

Hochschule Darmstadt

– Fachbereich Informatik–

Performance Vergleich zwischen Containerized Deployments und Serverless Functions

Abschlussarbeit zur Erlangung des akademischen Grades

Bachelor of Science (B.Sc.)

vorgelegt von

Robin Luley

Matrikelnummer: 759298

Referent : Prof. Dr. Daniel Burda

Korreferent : Prof. Dr. Benjamin Meyer

ERKLÄRUNG

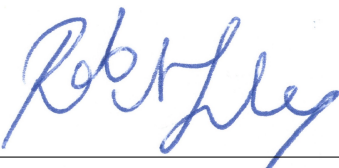
Ich versichere hiermit, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die im Literaturverzeichnis angegebenen Quellen benutzt habe.

Alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten oder noch nicht veröffentlichten Quellen entnommen sind, sind als solche kenntlich gemacht.

Die Zeichnungen oder Abbildungen in dieser Arbeit sind von mir selbst erstellt worden oder mit einem entsprechenden Quellennachweis versehen.

Diese Arbeit ist in gleicher oder ähnlicher Form noch bei keiner anderen Prüfungsbehörde eingereicht worden.

Darmstadt, 25. Februar 2021



Robin Luley

ZUSAMMENFASSUNG

In dieser wissenschaftlichen Arbeit wird die Performance von Serverless- und containerisierten Systemen anhand eines beispielhaften REST-Backends praktisch evaluiert und verglichen. Dazu wird zunächst eine klassische Notiz-Applikation entwickelt, die anschließend sowohl auf AWS Lambda als auch auf AWS Elastic Container Service mit dem Starttyp Fargate deployed wird. Im Anschluss folgt die Konzeption einer Test-Architektur, mit der die Performance der beiden Services untersucht und verglichen werden kann. Daraufhin werden die in der Konzeption erstellten Tests praktisch durchgeführt. Dabei werden jeweils ähnliche Konfigurationen bezüglich Arbeitsspeicher und CPU-Größe der Systeme in Relation gesetzt. Schlussendlich sollen die Kosten für den Betrieb der beiden Services gegenübergestellt werden.

Bei der Auswertung der Tests zeigt sich, dass die Container-Anwendung der Lambda-Variante in Hinblick auf die gemessenen Antwortzeiten um einige Millisekunden überlegen ist und eine geringere Varianz aufweist. Bei den Lambda-Funktionen kommt es in den Experimenten abhängig vom getesteten Anwendungsfall, der Schnelle des Last-Anstiegs und der ausgewählten Konfigurationsgröße zu geringen oder größeren Abweichungen in den Antwortzeiten. Es wird daher geschlussfolgert, dass sich Container besser für besonders zeitkritische Anwendungen mit hohen Anforderungen an konsistente Response-Times eignen, Serverless am Beispiel von Lambda aber durch ein solides automatisches Skalierungsverhalten überzeugt. Allgemein fällt die Differenz zwischen den Systemen jedoch gering aus. In der Untersuchung der Kosten zeigt sich, dass die Nutzung von Container preislich erst bei einer hohen Systemauslastung rentabel ist und daher der Betrieb von Serverless-Funktionen vor allem bei wenigen Anwendungsnutzern zu empfehlen ist.

INHALTSVERZEICHNIS

I THESIS

1	EINLEITUNG	2
1.1	Motivation	2
1.2	Forschungsfrage und Zielsetzung	2
1.3	Gliederung	3
2	THEORETISCHE GRUNDLAGEN UND VERWANDTE ARBEITEN	4
2.1	Containerisierung	4
2.2	Container-Orchestrierung	5
2.3	Serverless	5
2.4	AWS Lambda	5
2.4.1	Cold- und Warmstart	7
2.4.2	Nebenläufigkeit	7
2.5	Verwandte Arbeiten	8
3	KONZEPTION	10
3.1	Methodisches Vorgehen	10
3.2	Konzeption der Test-Anwendungen	11
3.2.1	Serverless	12
3.2.2	Container	13
3.3	Konzeption der Tests	14
3.3.1	Testing-Tool	14
3.3.2	Metriken	15
3.3.3	Test-Typen	16
3.3.4	Getestete Use-Cases	16
3.3.5	Ablauf der Tests	17
4	ANALYSE	18
4.1	128MB Konfigurationen	18
4.1.1	Pipe-Clean Tests	18
4.1.2	Stress-Tests	19
4.1.3	Load-Tests	24
4.2	Andere Konfigurationen (RQ3)	28
4.2.1	Pipe-Clean Tests	28
4.2.2	Stress- und Load-Tests	29
4.3	Mehrere Container (RQ4)	32
4.4	Kosten	33
4.4.1	Container	34
4.4.2	Lambda	35
5	DISKUSSION DER ERGEBNISSE	37
5.1	Beantwortung der Forschungsfragen (RQ1 - RQ5)	38
5.2	Skalierung und Optimierung	40
5.3	Kosten	42
5.4	Implikationen für die Praxis	43

6	ZUSAMMENFASSUNG UND AUSBLICK	45
	LITERATUR	47
II	APPENDIX	
A	TASK-DEFINITIONEN	52
A.1	128MB Container	52
A.2	256MB Container	55
A.3	512MB Container	57
B	REPOSITORY	61
B.1	Anwendungs-Quellcode	61
B.1.1	Container	61
B.1.2	Serverless	61
B.2	Test Dateien	61
B.2.1	Test Skripte und Konfiguration	62
B.2.2	Test-Artefakte	62
B.2.3	Python Analyse	63

ABBILDUNGSVERZEICHNIS

Abbildung 2.1	Vergleich zwischen Containern und Virtuellen Maschinen	4
Abbildung 2.2	AWS Lambda Funktionsweise	6
Abbildung 3.1	Testing-Architektur der Serverless Notiz-Anwendung .	12
Abbildung 3.2	Testing-Architektur der Container Notiz-Anwendung .	14
Abbildung 4.1	Vergleich der Pipe-Clean Tests für 128MB	18
Abbildung 4.2	Beispiel-Anstieg der gleichzeitigen VUs für einen Stress-Test	20
Abbildung 4.3	Container 128MB Stress-Test Vergleich	20
Abbildung 4.4	Container Use-Case A 128MB Stress-Test mit 1000 VUs Verlauf Vergleich	21
Abbildung 4.5	Lambda 128MB Stress-Test Vergleich	22
Abbildung 4.6	Lambda 128MB Stress-Test mit 1.000 VUs Verlauf Vergleich	23
Abbildung 4.7	Load- und Spike-Test mit 600 VUs Vergleich	24
Abbildung 4.8	Lambda Use-Case A Load- und Spike-Test mit 600 VUs Verlauf Vergleich	26
Abbildung 4.9	Lambda Use-Case B Load- und Spike-Test mit 600 VUs Verlauf Vergleich	26
Abbildung 4.10	Lambda Load-Test mit 600 VUs Verlauf Vergleich . . .	27
Abbildung 4.11	Vergleich der Pipe-Clean Tests für Lambda	28
Abbildung 4.12	Lambda 256MB Stress-Test mit 1.000 VUs Verlauf Vergleich	29
Abbildung 4.13	Lambda 128MB und 256MB Load- und Spike-Tests mit 600 VUs Verlauf Vergleich	30
Abbildung 4.14	Lambda 512MB Load- und Spike-Tests mit 1.200 VUs Verlauf Vergleich	32
Abbildung 4.15	Vergleich der Load-Tests für 2x256MB und 512MB mit 1.200 VUs	33
Abbildung 4.16	Anzahl der Aufrufe von Lambda-Funktionen dem Monatspreis von Fargate entsprechend	36

ABKÜRZUNGSVERZEICHNIS

AWS	Amazon Web Services
GCP	Google Cloud Platform
ECS	AWS Elastic Container Service
FaaS	Functions-as-a-Service
BaaS	Backends-as-a-Service
S3	Amazon Simple Storage Service
EC2	AWS Elastic Compute Cloud
ELB	Elastic Load Balancer
VU	Virtual User
CSV	Comma Separated Values
SUT	System-under-Test
HPA	Horizontal Pod Autoscaler
EKS	Elastic Kubernetes Service
vCPU	Virtual CPU

Teil I

THESIS

EINLEITUNG

In diesem Kapitel wird die Motivation und Zielsetzung dieser Arbeit erläutert und auf die Gliederung eingegangen.

1.1 MOTIVATION

Der Betrieb von Anwendungen in der Cloud erfährt steigende Aufmerksamkeit. Public-Cloud Anbieter wie Amazon Web Services ([AWS](#)), Microsoft Azure oder Google Cloud Platform ([GCP](#)) erlauben es Organisationen, schneller und günstiger ihre Anwendungen zu provisionieren. Schon ein einfaches Lift-and-shift von lokalen Rechenzentren hin zu virtuell bereitgestellten Server-Instanzen bedeutet einige Vorteile in puncto Agilität, Kosteneinsparungen und Ausfallsicherheit gegenüber On-Premises Infrastruktur. Die Einführung von Containerisierung und Orchestrierungs-Tools wie Kubernetes machte es möglich, Anwendungen einfacher zu skalieren und ausfallsicherer zu machen. Doch die von den Anbietern bereitgestellten Services werden immer raffinierter, um das Betreiben von Software weiterhin zu verbessern und zu erleichtern. Während der Betrieb von containerisierten Anwendungen noch eine manuelle Konfiguration gewisser Parameter erfordert, werden Serverless-Anwendungen fast vollständig von den Cloud-Anbietern gemanaged und der Server somit weg abstrahiert. Sie bieten dadurch eine theoretisch unbegrenzte Skalierbarkeit ohne aufwändige Konfigurierung und Verwaltung von Infrastruktur. Zudem verspricht Serverless durch ein flexibles Kostenmodell, bei dem nur für tatsächlich ausgeführten Code gezahlt wird, weitere Kosteneinsparungen. Allerdings ist bisher noch unklar, wie sich die Performance von Serverless-Anwendungen im Vergleich zu containerisierten Anwendungen verhält. Dieses Problem soll in dieser Arbeit untersucht werden.

1.2 FORSCHUNGSFRAGE UND ZIELSETZUNG

Ziel der Arbeit ist es, die Performance von containerisierten und Serverless-Anwendungen anhand eines REST-Backends zu vergleichen. Als Cloud-Anbieter wird [AWS](#) verwendet. Es soll eine Beispielanwendung als REST-API für beide Technologien angepasst entwickelt werden und verschiedenen Performance-Tests unterzogen werden. Dabei wird für die Serverless-Applikation [AWS](#) Lambda verwendet und für die containerisierte Anwendung das Orchestrierungs-Tool AWS Elastic Container Service ([ECS](#)). Im Anschluss wird auf die Kostenmodelle beider Services eingegangen und diese ins Verhältnis gesetzt.

Die Forschungsfrage lautet: Wie unterscheidet sich der Betrieb von containerisierten und Serverless-Anwendungen in der Cloud bezüglich ihrer Performance und Kosten am Beispiel eines REST-Backends auf [AWS](#) Lambda und [ECS](#)?

1.3 GLIEDERUNG

Zunächst wird auf die Grundlagen von Containerisierung und Serverless-Funktionen am Beispiel von [AWS](#) Lambda eingegangen. Es wird ein Konzept erstellt, mit dem die beiden Technologien evaluiert werden können. Schließlich wird die Evaluation praktisch durchgeführt und die Ergebnisse präsentiert und diskutiert.

THEORETISCHE GRUNDLAGEN UND VERWANDTE ARBEITEN

In diesem Kapitel werden grundlegende Begriffe erklärt, die für das Verständnis der Arbeit unabdingbar sind.

2.1 CONTAINERISIERUNG

Containerisierung ist eine Technik, den Code einer Anwendung mitsamt aller Abhängigkeiten, die zur deren Ausführung benötigt werden, zu verpacken. Ein solches Paket wird als Container-Image bezeichnet. Da alle Abhängigkeiten enthalten sind, werden dadurch die Laufzeitunterschiede auf unterschiedlichen Plattformen minimiert[43]. Zur Laufzeit wird ein Container-Image als Container bezeichnet. Um ein Container-Image auszuführen, wird eine Container-Runtime benötigt. Die erste Container Runtime-Engine wurde von Docker entwickelt, wird aber mittlerweile als Open-Source Projekt mit dem Namen containerd von der Open Container Initiative (OCI) betrieben[43].

Containerisierte Anwendungen unterscheiden sich von Virtuellen Maschinen vor allem darin, dass kein Hypervisor zur Ausführung der Images benötigt wird. Stattdessen teilen sich alle Container das Host-Betriebssystem und werden in einem eigenen User-Space isoliert. Dadurch sind Container-Images deutlich leichtgewichtiger und können schneller als Virtuelle Maschinen ausgeführt werden[43]. Abbildung 2.1 verdeutlicht nochmals den Unterschied beider Technologien.

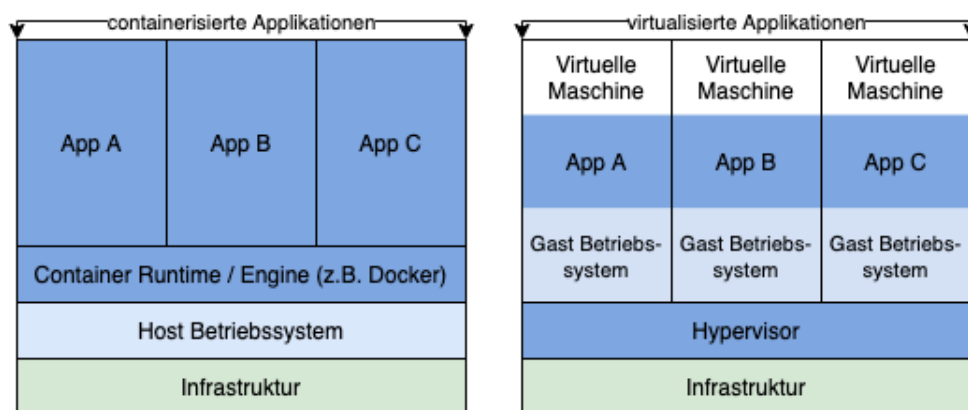


Abbildung 2.1: Vergleich zwischen Containern und Virtuellen Maschinen (angelehnt an: [43])

2.2 CONTAINER-ORCHESTRIERUNG

Container-Orchestrierung beschreibt die Automatisierung der „Bereitstellung, Verwaltung, Skalierung und Vernetzung von Containern“[38]. Mit Orchestrierungs-Tools lassen sich Container einfach auf Computer-Clustern provisionieren, skalieren und überwachen. Sie ermöglichen das Betreiben von infrastrukturunabhängigen und flexiblen Anwendungen auf Basis von Containern[39]. Beispiele für bekannte Orchestrierungs-Tools sind Kubernetes oder ECS, welcher in dieser Arbeit verwendet wird.

2.3 SERVERLESS

Serverless ist ein Software-Ausführungsmodell, bei dem ein Cloud Anbieter (z.B AWS, Microsoft Azure, GCP) sich vollständig um die Ausführung einer Anwendung inklusive Konfiguration der zugrundeliegenden Server und deren Skalierung kümmert. Ein Programmierer von Serverless-Anwendungen kann sich daher ausschließlich auf das Schreiben des Codes konzentrieren und muss keine Zeit in das Einstellen und Aufsetzen von Servern investieren[21].

Serverless Anwendungen werden grundlegend unterschieden in Functions-as-a-Service (FaaS) und Backends-as-a-Service (BaaS); wobei es sich bei letzterem um API-basierte, automatisch skalierte Services von Drittanbietern handelt, die grundlegende Bausteine einer Applikation ersetzen[21]. Beispiele für BaaS sind Amazon Simple Storage Service (S3) als Speicher von Dateien oder AWS DynamoDB als NoSQL-Datenbank.

In dieser Arbeit sollen ausschließlich FaaS betrachtet werden. Bei diesem Ausführungsmodell wird die Software in einem Stateless-Container, d.h einem Container ohne persistentem Speicher, ausgeführt. Der Cloud-Anbieter stellt dabei die Ressourcen dynamisch bzw. on-demand bereit, d.h die Anwendung wird nur dann ausgeführt, wenn eine diesbezügliche Anfrage, also eine Ereignis, eintrifft. Serverless verspricht dadurch eine effizientere Entwicklung und Kosteneinsparungen, da von den Anbietern nur der tatsächlich laufenden Code abgerechnet wird[40].

2.4 AWS LAMBDA

Die in dieser Arbeit getestete Serverless-Anwendung wird auf Lambda, einem FaaS-Angebot von AWS, ausgeführt. Die Funktionsweise des Lambda-Services lässt sich wie folgt beschreiben (alle Informationen dieser Sektion basieren auf dem AWS Lambda Entwicklerhandbuch[7]):

1. Es tritt ein Event ein, dass die Ausführung der Lambda-Funktion anfordert.
Dabei kann es sich um eines von vielen möglichen Ereignissen handeln, bspw. ein Dateiupload in S3, einen CRON-Job oder, im Falle eines Web-Backends, ein API-Aufruf durch ein API-Gateway.

2. Init Phase: Lambda erstellt automatisch eine Ausführungsumgebung (environment).

Dabei handelt es sich um eine isolierte Umgebung, die für die Ausführung der Lambda- Funktion benötigt wird. Für diese Umgebung lassen sich einige Parameter vom Anwender konfigurieren:

- a) Die Runtime: Dabei handelt es sich um die verwendete Programmiersprache bzw. das verwendete Framework. Als Optionen bietet Lambda beispielsweise Node.js, Java, Python oder Go an.
- b) Die Arbeitsspeichergröße: Sie bestimmt, wie viel Speicher der ausgeführten Funktion bereitsteht. Die Mindestgröße beträgt hier 128 Megabyte. Maximal sind etwas mehr als 10 Gigabyte möglich. Proportional zur Speichergröße legt Lambda ebenfalls die CPU-Leistung fest, mit der die Funktion ausgeführt wird.
- c) Timeout: Die maximale Ausführungszeit der Funktion bevor Lambda sie automatisch beendet. Hier sind maximal 15 Minuten möglich.

Die Umgebung wird mit den konfigurierten Ressourcen (Arbeitsspeicher und CPU) erstellt, der Code der Funktion aus [S3](#) in den Funktions-Container geladen und entpackt und der Initialisierungscode (nicht die Funktion an sich) ausgeführt. An dieser Stelle lässt sich bspw. eine Datenbankverbindung aufbauen.

3. Invoke Phase: In dieser Phase wird die eigentliche Lambda-Funktion, welche auch als Handler bezeichnet wird, ausgeführt. Eventuelle Rückgabewerte werden an den Aufrufer zurückgeliefert. Nachdem der Handler ausgeführt wurde, bleibt er verfügbar für weitere Anfragen. Der Initialisierungscode wird allerdings nicht mehr ausgeführt.
4. Shutdown Phase: Nachdem die Funktion für einige Zeit nicht mehr angefragt wurde, wird die Ausführungsumgebung wieder freigegeben.

