloud Computing, in: Journal of Information Technology & Software Engineering, 04. Jg., Nr. 02, S. 2–4
- Malvoni, Katja/Knezovic, Josip (2014):** Are your passwords safe: Energy-efficient Bcrypt cracking with low-cost parallel hardware, in: 8th USENIX Workshop on Offensive Technologies, WOOT 2014, Nr. Algorithm 1
- Meng, Michael/Steinhardt, Stephanie/Schubert, Andreas (2018):** Application programming interface documentation: What do software developers want?, in: Journal of Technical Writing and Communication, 48. Jg., Nr. 3, S. 295–330
- Microsoft Azure (2021):** Virtual machines: virtual computers within computers, <https://azure.microsoft.com/en-us/overview/what-is-a-virtual-machine/>, Abruf: 05.11.2021
- Molyneaux, Ian (2009):** The Art of Application Performance Testing
- Nevedrov, Dmitri (2006):** Using JMeter to Performance Test Web Services, in: Dev2Dev, S. 1–11

- NpmJS (o. J.):** node.bcrypt.js, <https://www.npmjs.com/package/bcrypt>, Abruf: 03.11.2021
- NumPy (2021):** NumPy Polyfit
- Ong, Shyue Ping/Cholia, Shreyas/Jain, Anubhav/Brafman, Miriam/Gunter, Dan/Ceder, Gerbrand/Persson, Kristin A. (2015):** The Materials Application Programming Interface (API): A simple, flexible and efficient API for materials data based on REpresentational State Transfer (REST) principles, in: Computational Materials Science, 97. Jg., S. 209–215
- Pautasso, Cesare/Wilde, Erik (2010):** RESTful web services, Nr. November, S. 1359
- Perez, Alfonso/Molto, German/Caballer, Miguel/Calatrava, Amanda (2016):** Serverless Computing for Container-based Architectures, 41. Jg., S. 80–86
- Rajan, Arokia Paul (2018):** Serverless Architecture - A Revolution in Cloud Computing, S. 93
- RedHat (2019):** What is Knative?, <https://www.redhat.com/en/topics/microservices/what-is-knative>, Abruf: 14.11.2021
- Reed, Jessie (2018):** Physical Servers vs. Virtual Machines: Key Differences and Similarities, <https://www.nakivo.com/blog/physical-servers-vs-virtual-machines-key-differences-similarities/>, Abruf: 30.09.2021
- Richards, Robert (2006):** Pro PHP XML and Web Services
- Rodrigues, A G/Demion, B/Mouawad, P (2019):** Master Apache JMeter - From Load Testing to DevOps: Master performance testing with JMeter, Packt Publishing
- Sereinig, Roman (2010):** Untersuchung akustischer Beschreibungsmethoden für Haushaltsgeräte, Nr. April
- Sriramya, P./Karthika, R. A. (2015):** Providing password security by salted password hashing using Bcrypt algorithm, in: ARPN Journal of Engineering and Applied Sciences, 10. Jg., Nr. 13, S. 5551–5556
- Sun, Haiyang/Schiavio, Filippo/Bonetta, Daniele/Binder, Walter (2019):** Reasoning about the Node.js Event Loop using Async Graphs
- Williamson, D. F./Parker, R. A./Kendrick, J. S. (1989):** The box plot: A simple visual method to interpret data, in: Annals of Internal Medicine, 110. Jg., Nr. 11, S. 916–921
- Zerouali, Ahmed (2018):** Analyzing technical lag in docker images, in: CEUR Workshop Proceedings, 2361. Jg., S. 6–10
- Zhuang, Yu (2021a):** Introduction to IBM Cloud Code Engine, <https://developer.ibm.com/series/ibm-cloud-code-engine-managed-serverless-platform/>, Abruf: 29.10.2021
- Zhuang, Yu/Werner, Jerermias/Xiang Cui, Xue (2021b):** Run an application on a managed serverless platform, <https://developer.ibm.com/articles/ibm-cloud-code-engine-run-application-on-managed-serverless-platform/>, Abruf: 29.10.2021

Erklärung

Ich versichere hiermit, dass ich meine Projektarbeit mit dem Thema: *„Performancevergleich einer NodeJS-API Bereitstellung auf IBM Cloud Code Engine mit der Bereitstellung auf IBM Virtual Server for VPC anhand von HTTP-Requests“* selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe. Ich versichere zudem, dass die eingereichte elektronische Fassung mit der gedruckten Fassung übereinstimmt.

(Ort, Datum)

(Unterschrift)