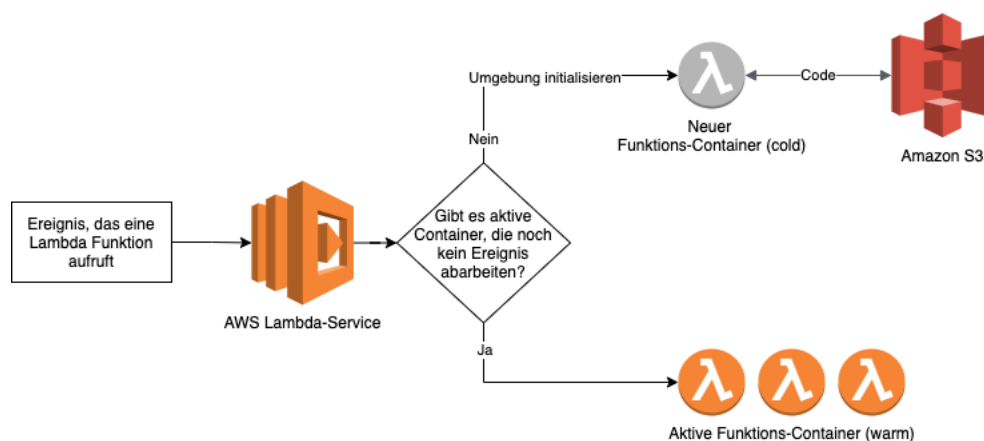


Abbildung 2.2: AWS Lambda Funktionsweise (angelehnt an: [\[35\]](#))

2.4.1 *Cold- und Warmstart*

Nachdem der Handler in der Invoke-Phase ausgeführt wurde, wird die Ausführungsumgebung nicht sofort wieder freigegeben, sondern für einige Zeit vorgehalten, um weitere Anfragen schneller zu bearbeiten. Dies wird auch als Warmstart bezeichnet, da die Runtime-Umgebung bereits initialisiert ist und der Handler sofort aktiviert werden kann. Muss die Umgebung nach einer eingetroffenen Anfrage erst noch initialisiert werden, also zunächst die Init-Phase ausgeführt werden, wird dies als Coldstart bezeichnet. Aufgrund des Mehraufwandes, erweist sich ein Coldstart als deutlich langsamer als ein Warmstart.

2.4.2 *Nebenläufigkeit*

Befindet sich der Handler einer Lambda-Funktion gerade in der Ausführung und es trifft ein weiteres Ereignis ein, startet Lambda zusätzlich zu der bereits laufenden Funktion eine weitere Ausführungsumgebung, um diese neue Anfrage zu verarbeiten. Da nun mehrere Umgebungen gleichzeitig ausgeführt werden, wird dies auch als Nebenläufigkeit (engl. concurrency) bezeichnet. Bei vielen gleichzeitigen Anfragen werden also so viele Umgebungen wie nötig erstellt. Der [AWS](#) Lambda-Service erlaubt standardmäßig 1.000 nebenläufige Funktionen pro Region, dieser Wert lässt sich allerdings nach Absprache mit [AWS](#) erhöhen.

Treten vermehrt Anfragen in kürzerer Zeit auf, werden automatisch neue Instanzen aufgesetzt, die diese bearbeiten können. Bereits initialisierte Umgebungen werden wiederverwendet. Lässt die Nachfrage nach, werden Umgebungen automatisch wieder heruntergefahren. [Abbildung 2.2](#) verdeutlicht nochmals den gesamten Prozess, in dem Lambda neue Funktions-Container erstellt oder bereits gestartete wiederverwendet.

2.5 VERWANDTE ARBEITEN

Serverless ist allgemein ein noch junges Thema, das erst mit der Einführung von [AWS Lambda](#) im Jahre 2014 Aufmerksamkeit erfuhr[21]. Es gibt bereits einige Studien zu der Performance von Serverless-Anwendungen, in den meisten Studien wurden jedoch nur Micro-Benchmarks betrachtet, d.h. Tests, die nur einen Aspekt untersuchen, bspw. die CPU-Floating-Point Performance und keine realistischen Applikationen untersuchen[34]. Vergleiche mit der Performance von Container-Anwendungen sind nur wenige zu finden.

McGrath und Brenner untersuchten die Performance von Serverless-Funktionen durch mehrere verschiedene Tests[27]. Unter anderem wurde ein Nebenläufigkeits-Test (concurrency test) vorgestellt, bei dem eine Funktion, die sofort terminiert, von einer linear ansteigenden Anzahl an Clients so oft wie möglich hintereinander aufgerufen wurde. Gemessen wurde dabei die Anzahl der Antworten pro Sekunde (responses per second). Ziel dieses Tests war, die performante Ausführung einer skalierten Funktion zu messen. Das Ergebnis zeigte unterschiedliches Skalierungsverhalten für diverse Cloud-Anbieter; bei [AWS Lambda](#) wurde ein weitestgehend linearer Anstieg festgestellt. Für zukünftige Forschung werden von den Autoren noch andere die Performance von Serverless-Funktionen beeinflussende Aspekte genannt, wie die Ausführung mehrerer Funktionen anstatt nur einer einzigen (single function execution), die Code-Größe einer Serverless-Funktion und unterschiedliche Funktionsgrößen (in der Studie wurde lediglich 512MB RAM verwendet).

Hendrickson et al. verglichen in ihrer Arbeit die Performance von [AWS Lambda](#) mit der eines in [AWS Beanstalk](#) laufenden Containers[16]. Dazu sendeten die Forscher jeweils 100 RPC-Requests an beide Services, wobei jeder Service eine Laufzeit von 200ms beanspruchte. Die Durchführung des Tests ergab, dass die Antwortzeit bei Lambda mit einem Medianwert von 1,6 Sekunden deutlich vor der von Beanstalk mit bis zu 20 Sekunden liegt. Als Grund dafür wird angeführt, dass Lambda innerhalb von einigen Millisekunden 100 Instanzen starte, auf die die Last verteilt wurde, während bei Beanstalk alle Anfragen von einer einzigen Instanz bearbeitet werden mussten. Denn die Beanstalk-Anwendung wurde nicht hoch skaliert, obwohl alle Einstellungen getroffen wurden, damit die Skalierung so schnell wie möglich erfolgen kann. Während es laut den Forschern bei Beanstalk zwanzig verschiedene Einstellungsmöglichkeiten zum Skalierungsverhalten gäbe, übernehme Lambda dies vollkommen automatisch. Allerdings wird auch deutlich, dass die Latenz von Lambda-Funktionen bei einer normalen Last deutlich über der eines Containers liegt. Mit der gleichen Testumgebung und unter leichter Last, performe Lambda zehn mal schlechter als Beanstalk. Da dieser Test im Jahre 2017 durchgeführt wurde, stimmen diese Ergebnisse aber vermutlich nicht mit den in der Zwischenzeit an Lambda vorgenommenen Performance-Verbesserungen überein.

Villamizar et al. verglichen die Performance und infrastrukturellen Kosten einer auf verschiedene Arten deployten Applikation[37]. Die Anwendung wurde als Monolith und als Microservices-Architektur sowohl mit, als auch ohne [AWS](#) Lambda betrieben. Dabei zeigte sich, dass die Kosten für den Betrieb bei einer auf Lambda basierenden Microservices-Architektur mehr als 70 Prozent geringer ausfielen, als bei einem Monolithen oder wenn die Microservices von den Entwicklern manuell gemanaged wurden. Ebenfalls wurde die Performance der verschiedenen Ansätze mittels der Metrik der durchschnittlichen Antwortzeit (average response time - ART) verglichen. Es wurde deutlich, dass Lambda in unterschiedlichen Test-Szenarien nahezu die gleich ART beibehält, was auf dessen Skalierungsmöglichkeiten zurückgeführt wurde. Des Weiteren performte Lambda teilweise besser als die monolithische Architektur und deutlich besser als die selbst-gemanagten Microservices.

Verschiedene Server-Hosting Technologien, darunter auch [AWS](#) Lambda und [AWS](#) Fargate, wurden 2020 von Jain et al. auf ihre Performance hin untersucht[20]. Ähnlich zu dieser Arbeit wurde eine Notiz-Anwendung deployed, allerdings handelte es sich dabei um eine Frontend-Anwendung. Als Test-Metriken wurde sich demnach ausschließlich auf solche beschränkt, die das Rendering der Applikation im Web-Browser des Endbenutzer betreffen, bspw. die Page-Load-Time und Start-Render-Time. Dies steht im Gegensatz zu dieser Arbeit, bei dem ausschließlich Backend-Technologien miteinander verglichen werden. Die Ergebnisse der Arbeit zeigten dennoch, dass [ECS](#) mit Fargate deutlich besser performte als [AWS](#) Lambda und auch [ECS](#) mit [AWS](#) Elastic Compute Cloud ([EC2](#)) übertraf. Es wurden jedoch weder Stress-Tests der Anwendungen durchgeführt, noch die Kostenunterschiede untersucht.

Alex DeBrie untersuchte 2019 in einem Artikel auf seiner Blog-Seite die Performance von [ECS](#) Fargate-Containern und [AWS](#) Lambda[13]. Dazu wurde ähnlich wie in dieser Arbeit ein HTTP-Endpunkt deployed und 15.000 Requests an jede Variante gesendet. Die HTTP-Endpunkte sendeten dann den Payload des Requests an den Amazon Simple Notification Service (SNS) bevor die Antwort zurückgesendet wurde. Zur Auswertung wurden Percentile der Antwortzeit verwendet. Es zeigte sich, dass die Fargate-Container den Lambda-Funktionen deutlich in der Performance überlegen waren. Allerdings wurde nur eine einzige Konfiguration für beide Technologien getestet.

Nach der Betrachtung der relevanten Arbeiten und Artikel zeigt sich also, dass es bisher noch keine Studie gibt, die die Performance von Containern und Serverless-Funktionen anhand mehrerer Konfigurationen und mit verschiedenen Performance-Tests im Kontext eines HTTP-REST-Backends evaluiert. Diese Lücke versucht die vorliegende Arbeit zu schließen, indem reproduzierbare Test mit verschiedenen Konfigurationen durchgeführt werden und auch auf die Kosten der beiden Technologien eingegangen wird.

KONZEPTION

In diesem Kapitel wird die grundlegende Konzeption der Arbeit vorgestellt.

3.1 METHODISCHES VORGEHEN

Es soll eine Beispielanwendung sowohl in containerisierter Form als auch Serverless entwickelt werden. Die Konzeption dieser beiden Anwendungen wird in den folgenden Abschnitten genauer beschrieben. Im Anschluss werden Performance-Tests beider Anwendungen durchgeführt und die Ergebnisse miteinander verglichen. Die Forschungsfrage wurde dazu in kleinere Teilfragen (Research-Questions RQ1 bis RQ5) aufgeteilt, die im Laufe der Analyse untersucht werden sollen und unterschiedliche Performance Aspekte betrachten. Die Fragen sind in Tabelle 3.1 aufgelistet.

RQ1	Wie viel Last kann eine einzige Container-Instanz tragen und wie vielen nebenläufigen Lambda-Funktionen entspricht dies?
RQ2	Wie unterscheidet sich die Performance bei normaler und schnell ansteigender Last?
RQ3	Welchen Einfluss hat die Nutzung größerer CPU / RAM Werte?
RQ4	Welchen Einfluss hat die Nutzung mehrerer Container-Instanzen auf die Performance?
RQ5	Welchen Einfluss haben die unterschiedlichen Use-Cases auf die Performance?

Tabelle 3.1: Research-Questions dieser Arbeit

Das methodische Vorgehen für den Aufbau der Test-Umgebung basiert dabei grundlegend auf den von Papadopoulos et. al. vorgestellten methodologischen Prinzipien für reproduzierbare Performance-Evaluation im Cloud Computing Umfeld, die in Tabelle 3.2 beschrieben werden[30].

Das Prinzip P4 wird erfüllt durch die Nutzung eines öffentlichen GitHub Repositories unter <https://github.com/rolule/ba>, in dem sowohl die Skripte die zur Generierung der Tests verwendet wurden, als auch die Output-Artefakte der Tests abgelegt sind (siehe Anhang B).

Im Anschluss an die Performance-Tests wird auf die Kostenmodelle der beiden Technologien eingegangen.

P1	Wiederholte Durchführung (Repeated experiments): Mehrere Experimente mit der selben Konfiguration durchführen. In dieser Arbeit wird jeder Test mindestens zwei mal durchgeführt, um einmalige Aussetzer erkennen zu können.
P2	Konfigurations-Abdeckung (Workload and configuration coverage): Experimente in verschiedenen Konfigurationen der relevanten Parameter durchführen. Beispielsweise verschiedene Hardware-Konfigurationen aber auch verschiedene Load-Testing-Typen (z.B. Spike-Test).
P3	Experimenteller Aufbau (Experimental setup description): Beschreibung der Hardware- und Software (inklusive Version), und anderer Parameter die für das Experiment genutzt wurden und einen Einfluss auf dessen Ausgang haben können. Zusätzlich sollte die Beschreibung das genaue Ziel des Experiments beinhalten.
P4	Offener Zugang zu Artefakten (Open access artifact): Möglichst viele für die Experimente genutzten Konfigurationsdateien, Protokolle, etc. sollten versioniert und offen zugänglich sein.
P5	Probabilistische Ergebnisbeschreibung der gemessenen Performance (Probabilistic result description of measured performance): Angemessenes Beschreiben und Visualisieren der Ergebnisse. Verwendung von Quantilen (z.B. Median oder 95. Quantil) und der Standardabweichung.
P6	Statistische Auswertung: Statistische Tests anwenden, um die Validität der Studie zu untermauern. Dieses Prinzip wird aus mangelnder Expertise nicht genauer betrachtet, daher hier der Hinweis, dass die Ergebnisse dieser Arbeit eventuell nicht signifikant sein könnten.
P7	Einheiten (Measurement units): Für alle Messungen die zugehörigen Einheiten angeben.
P8	Kosten (Cost): Kostenmodell, Ressourcennutzung und abgerechnete Kosten angeben.

Tabelle 3.2: Prinzipien für reproduzierbare Performance-Evaluation im Cloud-Computing Umfeld nach Papadopoulos et. al.[30]

3.2 KONZEPTION DER TEST-ANWENDUNGEN

Als Testanwendung wird ein einfaches HTTP-REST-Backend am Beispiel einer Notiz-Applikation entwickelt. Der Service stellt folgende Routen bereit:

1. GET /notes: Das Auflisten aller verfügbarer Notizen
2. GET /notes/{id}: Das Auflisten eines spezifischen Notiz mit der angegebenen id
3. PUT /notes/{id}: Das Ändern einer spezifischen Notiz mit der angegebenen id
4. POST /: Das Erstellen einer Notiz

Um die Anwendung möglichst einfach zu gestalten und Unterschiede zwischen den beiden Technologien zu minimieren, wird auf die Verwendung einer Datenbank verzichtet. Stattdessen wird ein Delay von 50 Millisekunden für jeden Request eingebaut. Auch auf Authorisierungs-Mechanismen wird der Einfachheit halber verzichtet. Dies erlaubt es außerdem, von einer konstanten Benutzer-Rate auszugehen, da alle „eingeloggten“ Benutzer auch gleichzeitig aktiv sind[29]. Als Runtime wird sowohl bei der Serverless- als auch bei der containerisierten Anwendung Node.js 12 verwendet.

3.2.1 Serverless

Für die Entwicklung der Serverless-Anwendung wird das mit mehr als 38.000 Github-Stars populäre Serverless-Framework in der Version 2.21.0 verwendet[46]. Dabei handelt es sich um ein Open-Source Framework zur Entwicklung von Serverless-Anwendungen. Es übernimmt die Erstellung von Anwendungs-Stacks, Rechteverteilung und Konfiguration bei allen gängigen Public-Cloud Anbietern wie [AWS](#), Microsoft Azure oder [GCP](#). Da in dieser Arbeit [AWS](#) verwendet wird, erstellt Serverless automatisch einen Anwendungs-Stack mit [AWS](#) CloudFormation. Abbildung 3.1 zeigt das Setup der mit der Serverless-Architektur konzipierten Webanwendung.

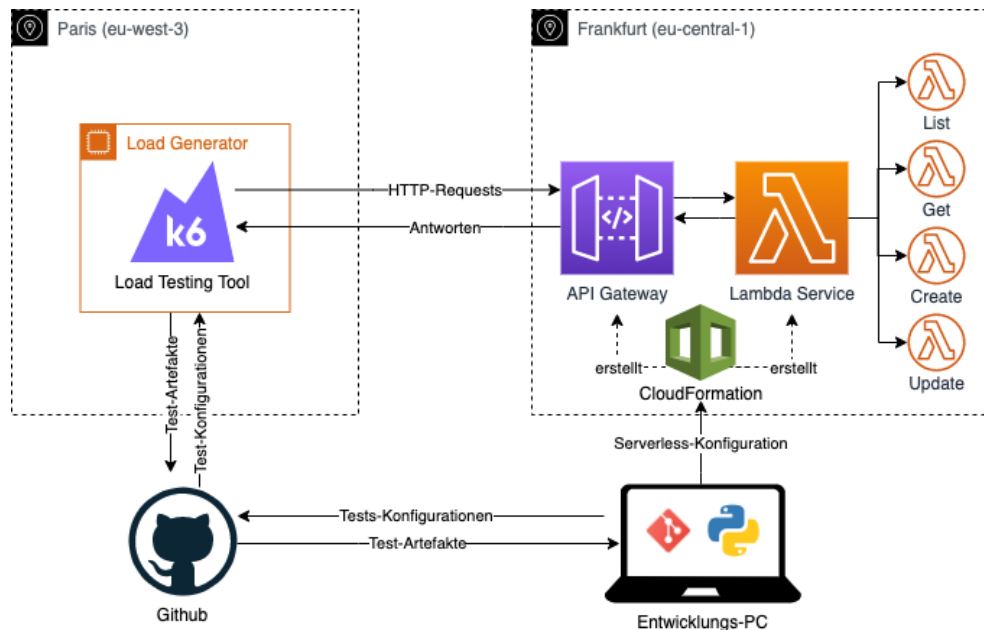


Abbildung 3.1: Testing-Architektur der Serverless Notiz-Anwendung

Der vom Serverless-Framework bzw. CloudFormation erstellte Anwendungs-Stack beinhaltet ein Amazon API-Gateway, das über die Routen der REST-Schnittstelle verfügt. Trifft eine HTTP-Anfrage des Load-Generators an eine der Routen ein, übernimmt das API-Gateway die Auslösung des passenden Events, wodurch der Lambda-Service die korrekte Lambda-Funktion startet und mit den im HTTP-Request übergebenen Para-

metern aufruft. Nach Beendigung der Funktion wird ihr Rückgabewert über das API-Gateway zurück an den Load-Generator gesendet. Jeder Start einer Lambda-Funktion erfordert das Herunterladen des Codes, der in der Funktion ausgeführt werden soll. Dazu wird der Code vom Serverless-Framework in einen [S3](#)-Bucket hochgeladen und bei Bedarf vom Lambda-Service angefordert (vgl. Abbildung 2.2).

Durch die Nutzung des Frameworks wird die Erstellung der Anwendung vereinfacht. Die Konfiguration des API-Gateways und des [S3](#)-Buckets werden automatisch erledigt. In Zukunft kann die Analyse durch Nutzung des Frameworks leicht auf [FaaS](#)-Angebote weiterer Cloud-Anbieter wie Microsoft Azure Functions oder Google Cloud Functions erweitert werden.

3.2.2 Container

Für die Entwicklung der containerisierten Anwendung wird das Express-Framework in der Version 4.17.1 verwendet, das mit über 51.000 Github-Stars ein weit verbreitetes Web-Frameworks für Node.js ist[44]. Die Anwendung wird unter Nutzung der Docker-Engine (Version 20.10.0) mittels einer Dockerfile in ein Container-Image gebaut. Dieses Image wird in der [AWS](#) Elastic Container Registry (ECR) gespeichert, um es von dort aus weiter zu verwenden. Die Ausführung der Anwendung erfolgt auf [ECS](#), einem von [AWS](#) bereitgestelltem Orchestrations-Tool. Es wird der Starttyp Fargate benutzt, welcher die Erstellung und Verwaltung eines Server-Clusters automatisch übernimmt und so die Konfiguration erleichtert. Des Weiteren wird ein Elastic Load Balancer ([ELB](#)) vom Typ Application verwendet, um die eingehenden Requests auf die verschiedenen Container zu verteilen.

Abbildung 3.2 zeigt das Setup der mit der Container-Architektur konzipierten Notiz-Anwendung. Bei der Konfiguration des Container-Clusters muss für Fargate eine Task-Definition erstellt werden. Diese definiert alle Tasks die gleichzeitig in einem Service laufen sollen. Ein Service konfiguriert die Ausführung mehrerer Container-Instanzen. In dieser Arbeit wird, außer bei den Tests mit mehreren Instanzen, für jeden Service nur ein einziger Task, also eine einzige Container-Instanz jedem Service zugeordnet. [ECS](#) erlaubt nur bestimmte Paare von CPU und Arbeitsspeicher-Einstellungen für einen Container. Für die Tests mit 128MB und 256MB Speicher wird das Einstellungs-Paar mit 512MB RAM und 0.25 vCPU festgelegt und die einzelnen Container mit entsprechenden harten Limits von 128MB und 256MB für den Arbeitsspeicher konfiguriert. Für die Tests mit 512MB wird die Einstellungs-Paar mit 1GB RAM und 0.5 vCPU verwendet, der ein hartes Limit von 512MB Speicher zugewiesen bekommt. Ein Virtual CPU ([vCPU](#)) beschreibt laut [AWS](#) hierbei einen Thread eines CPU-Kerns[10]. Die verwendeten Task-Definitionen sind in Anhang A aufgelistet. Sie werden mittels der [AWS](#) Konsole erstellt und konfiguriert.

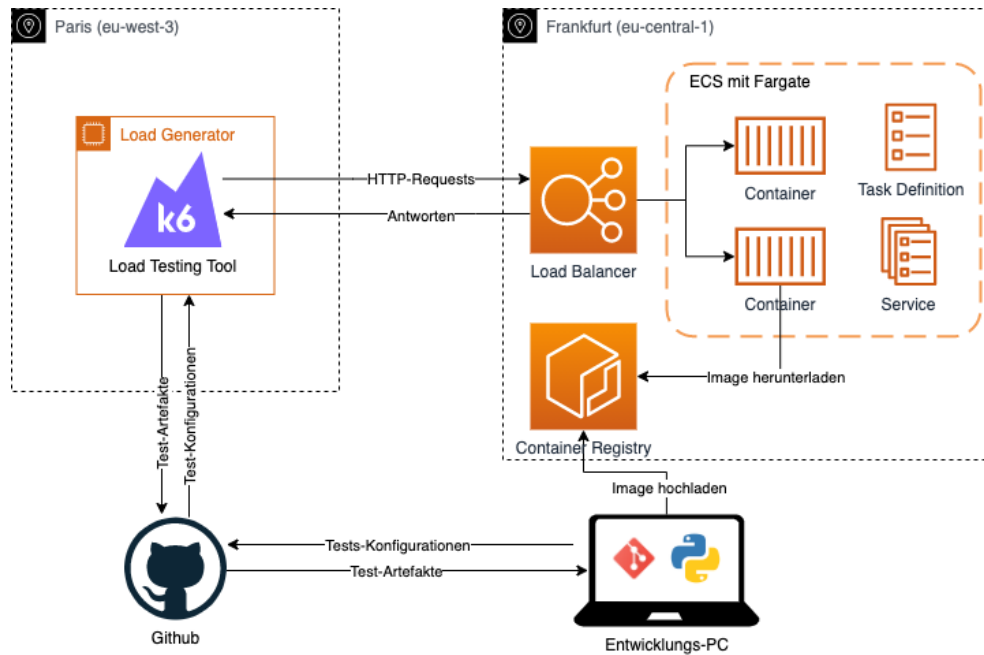


Abbildung 3.2: Testing-Architektur der Container Notiz-Anwendung

3.3 KONZEPTION DER TESTS

3.3.1 Testing-Tool

Um die Performance der eben vorgestellten Anwendungen zu testen, wird das Open-Source Performance-Testing-Tool k6 [25] verwendet. Durch eine optimierte CPU-Auslastung ermöglicht es, ohne die Notwendigkeit einer verteilten Ausführung, etliche virtuelle Benutzer zum Testen einer Anwendung zu kreieren[33]. Die Benutzer arbeiten parallel die in [Sektion 3.3.4](#) vorgestellten Use-Cases ab, bei denen verschiedene Anfragen an die REST-APIs der Services geschickt werden. Die Abläufe der Anwendungsfälle werden dafür in der von k6 unterstützten Skripting-Sprache JavaScript geschrieben und dem Tool bei der Ausführung übergeben.

Als Load-Generator wird eine in der [AWS](#) Region Paris (eu-west-3) stationierte [EC2](#) Instanz des Typs t3.xlarge verwendet. Nach eigenen Angaben benötigt k6 für jeden virtuellen Benutzer und für einen kleinen Test eine Speicherauslastung zwischen 1 und 5 Megabyte[33]. Da die verwendete t3.xlarge Instanz 16 Gigabyte RAM beinhaltet, sollten theoretisch mehr als 3.200 und maximal 16.000 gleichzeitige virtuelle Benutzer von der Testumgebung unterstützt werden. Die Instanz weist eine Netzwerkleistung von bis zu 5 Gbit/s auf, welche bei jeder Testausführung mittels des Tools *iftop* überwacht wird, um sicherzugehen, dass das Netzwerk nicht zum Flaschenhals wird. Auch die CPU-Auslastung wird mittels *htop* überprüft. Es wurde eine andere Region für den Load-Generator gewählt, um keine Ergebnisverfälschungen durch eventuell optimierte Routing-Mechanismen innerhalb eines [AWS](#) Rechenzentrums zu verursachen.

Die Tests werden ausgeführt, indem vom Entwicklungscomputer eine SSH Verbindung zum Load-Generator aufgebaut wird. Dann wird der Test-Code über das GitHub-Repository heruntergeladen. Zur Ausführung der Tests und automatisch komprimierter Speicherung der Test-Artefakte im Repository wurde ein Node.js Skript geschrieben, welches unter dem Dateinamen „test.js“ zu finden ist. Das Skript liest von der Kommandozeile die Parameter des aktuellen Tests ein und startet dann k6 mit dem vom Benutzer ausgewählten Test-Szenario (siehe Anhang B).

3.3.2 Metriken

Das Testing-Tool k6 speichert für jeden Request an einen bestimmten API-Endpoint unter anderem Werte folgender Metriken:

1. Virtual Users (VUs): Die Anzahl der virtuellen Benutzer zum aktuellen Zeitpunkt.
2. Antwortzeit (Response Time): Die Zeit vom Abschieken eines Requests bis zum Erhalt der Antwort in Millisekunden (ms). Dabei wird Aufwand für eventuelle DNS Lookups nicht mit einberechnet[28].
3. HTTP-Statuscodes: Geben den Status eines HTTP-Requests an. Wichtig für die Betrachtung sind vor allem die Codes:
 - a) 200 (OK): Der Request war erfolgreich[1].
 - b) 500 (Internal Server Error): Der Server konnte den Request aufgrund eines nicht vorhergesehenen Problems nicht angemessen verarbeiten[2].
 - c) 502 (Bad Gateway): Der Server ist überlastet oder kann aus anderen Gründen nicht erreicht werden[14].
 - d) 503 (Service Unavailable): Der Server ist überlastet oder für Wartungsarbeiten abgeschaltet[3].
 - e) 504 (Gateway Timeout): Der als Gateway / Proxy agierende Server konnte innerhalb eines festgelegten Zeitintervalls keine Antwort liefern[15].
4. Request-Rate: Die durchschnittliche Anzahl der Requests pro Sekunde.

Für Analysen und Vergleiche mehrerer Test-Ausführungen untereinander, werden die Ergebnisse jedes k6-Tests als Comma Separated Values (CSV) Text-Datei exportiert. Im Anschluss können diese Dateien mit dem Data-Science Framework Pandas[45] für die Programmiersprache Python analysiert und verglichen werden. Der Source-Code der Analysen und die Ergebnis-Artefakte der Tests sind im Code-Repository dieser Arbeit zu finden (vgl. Anhang B).

Weitere Metriken wie die CPU-Auslastung eines Fargate-Clusters und die maximale Nebenläufigkeit, Funktions-Dauer und Coldstart-Dauer werden von der AWS CloudWatch Konsole abgelesen und können daher nicht als Artefakte gespeichert werden.

3.3.3 Test-Typen

Bei Performance-Tests können verschiedene Testing-Strategien eingesetzt werden. Die für diese Arbeit relevanten Strategien sind Load- und Stress-Tests:

1. Load Testing: Dient der Evaluierung der System-Performance in Bezug auf die Anzahl der gleichzeitigen Benutzer oder der Requests-Rate. Es wird der Normalbetrieb des Systems simuliert[41].
2. Stress Testing: Dient dazu, die Grenzen des Systems in den Punkten Verfügbarkeit und Stabilität auszutesten. Man geht über den normalen Betrieb hinaus und eventuell so hoch, dass das System der Last nicht mehr standhalten kann. Bei einem Stress-Test wird schrittweise die Last auf das System erhöht. So kann man zum Beispiel herausfinden, ob das System großen Anstürmen, wie z.B einem Sale-Event bei einem Online-Shop, Stand halten kann[42].

Es gibt noch weitere Test-Typen wie Smoke- oder Soak-Tests, die allerdings keine Relevanz für die Zielfragen dieser Arbeit haben und daher nicht betrachtet werden.

3.3.4 Getestete Use-Cases

Die folgenden Use-Cases werden bei der Analyse der Systems-under-Test (SUTs) evaluiert. In der Aufzählung ist immer zuerst das verwendete HTTP-Verb (GET, POST, PUT) und danach der korrespondierende Endpunkt der Applikation aufgelistet. Die nach dem Pfeil angegebenen Zeitintervalle entsprechen der sogenannten „Think Time“, also der Zeit die ein Benutzer normalerweise benötigt um sich für seine nächste Handlung zu entscheiden. Es wird, wie von Molyneaux empfohlen, mit einer ± 10 prozentigen Abweichung versucht, eine realistischere Szenariodurchführung zu erreichen[29]. Um möglichst realitätsnahe Ergebnisse zu erzielen, wird davon ausgegangen, dass der Benutzer etwa eine Sekunde benötigt, um sich einen Überblick über alle Notizen zu verschaffen oder einen Fehler in der Notiz zu erkennen und etwa drei Sekunden, um eine Notiz zu erstellen oder zu bearbeiten. Am Ende jedes Use-Cases wird zudem eine Wartezeit von etwa einer Sekunde eingefügt, um den darauf folgenden ersten Request der nächsten Iteration nicht sofort nach dem letzten Request der aktuellen Iteration durchzuführen.

- Use-Case A: Alle Notizen abrufen
Alle Notizen einmal abrufen.
1. GET /notes \rightarrow 1s
- Use-Case B: Notiz bearbeiten
Erst alle Notizen, dann eine einzelne Notiz abrufen und bearbeiten.
1. GET /notes \rightarrow 1s

2. GET /notes/1 \rightarrow 3s
3. PUT /notes/1 \rightarrow 1s

- Use-Case C: Notiz falsch erstellt

Der Benutzer ruft alle Notizen ab. Er entscheidet sich eine Notiz zu erstellen. Er bemerkt dass er einen Fehler gemacht hat und ruft die erstellte Notiz ab. Er bearbeitet die Notiz und speichert die Notiz erneut ab.

1. GET /notes \rightarrow 1s + 3s = 4s
2. POST /notes \rightarrow 1s
3. GET /notes/1 \rightarrow 3s
4. PUT /notes/1 \rightarrow 1s

3.3.5 Ablauf der Tests

Alle Tests werden in Abhängigkeit der Konfiguration der Systeme bezüglich des Arbeitsspeichers und der Anzahl der Container-Instanzen durchgeführt. Nach Molyneaux sollte für jede Konfiguration zunächst ein Pipe-Clean Test durchgeführt werden[29]. Dabei wird jedes System von nur einem einzigen Benutzer für ein bestimmtes Szenario angefragt. Dadurch lassen sich Basiswerte für die betrachteten Metriken ableiten, welche in späteren Tests zu Vergleichen hinzugezogen werden können.

Anschließend wird ein Stress-Test für die Konfiguration der Container-Anwendung durchgeführt. Dies ist notwendig, da Lambda ein von Natur aus automatisch skalierendes System ist, während ein Container alleine nicht skaliert werden kann. Um eine zu hohe Auslastung des Containers während der Last-Tests zu vermeiden, werden Stress-Tests durchgeführt, um das VU-Limit für die aktuelle Container-Konfiguration zu ermitteln. Im Anschluss wird mit diesem VU-Limit ein Last-Test gegen beide Systeme durchgeführt und die Ergebnisse beider SUTs miteinander verglichen. Der Last-Test wird mit einem langsamen und einem schnelleren Anstieg der Virtuellen Benutzer durchgeführt. Dadurch lässt sich das Verhalten der Systeme unter in kurzem Zeitraum zunehmender Last evaluieren. Die Last-Tests mit dem schnellen Anstieg der Benutzer werden im Folgenden als Spike-Tests bezeichnet, um den Unterschied der beiden Tests deutlicher zu machen. Nachdem die Tests der 128MB Konfiguration beendet wurden, werden daraufhin die 256MB und 512MB Konfigurationen untersucht. Anschließend folgen Tests mit mehreren Container-Instanzen. Zwischen jedem Lambda-Test wird eine Pause von mindestens 10 Minuten eingelegt, um dem Lambda-Service Zeit zu geben, warme Funktionen zu beenden. Jeder Test wird jeweils zweimal ausgeführt und die Metriken bei der zusammengefassten Betrachtung über die Gesamtmenge der Daten aggregiert. Bei der Anzahl der Requests wird die Summe gebildet, bei den durchschnittlichen Requests pro Sekunde das arithmetische Mittel.

ANALYSE

In diesem Kapitel werden die in Kapitel 3 vorgestellten Systeme analysiert.

4.1 128MB KONFIGURATIONEN

Zunächst werden alle Tests für die 128MB Konfigurationen ausgeführt. In der folgenden Sektionen wird dann die Speicher- und Prozessorgröße erhöht.

4.1.1 Pipe-Clean Tests

Zunächst wird die Performance der SUTs in der bestmöglichen Situation mit einem einzigen Nutzer ermittelt. Ziel dessen ist es, eine Vergleichsgrundlage für die darauf folgenden Tests zu schaffen. Die Tests wurden mit einer Dauer von zehn Minuten und einem aktiven Benutzer für jeden Use-Case durchgeführt.

	Use-Case A		Use-Case B		Use-Case C	
Metrik \ Konfiguration	Cont. 128MB	Lamb. 128MB	Cont. 128MB	Lamb. 128MB	Cont. 128MB	Lamb. 128MB
Requests	1699	1628	1044	1017	780	768
Durchschn. Requests / s	0.94	0.9	0.58	0.56	0.43	0.42
Durchschnittliche Antwortzeit (ms)	60.48	108.20	59.68	107.25	60.15	113.67
Minimale Antwortzeit (ms)	59.12	75.81	58.27	75.46	58.26	76.44
Maximale Antwortzeit (ms)	66.94	644.17	69.55	747.50	68.74	736.37
Median Antwortzeit (ms)	60.46	102.20	59.62	98.50	60.3	99.59
P(90) Quantil d. Antwortzeit (ms)	60.81	124.42	60.06	110.74	60.25	118.18
P(95) Quantil d. Antwortzeit (ms)	61.12	137.28	60.65	146.59	60.88	172.21
Standardabweichung d. Antwortzeit	0.42	28.89	0.56	55.86	0.66	72.70
Variationskoeffizient	0.01	0.27	0.01	0.52	0.01	0.64

Abbildung 4.1: Vergleich der Pipe-Clean Tests für 128MB

Abbildung 4.1 zeigt die Ergebnisse des Tests für die Funktionen mit 128MB CPU bzw. RAM. Die Zahlen sind in der Tabelle aufgrund des Testing-Tools in der Notation mit Punkt anstatt eines Kommas angegeben. Es ist zu erkennen, dass die Container-Anwendung in allen Fällen eine deutlich geringere Antwortzeit als die Lambda-Anwendung aufweist. Im Median ist der Container bei Use-Case A etwa 41,74ms schneller als die Lambda-Funktion. Die Lambda-Anwendung weist eine überaus große Varianz in ihren Antwortzeiten auf, was in ihrer Standardabweichung und einem Variationskoeffizient von bis zu 0,64 deutlich wird, während dieser bei der Container-Anwendung in allen drei Use-Cases bei nur 0,01 liegt. Die Antwortzeit der Lambda-Anwendung im Median liegt jedoch in allen drei Use-Cases bei ähnlichen Werten um ca. 100ms. Andererseits nimmt die Abweichung vom Mittelwert und das P(95) Quantil der Antwortzeit bei Use-Case B im Vergleich zu Use-Case A deutlich zu. Auch bei Use-Case C ist ein deutlicher Anstieg der Wer-

te zu beobachten. Vermutlich ist dies auf die größere Anzahl an angefragten Endpunkten zurückzuführen, da dadurch mehr Coldstarts durchgeführt werden mussten. Auch bei der Container-Anwendung ist keine Veränderung der Median-Antwortzeit zwischen den einzelnen Use-Cases festzustellen, da sie sich in allen Tests um einen Wert von ca. 60ms bewegt. Die Standardabweichung der Antwortzeit veränderte sich mit einer Erhöhung von 0,42 auf 0,66 zwischen Use-Case A und C nur leicht. Der Variationskoeffizient blieb aber in allen drei Use-Cases bei 0,01.

Anhand der Vergleichstabelle lassen sich auch Erkenntnisse über die einzelnen Use-Cases ableiten. Use-Case A setzt sowohl bei der Container- als auch bei der Lambda-Anwendung deutlich mehr Requests in der Sekunde ab als es bei den Use-Cases B und C der Fall ist. Dies ist auf die größeren Think-Times von drei Sekunden zurückzuführen, wie sie bei den Use-Cases mit Benutzer-Eingaben zum Einsatz kommt.

4.1.2 Stress-Tests

Um RQ1 beantworten zu können, muss zunächst die maximale Auslastung eines einzelnen Containers ermittelt werden. Dazu werden mehrere Stress-Tests durchgeführt. Ein Stress-Test dient dazu, die Grenzen des SUT herauszufinden. Da Lambda ein von Natur aus automatisch horizontal skalierendes System ist, werden die Tests zunächst für die Container-Anwendung durchgeführt und im Anschluss die Lambda-Anwendung mit dem gleichen Test-Ablauf getestet. Die Anzahl der VUs wird bei den Stress-Tests, wie von Molyneaux[29] empfohlen, immer stufenweise erhöht. Das bedeutet, dass nach jedem linearen Anstieg, auch Ramp-Up genannt, eine gleichlange Periode mit einer konstanten Anzahl an Benutzern folgt. Abbildung 4.2 verdeutlicht dies anhand eines Stress-Tests, bei dem bis zu 600 gleichzeitige Virtuelle Benutzer erstellt werden. In 60 Sekunden Zeitabschnitten werden nach und nach immer 60 Benutzer hinzugefügt. Daraufhin folgt eine weitere 60 Sekunden lange Periode, in der die Anzahl der VUs nicht weiter erhöht wird. Der gesamte Test hat demnach eine Dauer von 40 Minuten. Eine Cooldown-Phase, bei der die Anzahl der Benutzer nach Erreichen des Limits langsam zurückgefahren wird, wird hier nicht eingelegt, da das Verhalten der Systeme beim Herunterskalieren nicht untersucht werden soll.

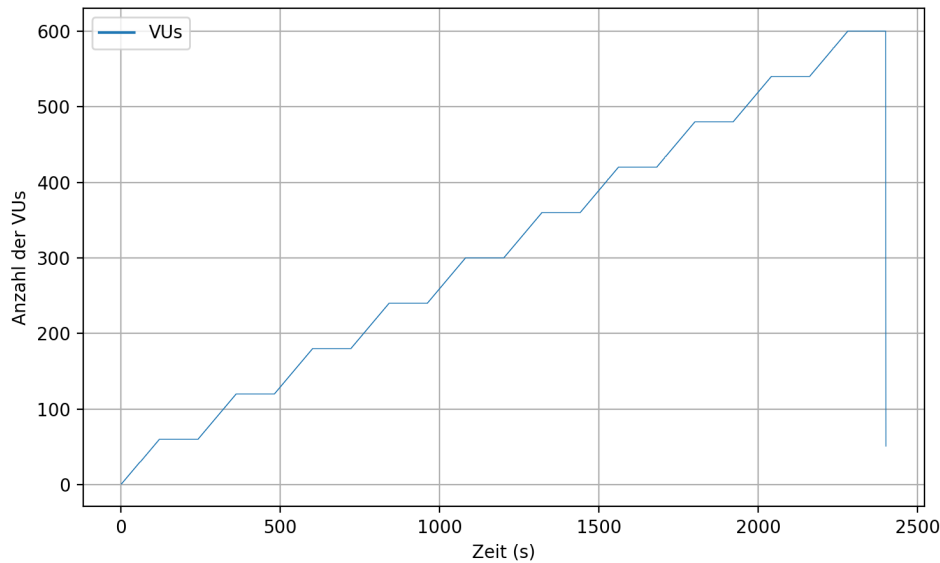


Abbildung 4.2: Beispiel-Anstieg der gleichzeitigen VUs für einen Stress-Test

4.1.2.1 Container

Für die containerisierte Anwendung in der 128MB Konfiguration wurden mehrere Stress-Tests durchgeführt, bis das Limit der gleichzeitigen Benutzer identifiziert werden konnte. Abbildung 4.3 zeigt die gemessenen Metriken für die einzelnen Use-Cases und Stress-Test Konfigurationen.

Metrik \ Max. VUs	Use-Case A			Use-Case B			Use-Case C		
	200	600	1000	200	600	1000	200	600	1000
Requests	463077	1424926	2272495	284808	876714	1460289	213316	656340	1094220
Durchschn. Requests / s	96.43	296.71	473.16	59.20	182.25	303.56	44.27	136.21	227.07
Durchschnittliche Antwortzeit (ms)	60.15	60.56	108.65	60.12	59.96	61.50	60.13	60.05	60.32
Minimale Antwortzeit (ms)	57.70	29.08*	9.34*	9.19*	8.34*	11.73*	8.59*	8.91*	9.10*
Maximale Antwortzeit (ms)	558.12	638.19	1013.35	276.58	304.38	1033.86	451.01	276.96	528.56
Median Antwortzeit (ms)	59.80	59.73	60.73	59.88	59.70	59.83	59.94	59.85	59.79
P(90) Quantil d. Antwortzeit (ms)	60.66	60.82	227.93	60.75	60.56	61.33	60.76	60.73	60.74
P(95) Quantil d. Antwortzeit (ms)	61.04	61.88	302.03	61.14	60.90	68.30	61.05	61.10	61.29
Standardabweichung d. Antwortzeit	5.28	6.86	84.34	2.42	3.12	10.12	4.65	2.26	4.90
Variationskoeffizient	0.09	0.11	0.78	0.04	0.05	0.16	0.08	0.04	0.08
Maximale CPU-Auslastung (%)	31.78	84.93	100	21.63	58.96	94.99	16.10	48.80	74.44
Fehler	0	5x502	77x502	1x502	1x502	8x502	2x502	4x502	3x502

Abbildung 4.3: Container 128MB Stress-Test Vergleich

Der erste Test wurde mit bis zu 200 Benutzern durchgeführt. Dabei konnte bei Use-Case A nicht einmal 32% CPU-Auslastung des Container-Clusters erreicht werden; Bei Use-Case B waren es sogar unter 22%. Der geringen Auslastung entsprechend, war kaum eine Veränderung der Antwortzeiten gegenüber den Pipe-Clean Tests festzustellen (vgl. mit Abbildung 4.1). Einige Metriken lagen sogar unter den Grundwerten. Lediglich die maximale Antwortzeit hat sich bei allen Use-Cases deutlich erhöht, bspw. bei Use-Case A von 60,46ms auf 558,12ms. Diese Erhöhungen sind allerdings nur vereinzelt der Fall, da immer noch 95% aller Anfragen innerhalb 62ms beantwortet wurden. Der Variationskoeffizient stieg bei allen Use-Cases leicht an.

Bei dem zweiten Stress-Test mit bis zu 600 VUs, änderte sich nicht viel im Vergleich zu dem Test mit 200 Benutzern. Es konnten für Use-Case A zwar bis zu 84,93 Prozent CPU-Auslastung erreicht werden und die maximale Antwortzeit stieg erneut an auf 638,19ms. Trotz dessen, konnten von dem einzelnen Container erneut 95% der Anfragen innerhalb von 62ms beantwortet werden. Bei Use-Case A fallen fünf Requests auf, die vom Server nicht beantwortet werden konnten und den HTTP 502 Fehlercode zurücksendeten. Dies entspricht jedoch nur einem äußerst geringen Anteil aller in diesem Stress-Test durchgeführten Anfragen. Bei Use-Case B trat nur ein Fehler auf, bei Use-Case C waren es vier.

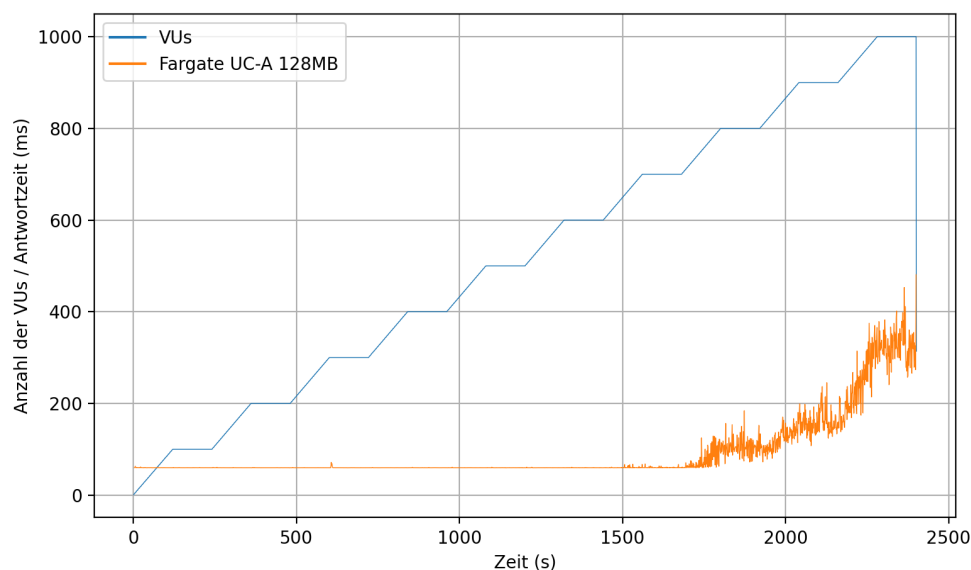


Abbildung 4.4: Container Use-Case A 128MB Stress-Test mit 1000 VUs Verlauf Vergleich

Der dritte Stress-Test für den 128MB Container wurde mit bis zu 1.000 VUs durchgeführt. Abbildung 4.4 zeigt den zeitlichen Verlauf eines der beiden Tests für Use-Case A. Es wird dabei die Anzahl der virtuellen Benutzer und die Antwortzeit im Median pro Sekunde dargestellt. Es wird deutlich, dass die Antwortzeit ab ca. 700 VUs deutlich ansteigt. Dies deckt sich der von AWS Cloudwatch gemessenen CPU-Auslastung von 100% die bei dieser Nutzerzahl gemessen wurde. In den darauf folgenden Perioden mit Anstieg der VUs steigt auch die Antwortzeit. In jeder Periode mit konstanten Nutzerzahlen können die Requests besser verarbeitet werden und die Kurve geht wieder leicht herunter. Während des Tests wurden trotz einer vollen CPU-Auslastung nur 77 HTTP 502 Fehlercodes vom Server ausgelöst, was einer Quote von 0,003% entspricht. Die durch Fehler hervorgerufenen Werte sind in allen Tabellen-Abbildungen mit einem Asterisk (*) gekennzeichnet. Die Antwortzeit konnte im Median weiterhin bei etwa 60ms gehalten werden; die durchschnittliche Antwortzeit stieg mit 108,65ms auf fast das doppelte des Grundwertes an. Bei Use-Case B lag die erreichte Prozessor-Auslastung bei 94,99%. Es wurden bis auf acht HTTP 502 Fehlercodes keine deutlichen

Abweichungen von den Pipe-Clean Tests ausgelöst. Der Variationskoeffizient stieg mit 0,16 leicht an, liegt aber immer noch deutlich unter dem Wert des Pipe-Clean Tests der 128MB Lambda-Funktion. Der Wertanstieg könnte durch die von den aufgetretenen Fehler verursachten geringen Antwortzeiten von bis zu 11,73ms ausgelöst worden sein. Um die maximale Auslastung des Containers für die anderen beiden Use-Cases zu erreichen, wurden noch weitere Stress-Tests mit bis zu 1.500 VUs durchgeführt. Bei Use-Case B konnte das Limit zwischen 1.000 und 1.100 Benutzern erreicht werden. Für Use-Case C liegt es bei ca. 1.200-1.300 Benutzern. Dadurch dass sich das Limit zwischen den beiden Ausführungen des gleichen Tests leicht unterscheiden kann, müssen hier Wertebereiche angegeben werden.

4.1.2.2 Lambda

Nachdem die maximale Performance des 128MB Containers ermittelt wurde, werden im Anschluss die gleichen Stress-Tests für die Lambda-Anwendung mit 128MB durchgeführt. Abbildung 4-5 zeigt die Ergebnisse der Tests. Allgemein liegt die Antwortzeit im Median immer noch deutlich über der des 128MB Containers; die Differenz ist allerdings im Vergleich zu den Pipe-Clean Tests gesunken, bspw. von 41,74ms auf 17,72ms bei den Tests mit bis zu 1.000 VUs für den ersten Use-Case. Für Use-Case A lässt sich ebenfalls erkennen, dass die Standardabweichung bei 200 und 600 VUs noch um die 18 beträgt, während sie bei 1.000 Usern auf ca. 41 ansteigt. Auch der Variationskoeffizient stieg von ca. 0.2 auf 0.47 an. Dies könnte allerdings an den extrem langen maximalen Antwortzeiten von bis zu 29.037ms liegen, die durch die aufgetretenen Fehler verursacht wurden. Die großen Fehler-Latenzen stehen im Gegensatz zu der Container Anwendung, bei der die maximale Antwortzeit nur knapp über einer Sekunde lag. Auffallend ist, dass Fehler mit HTTP Statuscode 500 etwa 10 Sekunden Antwortzeiten aufweisen, während Fehler mit HTTP Statuscode 504 die fast 30 Sekunden lange Latenz erzeugen. Die bei der Container-Anwendung erzeugten HTTP 502 Fehlercodes, führten zu kleineren Antwortzeiten von 12ms bis zu 260ms.

Metrik \ Max. VUs	Use-Case A			Use-Case B			Use-Case C		
	200	600	1000	200	600	1000	200	600	1000
Requests	450608	1389226	2318273	279795	862158	1437621	210312	648244	1080280
Durchschn. Requests / s	93.83	289.27	482.73	58.16	179.21	298.83	43.65	134.52	224.17
Durchschnittliche Antwortzeit (ms)	89.35	87.70	86.70	91.29	89.08	88.69	92.36	89.10	90.16
Minimale Antwortzeit (ms)	71.59	70.59	70.23	71.56	49.48*	42.05*	72.47	71.63	27.83*
Maximale Antwortzeit (ms)	1241.84	1178.54	29037.43*	689.92	6708.68	1271.16	1532.51	29018.74*	1760.97
Median Antwortzeit (ms)	82.13	80.26	78.45	86.22	81.53	81.15	87.30	81.28	83.75
P(90) Quantil d. Antwortzeit (ms)	108.57	103.94	102.07	111.79	108.50	108.20	112.45	106.94	110.32
P(95) Quantil d. Antwortzeit (ms)	118.99	116.93	116.30	121.79	118.98	118.55	122.80	118.52	120.79
Standardabweichung d. Antwortzeit	18.55	17.93	41.05	19.89	20.69	19.43	20.59	68.53	20.81
Variationskoeffizient	0.21	0.2	0.47	0.22	0.23	0.22	0.22	0.77	0.23
Maximale Funktions-Dauer (ms)	652.46	564.02	547.49	624.16	574.59	861.59	508.72	555.66	734.17
Maximale Cold-Start Dauer (ms)	279	282	300	338	294	274	260	315	525
Maximale Nebenläufigkeit	28	64	148	32	61	86	31	56	76
Fehler	0	0	6x500 3x504	0	1x500	1x500	0	3x500 3x504	1x500

Abbildung 4-5: Lambda 128MB Stress-Test Vergleich

Bei der Durchführung der Stress-Tests für die 128MB Lambda-Anwendung fällt außerdem auf, dass die Antwortzeit im Median bei steigenden Nutzer-

zahlen zu sinken scheint. Bei bis zu 200 VUs lag sie für Use-Case A noch bei 83.13ms, während sie bei bis zu 1.000 VUs auf 78.45ms sank. Die Werte für Use-Case B und C scheinen auch zwischen dem ersten und zweiten Stress-Tests zu sinken, zwischen 600 und 1.000 VUs gab es aber keinen Unterschied mehr in den Antwortzeiten. Die deutlich höhere Request-Rate von Use-Case A lässt darauf schließen, dass AWS die Funktionen schneller provisioniert, je mehr Anfragen sie erhalten. Dies würde auch den Verlauf aus Abbildung 4.6 erklären, bei dem jeweils eine Test-Ausführung der drei Use-Cases für den Stress-Test mit bis zu 1.000 Benutzern abgebildet ist.

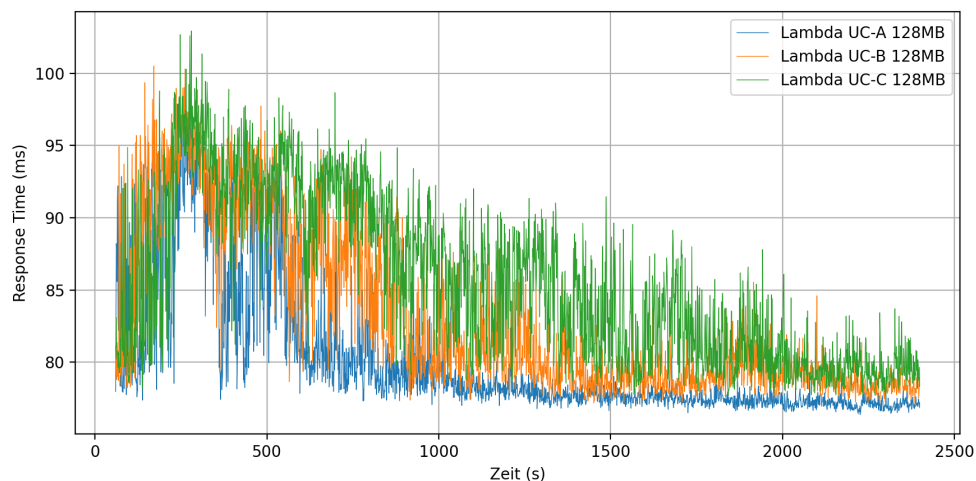


Abbildung 4.6: Lambda 128MB Stress-Test mit 1.000 VUs Verlauf Vergleich

Aus Gründen der Übersichtlichkeit wurde der Graph erst ab einer Minute dargestellt, da bei den Lambda-Funktionen vor allem in den ersten Sekunden der Tests verhältnismäßig hohe Response-Times auftreten. Es ist zu erkennen, dass die Antwortzeiten bei allen Use-Cases am Anfang ansteigen und nach ca. 300 Sekunden ihren Höhepunkt bei in etwa 95 bis 100ms erreichen. Danach flachen alle Kurven langsam ab, bis sie sich einem Wert von etwas unter 80ms annähern. Dabei scheint die Kurve von Use-Case A jedoch steiler als die von Use-Case B abzuflachen. Und auch dieser befindet sich schneller auf einem niedrigeren Niveau als der dritte Anwendungsfall.

Die höhere Request-Rate von Use-Case A macht sich ebenfalls in der Anzahl der nebenläufigen (concurrent) Lambda-Funktionen bemerkbar. Während bei 200 VUs für Use-Case B und C noch mehr Funktionen bereitgestellt wurden als für Use-Case A, zeigt sich für 600 und 1.000 VUs ein umgekehrtes Bild. Dort übertrifft der erste Use-Case den die anderen beiden deutlich in der Anzahl der nebenläufigen Funktionen.

4.1.3 Load-Tests

Für RQ2 soll die Performance unter normalen Nutzerzahlen mit der von schnell ansteigenden Nutzerzahlen verglichen werde. Dazu wird zunächst ein Load-Test für die in den vorangegangenen Stress-Tests ermittelten Nutzer-Grenzen der Container-Anwendung durchgeführt. Im Anschluss wird ein Spike-Test mit der gleichen Benutzerzahl durchgeführt. Ziel dessen ist es, die Performance der Anwendungen unter rasch ansteigender Last zu ermitteln.

In den bisherigen Stress-Tests wurden nur langsame Anstiege (Ramp-Ups) durchgeführt. Für den Container wurde eine maximale Belastbarkeit von ca. 700 Benutzern ermittelt. Deshalb wird nun für den Load-Test von einem normalem Load von 600 Benutzern ausgegangen, um den Container nicht zu stark zu belasten.

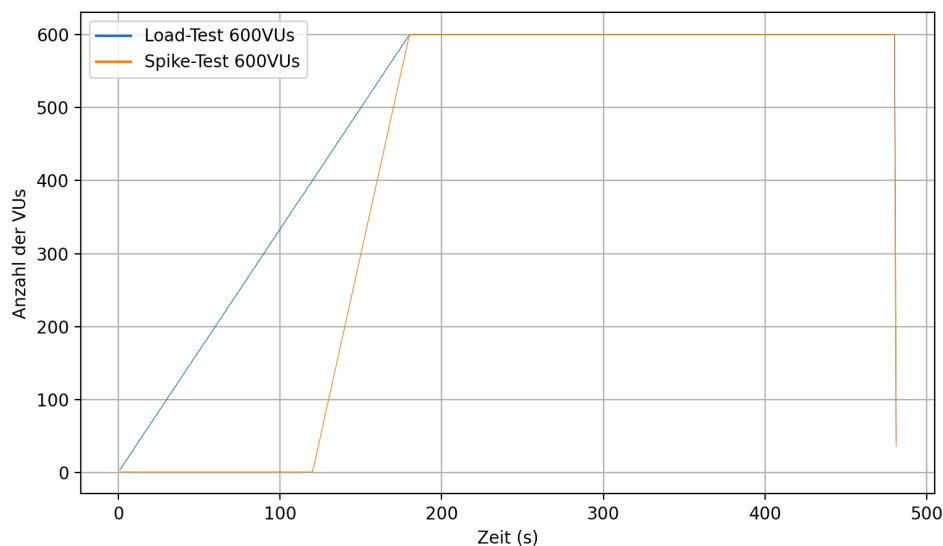


Abbildung 4.7: Load- und Spike-Test mit 600 VUs Vergleich

Abbildung 4.7 zeigt die unterschiedlichen Test-Verläufe der beiden Tests. Die Anzahl der Benutzer steigt bei dem Load-Test von Anfang an kontinuierlich an und erreicht innerhalb von drei Minuten die maximale Grenze. Für den Spike-Test erfolgt nach einer zwei-minütigen Warte-Phase, in der keine Benutzer auftreten, ein schneller Anstieg innerhalb von einer Minute auf die 600 VUs. Der Anstieg des Spike-Tests ist also drei mal steiler als der des Load-Tests. Nach Erreichen der Obergrenze wird bei beiden Tests die Anzahl der Benutzer für fünf Minuten konstant gehalten. Dies lässt eventuelle Nachwirkungen des Anfrage-Ansturms erkennen.

4.1.3.1 Container

Bei der Durchführung der Tests für den 128MB Container konnten keine Veränderungen der Performance für einen schnellen Anstieg der Benutzer

gegenüber dem langsamen Anstieg festgestellt werden. Für alle Use-Cases lag der Median der Antwortzeit knapp unter 60ms. Neunzig Prozent aller Requests wurden unter 62ms verarbeitet, das P(95) Quantil lag auch meist bei diesem Wert. Allein bei Use-Case A kam es zu erhöhten Werten von 65 bzw. 67ms bei den Load-Tests und fast 70ms bei den Spike-Tests. Dies ist vermutlich erneut auf die höhere Request-Rate des ersten Use-Case zurückzuführen.

Die Varianz veränderte sich stark zwischen den unterschiedlichen Test-Läufen. Der Wert von 0,01 aus den Pipe-Clean Tests wurde deutlich übertroffen. Use-Case C wies mit 0,05 und 7,39 den sowohl kleinsten als auch größten Wert der Load-Tests auf. Der letztere wurde durch einen starken Ausreißer verursacht, bei dem eine Zeit von fast 28 Sekunden verging, bevor die Antwort des Services registriert wurde. Bei den Spike-Tests lagen alle Werte der Use-Cases mit mehreren Endpunkten zwischen 0,04 und 0,08. Use-Case A wies Werte von 0,13 und 0,14 auf und lag damit deutlich vor den anderen getesteten Anwendungsfällen. Verursacht wurde dies vermutlich durch jeweils einen HTTP 502 Fehler, der die minimale Antwortzeit auf 16,28 bzw. 13,46ms schrumpfen ließ. Bei den anderen Use-Cases traten in den Spike-Tests keine Fehler auf.

4.1.3.2 *Lambda*

Bei den Lambda-Funktionen lassen sich für Use-Case A keine eindeutigen Unterschiede in der Antwortzeit für die beiden verschiedenen Anstiegs-Intervalle feststellen. In den beiden Load-Tests wurden 50 Prozent aller Requests in 80,24ms bzw. 84,49ms erledigt. Bei den Spike-Tests waren es 79,97ms bzw. 85ms. Im ersten Lauf der Load-Tests wurde eine maximale Nebenläufigkeit von 72 Funktionen festgestellt, im zweiten waren es 67. Bei den Spike-Tests waren es 67 und 68.

Abbildung 4.8 zeigt ein Vergleich jeweils eines Load- und Spike-Tests für Use-Case A. Für den Load-Tests (blaue Linie) ist zu erkennen, dass sich nach anfänglich hohen Antwortzeiten von ca. 95ms das Niveau bei 75 bis 80ms einpendelt. Ab 180 Sekunden (3 Minuten) ist bei beiden Tests die maximale User-Anzahl erreicht. Nach ca. 200 Sekunden überschreitet die Median-Antwortzeit für den Load-Test die 80ms Marke und erreicht ihren Höchststand nach ca. 350 Sekunden mit ca. 90ms. Danach sinkt die Median-Antwortzeit wieder trotz konstanter Nutzerzahlen. Für den Spike-Test scheint es sich ähnlich zu verhalten. Auch wenn nach 180 Sekunden die 600 VUs erreicht wurden, klettert die Antwortzeit erst mehr als eine Minute später über die 80ms Marke. Der Spike-Test erreicht ebenso wie der Load-Test nach ca. 350 Sekunden sein Maximum; dessen Antwortzeit liegt mit ca. 93ms nur leicht über der des Load-Tests.

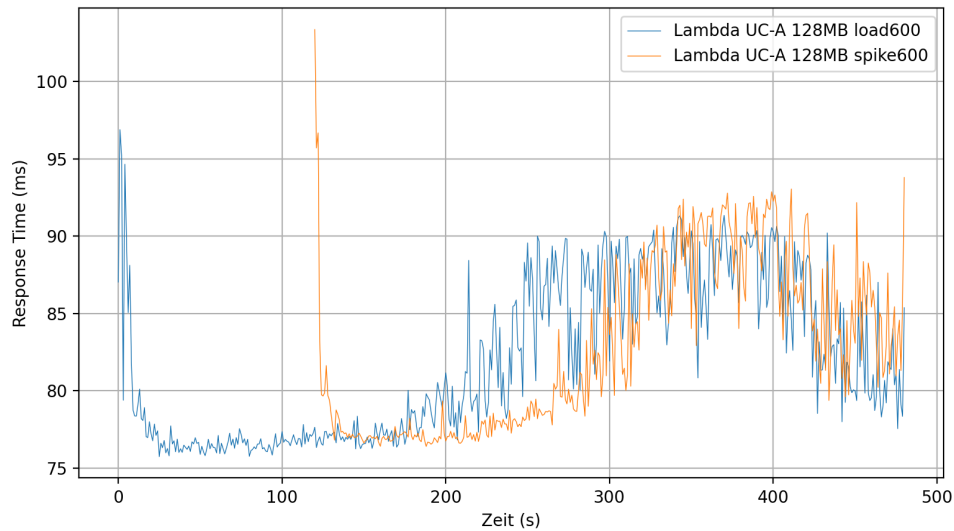


Abbildung 4.8: Lambda Use-Case A Load- und Spike-Test mit 600 VUs Verlauf Vergleich

Für Use-Case B (Abbildung 4.9) wurden zwei stark unterschiedliche Verläufe der Load-Tests festgestellt. Der erste Load-Test (blaue Linie) zeigt einen flacheren Verlauf als der zweite (orange Linie). Seine Kurve fällt nach Erreichen der höchsten Antwortzeit bei ca. 90ms schneller ab. Zudem liegt die maximale Antwortzeit des zweiten Load-Tests mit ca. 95ms in etwa 5ms über der des ersten. Auch der Spike-Test (grüne Linie) erreicht eine maximale (Median-)Antwortzeit von ca. 95ms, womit dessen Verlauf stärker dem zweiten Load Test ähnelt.

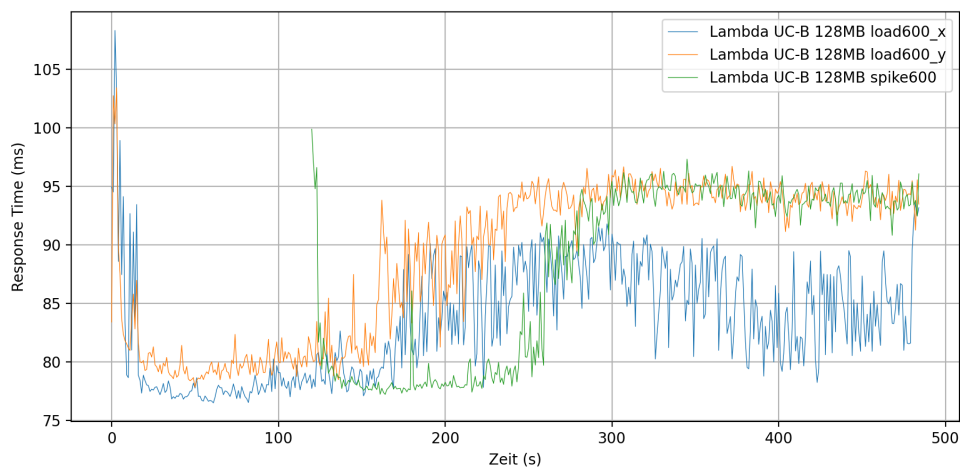


Abbildung 4.9: Lambda Use-Case B Load- und Spike-Test mit 600 VUs Verlauf Vergleich

Es wird deutlich, wie unterschiedlich die Verläufe eines identischen Tests bei der Lambda-Anwendung ausgeprägt sein können. Interessant ist außer-

dem, dass die Antwortzeit des ersten Load-Test eine größere Schwankung aufweist, als die des zweiten und des Spike-Tests, bei denen aufeinander folgende Werte stets relativ nah beieinander liegen. Aufgerufen wurden bei den Load-Tests jeweils 60 Funktionen, bei den Spike-Tests waren es 66 bzw. 62 Instanzen.

Auch bei Use-Case C konnte keinen großen Unterschiede zwischen den Load- und Spike-Tests mit 600 VUs festgestellt werden. In allen Tests wurden Median-Antwortzeiten von 86 bis 90ms gemessen. Hier wurden bei den Load-Tests 58 bzw. 56 und bei den Spike-Tests 58 bzw. 57 Funktionen parallel aufgerufen. Da die Antwortzeiten der Spike-Tests in allen Fällen trotz des dreimal steileren Anstiegs äußerst nah bei denen der Load-Tests liegen, ist daraus zu schließen, dass ein schneller Spike-Load nur eine geringfügige Veränderung der Antwortzeit mit sich bringt.

Es zeichnet sich wie auch schon bei den Stress-Tests ein Unterschied der Antwortzeiten zwischen den einzelnen Use-Cases ab. Exemplarisch zeigt Abbildung 4.10 den Verlauf der Antwortzeiten im Median für den Load-Test mit bis zu 600 VUs.

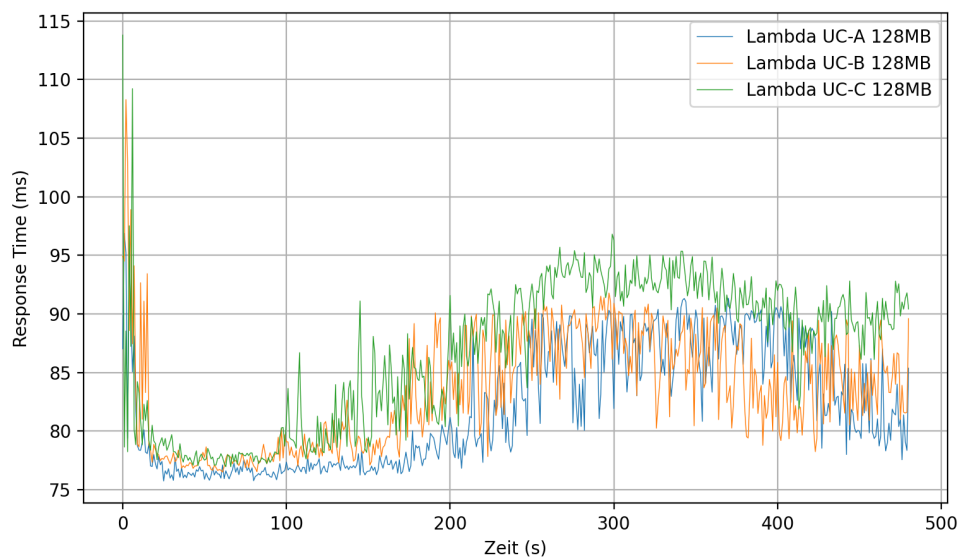


Abbildung 4.10: Lambda Load-Test mit 600 VUs Verlauf Vergleich

Es ist zu erkennen, dass alle Use-Cases einen generell ähnlichen Verlauf haben. Use-Case A mit nur einem Endpunkt weist jedoch überwiegend die geringste Antwortzeit auf. Die Linie von Use-Case B liegt nur leicht über der des ersten Use-Case und übertrifft diesen sogar teilweise gegen Ende des Vergleichs. Use-Case C mit den meisten Endpunkten weist auch die höchsten Antwortzeiten auf und erreicht mit über 95ms ein in etwa 5ms höheres Maximum als die anderen beiden Use-Cases.

4.2 ANDERE KONFIGURATIONEN (RQ3)

Als nächstes sollen Container und Lambda-Funktionen mit einer CPU- und Speicher-Größe von 256 und 512 Megabyte untersucht werden. Ziel dessen ist es, Unterschiede zu den in der vorangegangenen Sektion getesteten Konfigurationen zu erkennen.

4.2.1 Pipe-Clean Tests

Für die Pipe-Clean Tests der Container konnte keine Veränderung zur 128MB Variante festgestellt werden. Die Antwortzeit bewegte sich im Median erneut um die 60ms und auch der Variationskoeffizient zeigte nur eine äußerst geringfügige Veränderung. Bei allen Konfigurationen und Use-Cases wurden 90 Prozent aller Anfragen innerhalb von 62ms beantwortet. Größere CPU- und Arbeitsspeicher-Werte hatten hier also keine Verbesserung der Performance zur Folge. Abbildung 4.11 zeigt die aggregierten Metrik-Werte der Pipe-Clean Tests für die unterschiedlich großen Konfigurationen der Lambda-Funktionen.

	Use-Case A			Use-Case B			Use-Case C		
Metrik \ Konfiguration	128MB	256MB	512MB	128MB	256MB	512MB	128MB	256MB	512MB
Requests	1087	1089	1087	678	681	678	512	512	512
Durchschn. Requests / s	0.9	0.91	0.91	0.56	0.57	0.56	0.42	0.42	0.42
Durchschn. Antwortzeit (ms)	107.02	102.79	105.04	107.11	103.89	107.16	114.08	111.44	110.23
Minimale Antwortzeit (ms)	75.81	75.97	75.88	75.46	76.49	77.63	76.44	75.33	76.26
Maximale Antwortzeit (ms)	644.17	562.57	541.09	747.50	721.28	745.98	711.80	671.56	1106.45
Median Antwortzeit (ms)	100.91	99.88	101.05	98.27	97.75	99.59	99.61	100.20	99.32
P(90) Quantil d. Antwortzeit (ms)	123.43	113.19	115.71	110.38	106.79	111.98	118.87	117.63	113.15
P(95) Quantil d. Antwortzeit (ms)	136.81	118.32	128.19	139.76	116.28	133.33	185.12	132.71	124.36
Antwortzeit Standardabweichung	27.90	24.28	28.45	58.96	49.33	52.22	72.45	65.98	83.87
Variationskoeffizient	0.26	0.24	0.27	0.55	0.47	0.49	0.64	0.59	0.76
Maximale Funktions-Dauer (ms)	153.57	81.35	63.61	220.75	98.13	58.72	290.8	114.09	72.70
Maximale Cold-Start Dauer (ms)	245	216	248	269	248	245	245	246	266
Maximale Nebenläufigkeit	1	1	1	3	3	3	4	4	4
Fehler	0	0	0	0	0	0	0	0	0

Abbildung 4.11: Vergleich der Pipe-Clean Tests für Lambda

Es wird deutlich, dass auch hier fast keine Unterschiede zwischen den drei Ausführungen bestehen. Die mediane Antwortzeit liegt für alle Use-Cases in etwa bei 100ms. Die P(90) und P(95) Quantile variieren leicht zwischen den Tests der verschiedenen Konfigurationen. Tendenziell sind sie bei den größeren Konfigurationen niedriger als bei den kleineren, es lässt sich jedoch kein eindeutiges Muster erkennen. Es können außerdem keine Unterschiede in der Coldstart-Dauer oder der Varianz festgestellt werden. Allerdings ist zu erkennen, dass die maximale Funktionsdauer bei den größeren Konfigurationen deutlich geringer wird und sich dem minimalen Wert von 50ms annähert. Während der Wert für Use-Case A und die 128MB Funktion noch bei 153,57ms liegt, sinkt er für die 256MB Funktion auf 81,35ms herab. Für die 512MB Variante liegt der Wert nur noch knapp über 60ms. Ähnliches ist für die anderen beiden Use-Cases festzustellen.

4.2.2 Stress- und Load-Tests

Zunächst soll der 256MB Container untersucht werden. Trotz der doppelt so großen Werte für Arbeitsspeicher und vCPU konnte hier bei den Stress-Tests keine Verbesserung der maximalen Benutzer für die Container-Instanz erreicht werden. Genau wie der 128MB Container, begann die Antwortzeit des 256MB Containers für Use-Case A bei ca. 700 VUs stark anzusteigen. Für Use-Case B kam es ab ca. 900-1.100 VUs zum Anstieg der Antwortzeiten. Für Use-Case C wurde ein Limit von ca. 1.100-1.300 VUs erkannt. Damit ist kein großer Unterschied zu der 128MB Variante erkennbar.

Der Verlauf der Stress-Tests für der 128MB Serverless-Anwendung hatte gezeigt, dass eine größere Anzahl an Endpunkten zu höheren Antwortzeiten führen kann (vgl. Abbildung 4.6). Wie Abbildung 4.12 für den Stress-Test mit bis zu 1.000 VUs zeigt, ist dies auch bei den 256MB Lambda-Funktionen der Fall. Allerdings scheint der Größenunterschied zwischen den Use-Cases im Vergleich zu der kleineren Konfiguration zurückgegangen zu sein. Auch bei dieser Abbildung werden die ersten 60 Sekunden nicht angezeigt, da die initial sehr hohen Antwortzeiten die Darstellung der Verläufe stark verzerren würde.

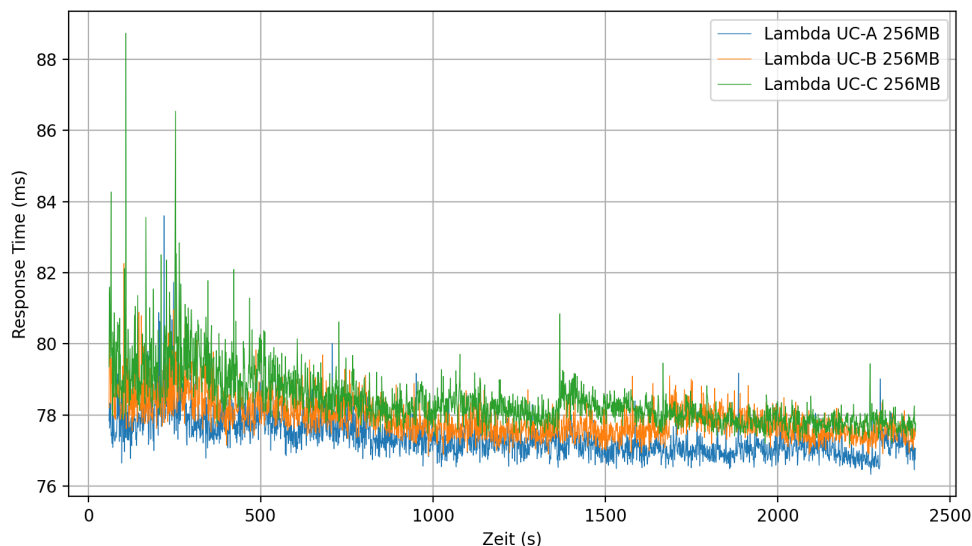


Abbildung 4.12: Lambda 256MB Stress-Test mit 1.000 VUs Verlauf Vergleich

Bei allen drei Use-Cases lagen die Antwortzeiten dennoch nahe um einen Median-Wert von ca. 78ms und es wurden 95 Prozent aller Anfragen in etwa 100ms verarbeitet. Bei der 128MB Variante reichte das P(95) Quantil noch von 114ms bis zu mehr als 122ms. Der Variationskoeffizient lag in allen Fällen bei 0,16 bis 0,18 und ist damit deutlich geringer als bei der kleineren Konfiguration, bei der er meistens 0,22 betrug, allerdings in einem Fall auch einen Wert von 0,64 erreichte. Damit ist eine deutliche Verbesserung der Performance und der Varianz in den Stress-Tests für die 256MB Variante festzustellen.

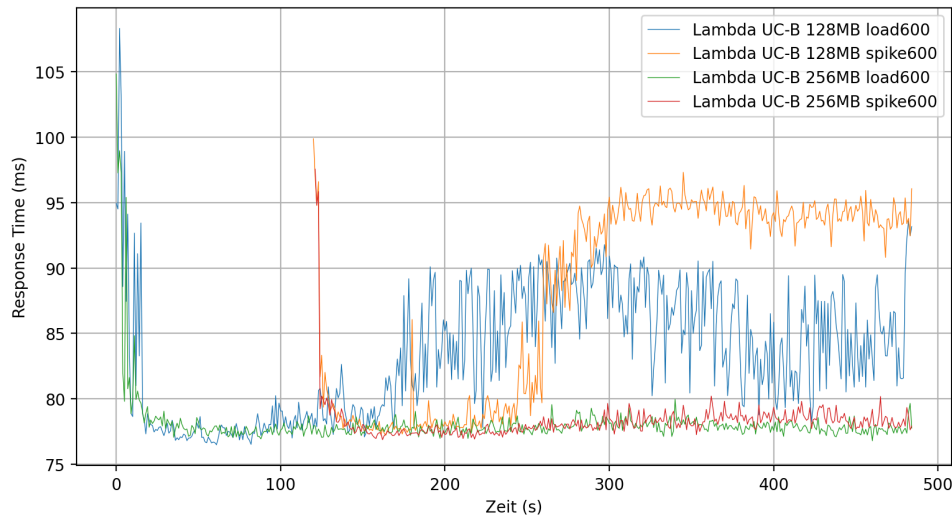


Abbildung 4.13: Lambda 128MB und 256MB Load- und Spike-Tests mit 600 VUs
Verlauf Vergleich

Im Anschluss wurden die Last- bzw. Spike Tests durchgeführt, um diese mit der 128MB Variante zu vergleichen. Als **VU**-Limit wurde erneut 600 Benutzer festgelegt. Dies ermöglicht einen einfachen Vergleich der beiden Varianten. Auch hier konnten für die Container-Anwendung keine Unterschiede zu der kleineren Konfiguration festgestellt werden.

Für die Lambda-Funktionen der Use-Cases A und B sind bei der 256MB Variante im Gegensatz zu der 128MB Konfiguration keine wesentliche Unterschiede zwischen den Load- und Spike Tests zu erkennen. Wie in Abbildung 4.13 für Use-Case B zu erkennen ist, sind die größeren Funktionen nahezu gar nicht von der Skalierung betroffen. Die Antwortzeit liegt, bis auf den anfänglichen Anstieg, beständig zwischen ca. 75 und 80ms. Im Vergleich zu der kleineren Variante, verkleinerte sich der Variationskoeffizient von 0,22 auf 0,17. Wie schon bei den Pipe-Clean Tests zu sehen war, sinkt auch hier die maximale Funktionsdauer von über 600ms bei 128MB auf unter 300ms ab. Bei den Antwortzeiten gab es ebenso eine geringfügige Verbesserung. Bei Coldstart-Dauer konnte dagegen keine Veränderung festgestellt werden. Allein bei Use-Case C gibt es Unterschiede in den Antwortzeiten. Während der Wert des Medians sich bei den Load-Tests nur etwas über 78ms befindet, liegt er bei den beiden Durchläufen der Spike-Tests bei 88,46 und 84,26ms. Vermutlich ist dies auf vermehrte Coldstarts durch die größere Anzahl an Endpunkten zurückzuführen. Bei den Load-Test-Durchläufen für Use-Case A wurden 69 bzw. 61 nebenläufige Funktionen registriert; bei den Spike-Tests waren es jeweils 62. Bei Use-Case B waren es 60 bzw. 58 und 57 bzw. 58. Die Tests für Use-Case C ergaben 53 bzw. 51 und 58 bzw. 55 parallele Funktionen. Damit liegt die Nebenläufigkeit für alle Use-Cases bei ähnlichen Werten im Vergleich zu den 128MB Funktionen.

Als nächstes sollen Container und Lambda-Funktionen mit einer CPU- und Speicher-Größe von 512 Megabyte untersucht werden. Ein Stress-Test

des Containers mit bis zu 1.500 Benutzern ergab für Use-Case A, dass die Antwortzeit des Containers ab einer VU-Anzahl von ca. 1.400 Benutzern deutlich zunimmt. Mit Eintreten der 1.400 Benutzer erreichte das Container-Cluster laut CloudWatch ebenso eine CPU-Auslastung von 100 Prozent. Dies bedeutet eine Verdoppelung des VU-Limits zu den kleineren Varianten. Bei einem zweiten Test konnte für Use-Case A allerdings keine Veränderung der Antwortzeit festgestellt werden. Darum wurden weitere Stress-Tests mit bis zu 2.000 Benutzern getestet und ein VU-Limit von ca. 1.600 identifiziert. Für Use-Case B musste ein weiterer Stress-Test ausgeführt werden, da bei 2.000 Benutzern nur eine CPU-Auslastung von 88% erreicht werden konnte. Dieser wurde mit bis zu 2.500 Benutzern durchgeführt. Bei ca. 2.200-2.400 VUs wurde die maximale Auslastung des Containers erreicht. Ein weiterer Stress-Test mit bis zu 3.000 VUs ergab für Use-Case C ein Limit von ca. 2.800-3.000 Benutzern. Wie schon in früheren Tests konnte keine Veränderung der Container Antwortzeiten erkannt werden, bevor eine annähernde Vollaustung des Prozessors erreicht wurde.

Im Anschluss an die Stress-Tests des Containers wurden diese mit den gleichen Konfigurationen gegen die Lambda-Anwendung durchgeführt. Dabei wurden einige Veränderungen zu den Stress-Tests der kleineren Varianten festgestellt. Beispielsweise schwankte die Median-Antwortzeit der Use-Cases B und C bei bis zu 2.000 Benutzern zwischen 81 und 88ms und fiel damit deutlich höher aus, als bei den Stress-Tests mit bis zu 1.000 Benutzern der 256MB Variante, bei der diese durchgehend zwischen 77 und 79ms gelegen hatte. Außerdem wurden bis zu 5ms Unterschiede in den Verläufen der gleichen Tests registriert. Der Abweichungskoeffizient erhöhte sich außerdem wieder auf Werte zwischen 0,19 und 0,28. Allein Use-Case A blieb sowohl bei der Median-Antwortzeit bei 77 bis 78ms und bei einem Variationskoeffizienten von 0,16 bzw. 0,17.

Die Load- und Spike-Tests wurden für die 512MB Konfiguration mit 1.200 VUs durchgeführt. Bei dem Fargate-Container konnte wie schon zuvor keine Unterschiede zwischen den beiden Tests festgestellt werden. Bis auf einige Ausreißer wurde wie schon bei den kleineren Konfigurationen für alle Use-Cases und für beide Anstiege eine Antwortzeit im Median von ca. 60ms festgestellt. Der Variationskoeffizient bewegt sich in den Testausführungen meist zwischen 0,03 und 0,05 und liegt damit nur geringfügig über den Werten der Pipe-Clean Tests. Bis auf eine höhere Anzahl der verarbeitbaren Anfragen konnten also keine Performance-Verbesserungen der 512MB Konfiguration festgestellt werden.

Die Load-Tests der 512MB Lambda-Funktionen bestätigten erneut, dass die Anzahl der Endpunkte einen Einfluss auf die Antwortzeiten hat. Es gab hier allerdings bei den Use-Cases A und B wie schon bei den Stress-Tests Unterschiede von einigen Millisekunden zwischen den Ausführungen des gleichen Anwendungsfalls. So war die beste Ausführung des Use-Case B besser als die schlechtere Ausführung des Use-Cases A. Die beste Variante des ersten Use-Cases war aber erneut besser als die performanteste Ausführ-

rung von Use-Case B. Der Use-Case mit den meisten Endpunkten schnitt wieder deutlich am unperformantesten ab.

Auch bei den Spike-Tests traten ähnliche Abweichungen wie bei den Load-Tests auf. Deshalb werden für die Analyse nur die jeweils besten Läufe betrachtet. Abbildung 4.14 zeigt den Verlauf der Load-Tests im Vergleich mit den Spike-Tests.

Auch hier zeichnet sich der Einfluss der Anzahl der Endpunkte ab. Der Median von Use-Case B liegt mit 78ms zwar nur knapp über den 77,02ms von Use-Case A. Deutlicher ist aber der Abstand von Use-Case C, dessen Antwortzeiten sich fast durchgängig bei in etwa 85ms befinden. Wie schon bei der 256MB Konfiguration ist kein Unterschied zwischen den Load-Test Ausführungen der Use-Cases A und B und den korrespondierenden Spike-Tests erkennbar. Für Use-Case C kann in der 512MB Konfiguration das gleiche beobachtet werden. Es zeigt sich also trotz der doppelt so großen Benutzerzahl im Vergleich zu den kleineren Konfigurationen eine leichte Verbesserung des Skalierungsverhaltens.

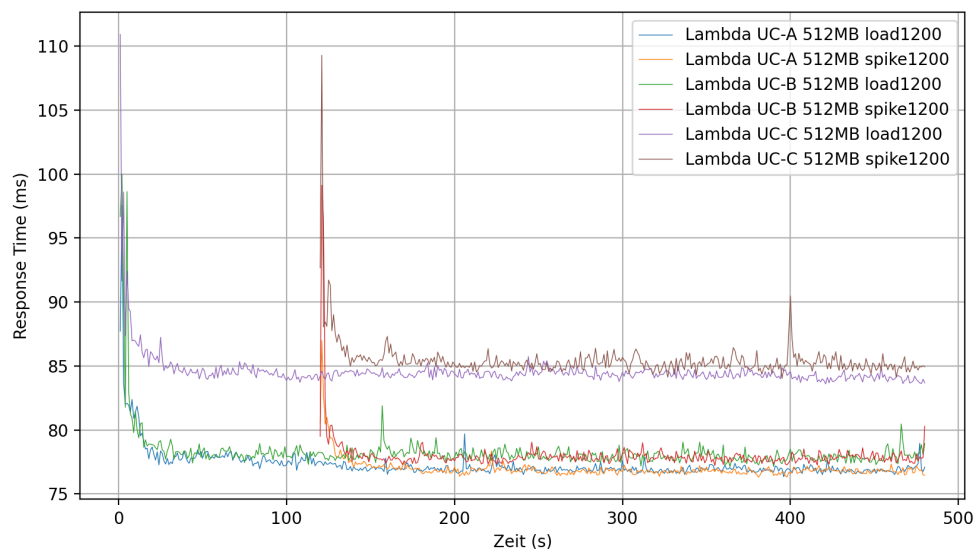


Abbildung 4.14: Lambda 512MB Load- und Spike-Tests mit 1.200 VUs Verlauf Vergleich

4.3 MEHRERE CONTAINER (RQ4)

Für die Untersuchung inwiefern sich die Anzahl der aktiven Container auf die Performance auswirkt, wurde ein Test mit der Fargate 256MB Task-Definition durchgeführt. Dazu wurden zwei Container-Instanzen gestartet und erneut Stress-Tests unterzogen. Bei Use-Case A wurde das Limit der Virtuellen Benutzer bei ca. 1.500 erkannt. Use-Case B wies eine Belastungsgrenze zwischen 2.100 und 2.300 VUs auf. Beim dritten Anwendungsfall wurden in etwa 2.800 bis 2.900 VUs registriert bevor die Performance degradierte. Damit liegen diese Werte sehr ähnlich an denen des 512MB Contai-

ners (siehe vorherige Sektion), bei dem Werte von 1.400, 2.200 bis 2.400 und 2.800 bis 3.000 VUs festgestellt wurden.

Im Anschluss wurden deshalb für die zwei 256MB Instanzen erneut Load-Tests mit bis zu 1.200 Benutzern durchgeführt, um eventuelle Performance Unterschiede zu der großen Konfiguration zu erkennen. Abbildung 4.15 zeigt die Ergebnisse der Tests im Vergleich mit denen des 512MB Containers.

	Use-Case A		Use-Case B		Use-Case C	
Metrik \ Konfiguration	2x256MB	512MB	2x256MB	512MB	2x256MB	512MB
Requests	883036	882055	545532	545577	409904	409892
Durchschn. Requests / s	917.58	916.52	561.94	562.04	418.77	418.76
Durchschnittliche Antwortzeit (ms)	61.19	62.20	60.21	59.94	60.04	59.85
Minimale Antwortzeit (ms)	26.45*	9.19*	8.74*	9.10*	57.69	9.40*
Maximale Antwortzeit (ms)	274.73	2643.78	278.12	278.52	403.73	275.71
Median Antwortzeit (ms)	59.60	59.83	59.63	59.73	59.68	59.67
P(90) Quantil d. Antwortzeit (ms)	60.89	62.34	60.55	60.68	60.56	60.54
P(95) Quantil d. Antwortzeit (ms)	68.45	65.79	61.21	61.16	60.98	60.91
Standardabweichung d. Antwortzeit	7.69	54.33	4.31	2.47	4.37	2.38
Variationskoeffizient	0.13	0.87	0.07	0.04	0.07	0.04
Maximale CPU-Auslastung (%)	88.11	80.28	64.49	52.03	50.57	40.60
Fehler	1x502	3x502	2x502	1x502	0	1x502

Abbildung 4.15: Vergleich der Load-Tests für 2x256MB und 512MB mit 1.200 VUs

Es wird deutlich, dass keine eindeutigen Unterschiede in der Performance zweier 256MB Instanzen zu einer einzelnen 512MB Instanz sichtbar werden. Beide Konfigurationen weisen sowohl ähnliche Requests pro Sekunde, als auch nahezu identische Antwortzeiten auf. Die Varianz ist bei allen Ausführungen äußerst niedrig und nur leicht über dem Niveau der Pipe-Clean Tests. Allein bei Use-Case A gab es größere Abweichungen vom Mittelwert, dieser wies aber auch in beiden Fällen die größte CPU-Auslastung auf. In der Auslastung des Clusters ist ebenfalls der größte Unterschied der beiden Konfigurationen zu erkennen. Bei allen Use-Cases lag diese bei den zwei Instanzen deutlich über der des einzelnen Containers. Bei Use-Case A waren es ca. acht Prozent, bei Use-Case B mehr als zwölf und bei Use-Case C ca. zehn Prozent mehr.

4.4 KOSTEN

In dieser Sektion sollen die Kostenmodelle der in dieser Arbeit getesteten Technologien untersucht und verglichen werden. Dabei wird sich erneut auf AWS beschränkt. Es kann daher bei anderen Public-Cloud Anbietern deutliche Unterschiede geben. Da es zwischen den einzelnen AWS Regionen ebenso Preisunterschiede bei der Nutzung des gleichen Services gibt, beziehen sich die folgenden Ausführungen ausschließlich auf die Region Frankfurt (eu-central-1).

4.4.1 Container

Da viele verschiedene Wege existieren Container-Anwendungen in der Public-Cloud zu betreiben, wird für die Kosten-Schätzung nur der Betrieb der Anwendung mit [ECS](#) und dem Starttyp Fargate betrachtet, wie er in dieser Arbeit verwendet wurde.

Bei Fargate wird die Abrechnung in die gewählte Arbeitsspeicher und [vCPU](#)-Größe unterteilt. Der Preis für ein Arbeitsspeicher Gigabyte pro Stunde beträgt aktuell 0,00511\$ und der für ein [vCPU](#) pro Stunde 0,04656\$[4]. Die Formel 4.1 zeigt die Berechnung der Kosten für ein Fargate-Cluster.

$$\text{Kosten} = \text{Tasks} * \text{Stunden} * \text{Tage} * (\text{SpeicherGB} * 0,00511\$ + \text{AnzahlvCPUs} * 0,04656\$) \quad (4.1)$$

Es wird davon ausgegangen, dass mindestens ein Task dauerhaft laufen muss, damit der Service erreichbar bleibt. Für die in dieser Arbeit verwendete 128MB und 256MB Konfigurationen (mit 0.25 [vCPU](#)) ergäben sich für einen laufenden Task pro Monat (30 Tage) und Dauerbetrieb (24 Stunden pro Tag) Kosten von:

$$\text{Kosten} = 1 * 24 * 30 * (0,5 * 0,00511\$ + 0,25 * 0,04656\$) = 10,22\$$$

Dieser Wert konnte mit den in dieser Arbeit durchgeführten Tests bestätigt werden, da für jeden Tag Kosten von 0,34\$ für die Nutzung des Clusters von [AWS](#) abgeführt wurden. Bei 30 Tagen Laufzeit ergeben sich damit tatsächlich Gesamtkosten von 10,2\$.

Aus der Formel lässt sich ableiten, dass sich für eine n-fache Anzahl an Tasks auch die n-fachen Kosten pro Monat ergeben. Variationen in der Anzahl der Tasks aufgrund von (Auto-) Skalierung sind unmöglich vorauszusagen und werden daher nicht betrachtet.

Zusätzlich zu den Fargate Kosten sind noch Gebühren für die Nutzung des [ELBs](#) fällig, dessen Nutzung bei der Verwendung mehrerer Instanzen unabdingbar ist. Für einen Application Load Balancer fallen pro angefangener Stunde Kosten von 0,027\$ an. Das ergibt für 30 Tage Kosten von 19,44\$. Zusätzlich fallen noch Gebühren für die Auslastung des Load-Balancers an; bspw. für die aktiven Verbindungen pro Minute[32]. Auch diese können nicht vorausgesagt werden und werden daher bei der Berechnung vernachlässigt.

Die folgende Übersicht zeigt die monatlichen Fixkosten einer Instanz für die unterschiedlichen in dieser Arbeit verwendeten Konfigurations-Größen inklusive Nutzung eines Application Load Balancers:

- 128MB = 0,5GB und 0,25 vCPU $\Rightarrow 10,22\$ + 19,44\$$ ([ELB](#)) = 29,66\$
- 256MB = 0,5GB und 0,25 vCPU $\Rightarrow 10,22\$ + 19,44\$$ ([ELB](#)) = 29,66\$
- 512MB = 1 GB und 0,5 vCPU $\Rightarrow 20,44\$ + 19,44\$$ ([ELB](#)) = 39,88\$

- $1024\text{MB} = 2\text{ GB}$ und $1\text{ vCPU} \Rightarrow 40,88\$ + 19,44\$ (\text{ELB}) = 60,32\$$

Kosten für Skalierung und die Nutzung des Load-Balancers müssen, wie bereits beschrieben, noch separat hinzugerechnet werden. Es wird deutlich, dass die Kosten für zwei 256MB Container der eines 512MB Containers entsprechen. Genau so ergeben zwei 512MB Container die gleichen Kosten eines einzelnen 1024MB Containers.

4.4.2 *Lambda*

Die Ausführung von Lambda-Funktionen wird nach der Ausführungszeit in Millisekunden abgerechnet. Dabei wird die Zeit für einen Cold-Start mit einberechnet. Auch die konfigurierte Arbeitsspeichergröße fließt mit in das Ergebnis ein. Derzeit berechnet [AWS](#) in der Region Frankfurt Kosten von 0,0000166667 US-Dollar pro Gigabyte-Sekunde (GB-Sekunde). Zusätzlich müssen pro einer Millionen Ausführungen einer Funktion im Monat noch einmal 0,20 US-Dollar Anforderungsgebühren gezahlt werden. [AWS](#) bietet ein kostenloses Nutzenkontingent von 400.000 Aufrufen pro Funktion im Monat an[23].

Für die Ausführung der Lambda-Funktionen wird ein Event-Auslöser für REST-Anfragen benötigt. Dazu wurde in dieser Arbeit der Amazon API-Gateway Service verwendet, welches eingehende Requests auf die vorgesehene Lambda-Funktion weiterleitet. Für die Verwendung des API-Gateways fallen zusätzliche Kosten an. Bei Nutzung des Serverless-Frameworks wird automatisch eine API des Typs REST erstellt. Es fallen für diesen API-Typ Kosten von 3,70 Euro pro einer Millionen Aufrufe an. Diese werden für jede API insgesamt berechnet und gelten damit nicht für jede einzelne aufgerufene Lambda-Funktion. Es gibt Vergünstigungen ab 333 Millionen Aufrufen im Monat. In dieser Arbeit werden keine Caching Mechanismen des API-Gateways verwendet. Werden diese in den Einstellungen der API aktiviert, fallen zusätzliche Kosten an. [AWS](#) bietet für das API-Gateway ein kostenloses Nutzenkontingent von einer Millionen Aufrufen im Monat[6]. Formel 4.2 zeigt die zusammengesetzte Berechnung der Gesamtkosten für eine Lambda-Funktion.

$$\text{Kosten} = \text{Aufrufe} * \left(\frac{\text{LaufzeitMs}}{1000\text{ms}} * \frac{\text{FunktionsMb}}{1024\text{Mb}} * 0,0000166667\$ + \frac{0,20\$ + 3,70\$}{1.000.000} \right) \quad (4.2)$$

In den Tests der vorherigen Kapitel lag die Antwortzeit der Lambda-Funktionen im Median oft um einen Wert von ca. 80ms. Unter Nutzung der Kostenformel 4.2 ergibt sich für eine Millionen Aufrufe einer einzigen Lambda-Funktion mit einer Größe von 128MB und dieser Laufzeit, ohne Einberechnung der kostenlosen Nutzenkontingente, ein Preis von:

$$\text{Kosten} = 1.000.000 * \left(\frac{80}{1000} * \frac{128}{1024} * 0,0000166667\$ + \frac{0,20\$+3,70\$}{1.000.000} \right) = 4,07\$$$

Die nachfolgende Tabelle zeigt die Kosten von unterschiedlichen Funktionsgrößen pro einer Millionen Aufrufe und für eine Laufzeit von 80ms.

Funktionsgröße	128MB	256MB	512MB	1024MB
Kosten	4,07 \$	4,23 \$	4,57 \$	5,23 \$

Da bei Lambda im Gegensatz zu [ECS](#) mit Fargate nur die tatsächlich aktive Zeit der Funktion berechnet wird, lässt sich grob abschätzen, wie viele Funktionsaufrufe zu dem gleichen Preis der Nutzung eines Fargate-Containers möglich sind (siehe vorherige Sektion). Es wird im folgenden davon ausgegangen, dass eine Funktion eine Laufzeit von 1ms bis zu 100ms hat. Die Umstellung der Kostenformel [4.2](#) ergibt Formel [4.3](#) zur Berechnung der Aufrufe bei Angabe der Kosten.

$$\text{Aufrufe} = \frac{\text{Kosten}}{\frac{\text{LaufzeitMs}}{1000ms} * \frac{\text{FunktionsMb}}{1024Mb} * 0,0000166667\$ + \frac{0,20\$+3,70\$}{1.000.000}} \quad (4.3)$$

Abbildung [4.16](#) zeigt das Ergebnis der Berechnungen. Die Werte der 128MB und 256MB Funktionen stimmen überein, da der Preis für die Fargate Konfiguration identisch ist.

Container \ Lambda	128MB	256MB	512MB	1024MB
128MB = 29,66\$	7.219.471 – 7.601.067	6.871.041 – 7.597.011	6.266.194 – 7.588.912	5.328.140 – 7.572.765
256MB = 29,66\$	7.219.471 – 7.601.067	6.871.041 – 7.597.011	6.266.194 – 7.588.912	5.328.140 – 7.572.765
512MB = 39,88\$	9.707.098 – 10.220.181	9.238.608 – 10.214.727	8.425.349 – 10.203.837	7.164.067 – 10.182.127
1024MB = 60,32\$	14.682.351 – 15.458.408	13.973.742 – 15.450.160	12.743.657 – 15.433.688	10.835.921 – 15.400.850

Abbildung 4.16: Anzahl der Aufrufe von Lambda-Funktionen dem Monatspreis von Fargate entsprechend

Der jeweils linke Wert in den Tabellenzellen gibt dabei die minimale Anzahl der Aufrufe an, wenn jeder Funktionsaufruf bei 100ms liegt. Der rechte Wert ist der für eine Funktions-Dauer von 1ms und beschreibt damit den maximal möglichen Wert, da Lambda immer mindestens eine Millisekunde abrechnet[24]. Es wird deutlich, dass schon für die Nutzung eines 128MB Containers im Monat zwischen sieben und fast acht Millionen Aufrufe einer Lambda-Funktion möglich sind. Das entspricht mehr als 240.000 Aufrufen täglich. Je größer der genutzte Container wird, desto größer wird auch die Anzahl der mit einer gleichwertigen Lambda-Funktion möglichen Aufrufe. Bei einem 512MB Container sind so schon zwischen acht und zehn Millionen Aufrufe einer 512MB Lambda-Funktion monatlich möglich. Bei einem 1024MB Container sind es sogar zwischen 10,8 und fünfzehn Millionen, was zwischen 360.000 und 500.000 Aufrufen täglich entspricht. Es muss darauf hingewiesen werden, dass die Kosten deutlich abhängig sind von der tatsächlichen Nutzung der Anwendung, da dadurch viele oder wenige Coldstarts ausgelöst werden können, die unter Umständen einige hundert Millisekunden dauern können.

DISKUSSION DER ERGEBNISSE

In diesem Kapitel sollen die Ergebnisse aus der vorangegangenen Analyse diskutiert werden. Im Allgemeinen ist in den Experimenten deutlich geworden, dass die Performance der Container-Anwendung die der Lambda-Funktionen übertrifft. Die Antwortzeiten lagen bei der Nutzung eines Containers im Median in fast allen Tests bei ca. 60ms. Nur wenn die CPU-Auslastung eines Containers sich einhundert Prozent annäherte, stiegen die Response-Times an. Dagegen lagen die Werte der Lambda-Funktionen bspw. bei den Pipe-Clean Tests um einen Wert von 100ms. Allerdings konnte die Performance von Lambda mit einer größerer Anzahl an Requests verbessert werden. Bei den Load- und Stress-Tests konnten so mediane Antwortzeiten von in etwa 76ms erreicht werden. Die kleinste Request-Dauer die in allen Lambda-Tests gemessen wurde betrug, Fehler ausgenommen, 69,99ms. Größere Ausreißer vom Mittelwert gab es bei beiden Technologien. Meist wurden diese durch Fehler verursacht. Während bei Fargate ausschließlich HTTP 502 (Bad Gateway) auftrat, sind bei Lambda sowohl HTTP 500 (Internal Server Error) und HTTP 504 (Gateway Timeout) Fehler vorzufinden. Ähnlich zu den verlängerten Antwortzeiten, sind die Fehler bei Fargate aber meistens nur bei hoher CPU-Auslastung ausgelöst worden. Es gab aber auch einzelne Ausfälle bei niedriger Auslastung. Zu einem kompletten Absturz des Servers oder HTTP 503 Statuscodes kam es in keinem Fall. Bei Lambda ist eine solche Auslastung durch die automatische Skalierung nicht möglich. Es ist daher sicher anzunehmen, dass die Fehler durch das API-Gateway verursacht wurden und nichts mit den eigentlichen Funktionen zu tun haben. Generell traten aber bei beiden Services wenige Fehler auf. Die der Serverless-Anwendung sind allerdings mit einer Request-Dauer von bis zu 30 Sekunden als verheerender einzustufen.

Auch in der Varianz der Antwortzeit zeigten sich große Unterschiede der Technologien. Die Fargate-Container wiesen schon in den Pipe-Clean Tests eine äußerst geringe Abweichung vom Mittelwert auf. Bei allen Konfigurationen und für alle Use-Cases betrug der Variationskoeffizient dort nur etwa 0,01. Bei den Lambda-Funktionen hatte sowohl die Funktionsgröße als auch der Use-Case einen Einfluss auf die Ergebnisse. In den Pipe-Clean Tests wies Use-Case A einen Variationskoeffizient von ca. 0,23 bis 0,29 auf, während er bei Use-Case B zwischen 0,45 bis 0,52 schwankte. Bei Use-Case C waren es mit 0,59 bis 0,64 noch größere Abweichungen. Während in den Pipe-Clean Tests keine Unterschiede in der Varianz der unterschiedlichen Funktions-Konfigurationen auffielen, sah es bei den Stress- und Load-Tests anders aus. Die 128MB Variante zeigte im Stress Test mit 600 VUs noch einen Abweichungskoeffizient von 0,22 (Use-Case A), bei 256MB schrumpfte der Wert für den gleichen Test auf 0,17. Dies scheint im Zusammenhang

mit der Funktions-Dauer zu stehen, deren maximaler Wert bei der kleineren Konfiguration fast bei 600ms lag, bei der größeren aber nur noch knapp 240ms betrug. Zwischen 256MB und 512MB Variante gab es allerdings keine Unterschiede mehr in der Abweichung vom Mittelwert; nur die maximale Funktionsdauer sank noch weiter.

5.1 BEANTWORTUNG DER FORSCHUNGSFRAGEN (RQ1 - RQ5)

Im Folgenden sollen die in Kapitel 3 vorgestellten Forschungsfragen anhand in der Analyse festgestellter Ergebnisse beantwortet werden.

Um die Anzahl der nebenläufigen Lambda-Funktionen zu ermitteln, die dem maximalen Load eines Containers entsprechen (RQ1), wurden mehrere Stress-Tests durchgeführt und dadurch das VU-Limit der Container-Konfigurationen ermittelt. Für die 128MB und 256MB Container-Instanzen konnte für Use-Case A ein maximaler Load von ca. 700 virtuellen Benutzern ermittelt werden. In den anschließenden Load-Tests mit bis zu 600 Benutzern wurde eine Nebenläufigkeit von maximal 72 Funktionen festgestellt. Bei der 512MB Konfiguration wurde eine Grenze von etwa 1.400 parallelen Benutzern ermittelt. Bei den Load-Tests mit 1.200 Benutzern konnten bis zu 137 Lambda-Funktionen registriert werden. Für die anderen Use-Cases lag die Request-Rate deutlich unter der von Use-Case A. Dadurch konnten z.B. bei der 512MB Funktion für Use-Case B mehr als 2.200 Benutzer, bei Use-Case C sogar mehr als 2.800 Benutzer registriert werden, bevor die Antwortzeiten anstiegen. Die geringere Request-Rate führte auch zu einer geringeren Nebenläufigkeit bei den Load-Tests der Use-Cases B und C. Daher sollte bei zukünftigen Experimenten von einer ähnlichen Request-Rate anstatt einer gleichen Anzahl der Benutzer ausgegangen werden. Darüber hinaus ist die Nebenläufigkeit von Lambda-Funktionen abhängig von der Laufzeit der Funktion, welche bei diesen Experimenten auf mindestens 50ms beschränkt war und darum in einer anderen Anwendung auch stark variieren könnte. In Zukunft könnten daher verschieden in ihrer Laufzeit beschränkte Lambda-Funktionen auf ihre Nebenläufigkeit untersucht werden. Forschungsfrage RQ1 lässt sich also nicht eindeutig beantworten, da sie sich als eindeutig anwendungsspezifisch erwiesen hat. Es lässt sich aber feststellen, dass eine einzige Container-Instanz durchaus mehreren Hunderten, evtl. sogar Tausenden Lambda-Funktionen entsprechen kann.

Für die Forschungsfrage RQ2 wurden die beiden Anwendungen Load-Tests mit unterschiedlich schnellem Anstieg der Benutzerzahlen unterzogen. Für die Performance des Containers konnte in den Experimenten keine Veränderung bei dem steilen Anstieg festgestellt werden. Bei den Lambda-Funktionen zeigte sich ein differenziertes Bild. Die Spike-Tests der 128MB Variante lösten einen verzögerten Anstieg der Antwortzeiten aus, welche bei Use-Case A noch ähnlich zu denen des Load-Tests; bei Use-Case B jedoch deutlich über denen des Load-Tests lagen. Bei den 256MB Funktionen konnte nur noch für Use-Case C ein geringer Unterschied festgestellt werden und bei den 512MB Funktionen war gar keine Diskrepanz mehr zwischen schnell-

lem und langsamen Anstieg erkennbar. Bei höheren Nutzerzahlen könnte dieser Effekt jedoch eventuell größer ausfallen. Es ist daher noch unklar, wann genau es zu Unterschieden zwischen schnellem und langsamen Anstiegen bei Lambda-Funktionen kommt.

Die Forschungsfrage [RQ3](#) befasste sich mit der Nutzung größerer RAM und CPU-Werten für sowohl die Container-Anwendung als auch die Lambda-Funktionen. Bei einem Wechsel von 128MB auf 256MB konnte bei der Container-Anwendung keine Veränderung der maximalen Benutzerzahlen festgestellt werden. Beide Container konnten z.B. für Use-Case A etwa 700 Benutzer bedienen, bevor die Antwortzeiten anstiegen. Vermutlich hängt dies damit zusammen, dass beide die gleichen Task-Gruppe von 512MB RAM und 0,25 vCPU zugewiesen bekamen. Dies war nötig, da [AWS](#) Fargate keine geringere Task-Gruppierung mit bspw. 128MB RAM und 0,125 vCPU zur Verfügung stellt. Um dieses Problem auszugleichen wurden beiden Containern harte Ressourcen-Limits für CPU und RAM gesetzt. Es wird vermutet, dass die Limits ignoriert wurden und deshalb beide Zugang zu 0,25 vCPU hatten. Im Gegensatz dazu, konnte beim Wechsel von 256MB auf 512MB eine Veränderung erkannt werden. Während der 256MB Container bei Use-Case A noch ein Limit von ca. 700 VUs hatte, konnten beim 512MB Container (mit 0,5 vCPU) in etwa 1.400 VUs, also in etwa doppelt so viele Benutzer vor Steigen der Antwortzeit bedient werden. Eine Verbesserung in den Antwortzeiten war allerdings nicht zu erkennen. Bei den Lambda-Funktionen zeigten sich einige Unterschiede in der Performance verschiedenen Konfigurationsgrößen. Bei den Pipe-Clean Tests lagen noch alle Metriken für alle Varianten bei ähnlichen Werten. Lediglich die maximale Funktionsdauer verbesserte sich eindeutig zwischen den Größen. Die Stress-Tests der 256MB Funktion zeigten dagegen eine deutliche Verbesserung der Antwortzeiten im Vergleich zu denen der 128MB Funktion. Auch die Abweichung vom Mittelwert wurde geringer. Außerdem wurde das Skalierungsverhalten bei den größeren Funktionen resilienter gegenüber schnellem Benutzeranstieg. Zwischen der 256MB und der 512MB Variante gab es aber bis auf die niedrigere maximale Funktionsdauer keine erkennbaren Verbesserungen mehr.

Für die Forschungsfrage [RQ4](#) wurde die Performance von mehreren Container-Instanzen zu der von einzelnen Containern in Beziehung gestellt. Bei Vergleich der Stress-Tests von zwei 256MB Instanzen und einer 512MB Instanz konnte gezeigt werden, dass beide Konfigurationen ungefähr die gleichen Limits an Benutzerzahlen aufwiesen. In den anschließenden Load-Tests gab es keine erkennbaren Unterschiede in den Antwortzeiten. Auffällig war, dass die Konfiguration mit zwei Containern eine höhere CPU-Auslastung aufwies als der einzelne Container. Dies könnte sehr wohl auf einen Unterschied in den Benutzerzahlen hindeuten, da die Volllastung des Clusters eventuell früher erreicht werden könnte. In Zukunft könnte dieser Effekt also noch detaillierter erforscht werden.

Untersuchungsgegenstand der letzten Forschungsfrage [RQ5](#) war, die Auswirkungen der Anzahl von Endpunkten eines Anwendungsfalls auf die Performance der Anwendungen zu untersuchen. Bei der Container-Anwendung

konnte in keinem Fall eine Veränderung der Antwortzeiten zwischen den Use-Cases festgestellt werden. Bei der Lambda-Funktion trat in den Pipe-Clean Tests kein Unterschied zwischen den Use-Cases auf, was darauf zurückzuführen ist, dass nur jeweils ein Coldstart pro Endpunkt benötigt wurde. Anders sah es bei den Stress- und Load-Tests aus. Dort hatte die Anzahl der Endpunkte durchaus einen Einfluss auf die Entwicklung der Antwortzeit. Bei den Stress-Tests der 128MB Funktionen sanken die Antwortzeiten bei mehreren Endpunkten langsamer auf das untere Niveau ab, als die eines Endpunktes. Bei den größeren Funktionen trat dieser Effekt nicht auf. Allerdings erreichen die Use-Cases mit mehreren Endpunkten in fast allen Load- und Stress-Tests eine allgemein höhere Antwortzeit. Es traten aber auch Schwankungen auf, bei denen bspw. der Verlauf von Use-Case B unter dem von Use-Case A lag. Tendenziell lässt aber der Vergleich der jeweils besten Verläufe vermuten, dass die Anzahl der Endpunkte einen Einfluss auf die Performance der Lambda-Anwendung hat. Allerdings konnten in den Tests dieser Arbeit nur geringfügige Differenzen in den Antwortzeiten festgestellt werden.

5.2 SKALIERUNG UND OPTIMIERUNG

Da Container auf jedem Computersystem mit Unterstützung einer Container-Runtime betrieben lauffähig sind, bieten sich vielfältige Wege an eine containerisierte Anwendung zu betreiben und zu skalieren. Deshalb kann in dieser Arbeit nicht auf alle eingegangen werden. Meist wird ein Container-Orchestrations-Tool verwendet, um das Verteilen der Container auf Computer-Clustern zu automatisieren. Bei Kubernetes gibt es den Horizontal Pod Autoscaler (HPA), der die automatische Skalierung von Pods in Abhängigkeit der CPU-Auslastung ermöglicht[18]. Auch bei Nutzung des AWS Elastic Kubernetes Service (EKS) ist eine automatische Skalierung mittels HPA möglich[17]. Des Weiteren ist es eine Möglichkeit, einen Vertical Pod Autoscaler (VPA) zu verwenden, um automatisch die Ressourcen einzelner Container zu skalieren[36]. Es kann auch ein Cluster-Autoscaler verwendet werden, um zusätzlich zu der Anzahl der Container auch die Anzahl der Kubernetes-Knoten zu skalieren[12]. Auf die Performance-Tests all dieser Konfigurationen kann in dieser Arbeit unmöglich eingegangen werden, da es zum einen zu viele Möglichkeiten der Skalierung und zum anderen auch zu viele Wege zur Optimierung gibt. Beispielsweise lässt sich die Schwelle der CPU-Auslastung einstellen, ab der die Anzahl der Container hoch oder runter skaliert werden soll. Liegt die Schwelle zu niedrig, skaliert das System evtl. zu früh und die Kosten steigen. Liegt die Schwelle zu hoch, skaliert das System evtl. zu spät und die Performance sinkt.

Um die Einstellung eines Clusters nicht vornehmen zu müssen, wurde in dieser Arbeit ECS mit dem Fargate-Starttyp verwendet. Fargate verwaltet die Knoten des Clusters automatisch. Der ordinäre Weg der Cluster-Erstellung mit ECS wäre, mehrere Virtuelle Maschinen zu starten, zu konfigurieren und jede dem Cluster zuzuweisen. Durch die Nutzung von Fargate kann die Kon-

figuration von Container-Clustern also erheblich vereinfacht werden. [ECS](#) und Fargate bieten mit Auto-Scaling eine weitere Möglichkeit der automatischen Skalierung an. Diese funktioniert zusammen mit [AWS](#) CloudWatch Alarmen. Dazu lässt sich eine von zwei verschiedenen Policies einstellen.

Bei einer Step Scaling Policy lassen sich, ähnlich wie bei Kubernetes [HPA](#), Grenzen für die Skalierung auf Basis von CloudWatch Metriken, also bspw. CPU- oder Arbeitsspeicher-Auslastung, festlegen. CloudWatch überprüft dann in bestimmten Intervallen die Metriken und schlägt Alarm, wenn eine Metrik über oder unter dem spezifizierten Schwellwert liegt. Fargate reagiert auf den Alarm und skaliert die Anzahl der Container innerhalb des Clusters hoch oder herunter[9].

Bei einer Target Tracking Policy, lässt sich für Metriken ein Ziel festlegen, das eingehalten werden soll. Beispielsweise könnte man ein Ziel von 75% CPU-Auslastung festlegen. Mithilfe der CloudWatch Alarme wird dann versucht, dieses Ziel möglichst einzuhalten[9].

Problematisch ist allerdings die Größe des Intervalls, in dem die Metriken überprüft werden können. Standardmäßig kann dies nur alle fünf Minuten erfolgen. Träfe die Container-Anwendung eine plötzliche Spitzenlast, würde es fünf Minuten dauern, bis reagiert und die Kapazität erhöht werden könnte. Da es sich bei der CPU-Auslastung um eine Standard-Metrik handelt, lässt sich dieses Intervall auf bis zu einer Minute herab senken[8]. Dies ist allerdings, verglichen mit den Skalierungsmöglichkeiten von [AWS](#) Lambda, noch immer langsam für Anwendungen die schnell auf hohe Lasten reagieren müssen. Eine weitere Senkung des Intervalls auf bis zu zehn Sekunden ist nur mit den sogenannten hochauflösenden Custom-Metriken möglich. Dafür fallen aber wiederum die dreifachen Kosten der Nutzung einer Standardmetrik an[31].

Wichtig für eine schnelle Skalierung kann auch die Größe des Containers-Abbilds sein. Da bei jeder neu hinzugefügten Container-Instanz das Container-Image von der Registry heruntergeladen werden muss[9], ist es notwendig, die Image-Größe möglichst zu minimieren. Zum Einsatz kommende Techniken sind hierbei beispielsweise die Nutzung von Alpine-Images als Basis-Image, einer besonders kleinen Linux-Distribution, Multi-Stage Builds, mit denen unnötige Dateien entfernt werden können, oder Layer-Merging, bei dem die Anzahl der Image-Layer minimiert wird[19].

Lambda ist dagegen ein von Natur aus horizontal skalierendes System. Daher müssen vom Entwickler theoretisch keine Einstellungen vorgenommen werden, um eine hochverfügbare Anwendung zu erschaffen. Um Cold-Starts möglichst zu vermeiden, lässt sich die gewollte Nebenläufigkeit allerdings auch schon vor einem Benutzer-Ansturm definieren. Diese Einstellungsmöglichkeit wird als Provisioned Concurrency bezeichnet. Lambda startet in diesem Fall bereits Funktionen vor, die dann bei Bedarf keinen Kaltstart mehr erfordern, sondern direkt einen Warmstart durchführen können. Kombinieren lässt sich dies mit Autoscaling, um die provisionierte Kapazität an steigende Anfragezahlen anzupassen[7].

Auch die Größe des Arbeitsspeicher einer Lambda-Funktion kann einen Einfluss auf das Skalierungsverhalten nehmen. Da eine Funktion mit größerem Speicher auch mehr CPU zugewiesen bekommt, können Anfragen, bei besonders CPU-intensiven Aufgaben, schneller verarbeitet werden und es werden eventuell weniger nebenläufige Funktionen und damit weniger Cold-Starts benötigt. Es gibt Tools wie „AWS Lambda Power Tuning“[11], die es für solche Funktionen ermöglichen, die beste Konfiguration zu finden.

Es gibt darüber hinaus vielfältige Wege, die Coldstart Zeit einer Lambda-Funktion zu verringern. Ähnlich zum einem Container-Image, hat auch die Größe einer Lambda-Funktion Einfluss auf die Startzeit, denn bei jedem Coldstart muss der Funktions-Code aus S3 in den neuen Lambda-Container geladen werden. Bei einigen Programmiersprachen wie Java oder .NET macht ebenfalls die Konfiguration des Lambda-Arbeitsspeichers einen großen Unterschied bei der Schnelligkeit eines Coldstarts[26]. Wie in den Tests dieser Arbeit bestätigt wurde, ist dies bei Node.js allerdings nicht der Fall.

5.3 KOSTEN

In Sektion 4.4 wurden die Kostenmodelle der beiden genutzten Services verglichen. Es wurden Formeln zur Abschätzung der Nutzungskosten eines Services für einen Monat vorgestellt. Der große Unterschied zwischen beiden Technologien liegt in der Abrechnung der tatsächlich verwendeten Rechenzeit. Während bei Lambda nur die Laufzeit einer Funktion im Millisekunden-Bereich abgerechnet wird, zahlt man pro Container einen Preis für seine gesamte Laufzeit - auch wenn er keine Anfragen bearbeitet. Für einen Container bezahlt man also gleich viel egal ob er mehr oder weniger ausgelastet ist. Vor allem zu Zeiten in denen die Auslastung gering ist, bspw. in der Nacht, ist die Nutzung eines Containers ein großer Nachteil. Der Vorteil von Lambda liegt darin, dass bei geringer Nutzung auch nur wenige Kosten anfallen. Im Gegensatz dazu, kann bei einer hohen Last die Nutzung eines Containers erheblich günstiger sein als eine Lambda-Anwendung, denn die Kosten pro einer Millionen Aufrufe des API-Gateways fallen schwer ins Gewicht. Bei welcher Technologie man weniger bezahlt, hängt also von der konkreten Auslastung des Services ab. Wie in Sektion 4.4.2 gezeigt wurde, lässt sich ungefähr bestimmen, wie viele monatliche Requests mit Lambda und API-Gateway möglich sind, bevor die Kosten die eines einzelnen Containers überschreiten. Andererseits hat ein Container eine Grenze, wie viele Requests er pro Sekunde verarbeiten kann, bevor ein weiterer Container hinzugezogen (skaliert) werden muss. Für jeden weiteren Container fallen auch weitere Kosten an, allerdings nur so lange er benötigt wird. Dafür müssen aber unter Umständen komplexe Skalierungs-Konfigurationen erstellt werden, was bei Lambda nicht notwendig ist. Hinzu kommen ebenso die ungewissen Nutzungskosten eines ELB. Welche Technologie das bessere Preis-Leistungs-Verhältnis aufweist, lässt sich also nur im Einzelfall bestimmen. Ist die Anzahl der Anfragen oder der Benutzer der Anwendung bekannt, kann es aber

für Organisationen hilfreich sein, die Kosten mit den vorgestellten Formeln abzuschätzen und mit Performance-Tests zu evaluieren, um die passendere Alternative auszuwählen.

Um Prinzip P8 zu erfüllen, werden im folgenden die tatsächlich für diese Arbeit angefallenen Kosten angegeben, um für eine erneute Durchführung der Experimente eine Einschätzung der nötigen Ressourcen zu geben. Insgesamt fielen für die Tests dieser Arbeit laut des AWS Cost-Explorers 215,40 US-Dollar für die Nutzung des API-Gateways an. Für Lambda waren es nur 19,91 Dollar. Die Gesamtkosten der Serverless-Anwendung berufen sich also auf ungefähr 235 Dollar. Im Gegensatz dazu wurden für die Container-Anwendung nur ca. 33 Dollar fällig, davon 12,94 Dollar für ECS mit Fargate und 20,26 Dollar für den ELB. Zusätzlich müssen noch Kosten in Höhe von 48,52 Dollar für die EC2-Instanzen, die als Load-Generator fungierten, hinzugerechnet werden. Es muss darauf hingewiesen werden, dass diese Kosten von der Betriebsdauer der EC2 Instanzen und des ECS-Clusters abhängig sind. Des Weiteren fallen noch Kosten von 45,65 Dollar für AWS Cloud-Watch an. Die Durchführung aller Tests in dieser Arbeit veranschlagte also insgesamt Kosten von ca. 317 Dollar. Dass das API-Gateway alleine fast 60 Prozent der Gesamtkosten veranschlagte und Fargate nur in etwa neun Prozent, zeigt, dass die Nutzung von Container eindeutig effizienter ist wenn beide Services die gleiche Last bearbeiten müssen und der Container der Last gewachsen ist.

5.4 IMPLIKATIONEN FÜR DIE PRAXIS

Durch die in dieser Arbeit durchgeführten Experimente wurde deutlich, dass die Performance eines Container-Backends auf AWS Fargate der einer Serverless-Anwendung mit AWS Lambda und API-Gateway in Bezug auf die Antwortzeiten überlegen ist. Allerdings liegt der Unterschied der beiden Technologien nur im Bereich von einigen Millisekunden. Wird eine zeitkritische oder in ihren Antwortzeiten konsistente Anwendung benötigt, bspw. eine Banking- oder Trading-Plattform, sollte dennoch eher die Verwendung einer Container-Architektur bevorzugt werden.

Lambda bietet ein stabiles Skalierungsverhalten, bei dem nach anfänglichen von Coldstarts verursachten Spitzenwerten beinahe konsistente Antwortzeiten erreicht werden und das auch bei schnellem Last-Anstieg gut zu funktionieren scheint. Auch Container lassen sich skalieren, jedoch muss der nicht zu vernachlässigende Aufwand der konkreten Implementierung des Skalierungsverhaltens beachtet werden. Wird ECS ohne Fargate oder ein Kubernetes-Service verwendet, muss sich zusätzlich um die Konfiguration des Clusters gekümmert werden. All das wird von Lambda vollständig übernommen, was zu enormen Einsparungen an Entwicklungskosten führen kann. Ist bereits eine Container-Architektur für den Betrieb einer anderen Anwendung aufgebaut worden oder die Nutzung begleitender containerisierter Services (z.B. einer bestimmten Datenbank) unabdingbar, kann eine Container-Anwendung vorzuziehen sein. Auch die Nutzung von Fargate er-

leichtert den Betrieb von Container-Clustern, führt aber zu Einschränkungen in Bezug auf die Konfiguration des Skalierungsverhaltens.

Bei den Service-Kosten muss abgewogen werden, welche Technologie für den konkreten Anwendungsfall und unter Betrachtung der normalen Last der Anwendung besser geeignet ist. Bei konstanter, relativ hoher Last auf dem Service ist es wahrscheinlich, dass das API-Gateway erhebliche Kosten verursacht und die Nutzung eines oder mehrerer Container sich als wirtschaftlicher erweist. An dieser Stelle könnte man die Verwendung von alternativen API-Gateways, bspw. Kong[22], erwägen. Hierbei muss jedoch beachtet werden, dass zusätzliche Gebühren für das Hosting und die Integration dieser Alternativen anfallen. Bei nur kurzzeitig hoher Spitzenlast oder geringer Nutzung des Services ist in Bezug auf die Gebühren vermutlich die Nutzung der FaaS-Anwendung geeigneter, da ansonsten für nicht genutzte Kapazität des Containers gezahlt werden muss. Besonders bei der erstmaligen Einführung einer Anwendung könnte es aufgrund der unbekannten Nutzungsrate günstiger sein, ein Serverless-Backend zu verwenden, um nicht für ungenutzte Kapazität zahlen zu müssen. Ist die normale Last der Applikation bekannt, können die Kosten der bestehenden Anwendung evaluiert werden und eventuell eine Container-Architektur aufgebaut werden. Andererseits wurde in den Kostenmodell-Vergleichen dieser Arbeit deutlich, dass einige Millionen-Aufrufe einer Lambda-Funktion monatlich möglich sind, bevor die Kosten einer einzigen Container-Instanz erreicht werden, was für viele kleinere Services ausreichend sein könnte. Des Weiteren ist zu empfehlen, stets die Last seiner Anwendung zu überwachen und mit dem Einsatz verschiedener Technologien wie Fargate oder Lambda zu experimentieren, um Kosten in der Entwicklung und beim Betrieb der Anwendung einzusparen. Es wird vorgeschlagen im Falle eines Backends die Kernfunktionalität zu extrahieren, um sie in beiden Services nutzbar zu machen. Diese könnten dann in einer Art A/B-Testing Strategie parallel oder abwechselnd betrieben und kontinuierlich evaluiert werden.

ZUSAMMENFASSUNG UND AUSBLICK

Im Rahmen dieser Bachelor-Thesis wurden Serverless-Funktionen mit [AWS Lambda](#) und Container-Anwendungen am Beispiel von [ECS](#) mit Fargate auf ihre Performance untersucht und deren Kostenmodelle analysiert. Dabei wurde deutlich, dass beide Services sowohl Vor- als auch Nachteile in Bezug auf die Performance des Backends hatten. Während Fargate eine allgemein bessere Performance und geringere Varianz aufwies, ist diese aber vor allem durch die CPU-Größe des Containers beschränkt. Lambda zeigte in den Tests größere Antwortzeiten als der Container, überzeugte aber in seinem Skalierungsverhalten bei langsamen und schnellen Anstiegen der virtuellen Benutzer. Es konnte jedoch ein Einfluss mehrerer Endpunkte auf die Antwortzeiten der Lambda-Anwendung gezeigt werden. Dagegen hatte bei der Container-Anwendung weder ein schneller Anstieg der Anfragen noch eine erhöhte Anzahl an Endpunkten eine Veränderung der Antwortzeiten zur Folge. Ebenso führte ein größerer Container oder die Nutzung zweier Instanzen nicht zu geringeren Antwortzeiten. Bei der Lambda-Anwendung hatte die Größe der Funktionen jedoch durchaus einen Einfluss auf die Funktionslaufzeit und damit auch auf die Antwortzeiten. In der Kostenuntersuchung der beiden Services wurde deutlich, dass diese, obwohl sich gewisse Werte im Voraus schätzen lassen, von der konkreten Nutzung der Anwendung abhängig sind. Besonders bei großer kontinuierlichen Last empfiehlt es sich, eine Lambda-Anwendung mit API-Gateway zu vermeiden und Alternativen wie Container zu bevorzugen.

Es gibt viele verschiedene Variablen die Einfluss auf die Performance von Serverless- und containerisierten Anwendungen nehmen und in zukünftigen Studien untersucht werden können. Beispielsweise könnte das in dieser Arbeit genutzte 50ms Delay variiert, oder durch Nutzung echter Services wie z.B. DynamoDB ersetzt werden. In dieser Arbeit wurde sich auf die Performance eines REST-Backends beschränkt. Dabei wurden verschiedene Container und Lambda-Größen untersucht und unterschiedliche Test-Szenarien durchgeführt. Es konnten aber unmöglich alle verschiedenen Konfigurationen betrachtet und evaluiert werden. Beispielsweise lassen sich sowohl die Größen der Lambda-Funktionen als auch die der Container-Instanzen noch auf mehrere Gigabyte erweitern. Des Weiteren lassen sich Container auf viele verschiedene Wege deployen und betreiben, z.B. auf einem selbst gemanagten [EC2](#) Cluster, mit [ECS](#), einem [EKS](#) Cluster oder mit Elastic Beanstalk. Auch Aspekte der Optimierung und des unter Umständen komplexen Skalierungsverhaltens einer Container-Anwendung können in zukünftigen Performance-Tests betrachtet werden.

Immer häufiger werden auch Mischformen von Containern und Serverless-Funktionen. Beispielsweise bietet [AWS](#) seit Ende 2020 an, Container über

Lambda verfügbar zu machen[5]. Darüber hinaus könnten Container und FaaS-Angebote verschiedener Cloud-Anbieter verglichen werden, zum Beispiel von Microsoft Azure oder Google Cloud Platform.

Die in dieser Arbeit durchgeführten Tests waren auf eine einzige Beispielanwendung beschränkt. Da sich aber für jede beliebige Anwendung Performance-Tests durchführen lassen, könnte das für diese Arbeit genutzte Testing-System in Zukunft ausgebaut werden, damit es für alle möglichen Anwendungs-Architekturen nutzbar ist. In das Analyse-System könnten weitere Metriken aus AWS CloudWatch, wie z.B. die tatsächlichen Funktionskosten, und andere Container-Services wie EKS oder Elastic Beanstalk integriert und somit der Vergleich beider Technologien noch weiter verbessert werden. Dadurch könnte das Tool Organisationen praktisch dabei helfen, die Performance und Kosten der Nutzung von Cloud-Services besser zu evaluieren.

LITERATUR

- [1] *200 OK - HTTP* | MDN. URL: <https://developer.mozilla.org/de/docs/Web/HTTP/Status/200> (besucht am 10.02.2021).
- [2] *500 Internal Server Error - HTTP* | MDN. URL: <https://developer.mozilla.org/de/docs/Web/HTTP/Status/500> (besucht am 10.02.2021).
- [3] *503 Service Unavailable - HTTP* | MDN. URL: <https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/503> (besucht am 10.02.2021).
- [4] *AWS Fargate-Preise* | *Serverless Compute Engine* | Amazon Web Services. de-DE. URL: <https://aws.amazon.com/de/fargate/pricing/> (besucht am 02.02.2021).
- [5] *AWS Lambda now supports container images as a packaging format*. en-US. URL: <https://aws.amazon.com/about-aws/whats-new/2020/12/aws-lambda-now-supports-container-images-as-a-packaging-format/> (besucht am 07.01.2021).
- [6] *Amazon API Gateway – Preise* | *API-Management* | Amazon Web Services. de-DE. URL: <https://aws.amazon.com/de/api-gateway/pricing/> (besucht am 02.02.2021).
- [7] Amazon AWS. *AWS Lambda - Developer Guide*. en. 2020. URL: <https://docs.aws.amazon.com/lambda/latest/dg/lambda-dg.pdf>.
- [8] *Amazon CloudWatch Concepts* - Amazon CloudWatch. URL: https://docs.aws.amazon.com/AmazonCloudWatch/latest/monitoring/cloudwatch_concepts.html (besucht am 24.02.2021).
- [9] *Amazon ECS - User Guide for AWS Fargate*. en. URL: <https://docs.aws.amazon.com/AmazonECS/latest/userguide/ecs-ug.pdf> (besucht am 15.01.2021).
- [10] *CPU-Optionen optimieren* - Amazon Elastic Compute Cloud. URL: https://docs.aws.amazon.com/de_de/AWSEC2/latest/UserGuide/instance-optimize-cpu.html (besucht am 13.02.2021).
- [11] Alex Casalboni. *alexcasalboni/aws-lambda-power-tuning*. original-date: 2017-03-27T15:18:12Z. Jan. 2021. URL: <https://github.com/alexcasalboni/aws-lambda-power-tuning> (besucht am 06.01.2021).
- [12] *Cluster Autoscaler* - Amazon EKS. URL: <https://docs.aws.amazon.com/eks/latest/userguide/cluster-autoscaler.html> (besucht am 02.02.2021).
- [13] Alex DeBrie. *AWS API Performance Comparison: Serverless vs. Containers vs. API Gateway integration*. en. Feb. 2019. URL: <https://alexdebrie.com/posts/aws-api-performance-comparison/> (besucht am 22.01.2021).

- [14] *Error 502 Bad Gateway: Wo liegt das Problem?* de. URL: <https://www.ionos.de/digitalguide/hosting/hosting-technik/was-bedeutet-502-bad-gateway-erklaerung-loesung/> (besucht am 10.02.2021).
- [15] *HTTP response status codes - HTTP | MDN*. URL: <https://developer.mozilla.org/de/docs/Web/HTTP/Status> (besucht am 13.01.2021).
- [16] Scott Hendrickson, Stephen Sturdevant, Tyler Harter, Venkateshwaran Venkataramani, Andrea C Arpaci-Dusseau und Remzi H Arpaci-Dusseau. *Serverless Computation with OpenLambda*. en. 2017. URL: https://www.usenix.org/system/files/conference/hotcloud16/hotcloud16_hendrickson.pdf.
- [17] *Horizontal Pod Autoscaler - Amazon EKS*. URL: <https://docs.aws.amazon.com/eks/latest/userguide/horizontal-pod-autoscaler.html> (besucht am 02.02.2021).
- [18] *Horizontal Pod Autoscaler*. de. URL: <https://kubernetes.io/de/docs/tasks/run-application/horizontal-pod-autoscale/> (besucht am 02.02.2021).
- [19] *How to Reduce Docker Image Size - DZone Cloud*. en. URL: <https://dzone.com/articles/how-to-reduce-docker-image-size> (besucht am 03.02.2021).
- [20] Prerna Jain, Yogesh Munjal, Jatin Gera und Pooja Gupta. "Performance Analysis of Various Server Hosting Techniques". en. In: *Procedia Computer Science* 173 (2020), S. 70–77. ISSN: 18770509. DOI: [10.1016/j.procs.2020.06.010](https://doi.org/10.1016/j.procs.2020.06.010). URL: <https://linkinghub.elsevier.com/retrieve/pii/S187705092031512X> (besucht am 01.02.2021).
- [21] Ken Owens, Sarah Allen, Ben Browning, Lee Calcote, Amir Chaudhry, Doug Davis und Louis Fourie. *CNCF WG-Serverless Whitepaper v1.0*. en. 2018. URL: https://github.com/cncf/wg-serverless/blob/master/whitepapers/serverless-overview/cncf_serverless_whitepaper_v1.0.pdf (besucht am 08.01.2021).
- [22] *Kong/kong*. original-date: 2014-11-17T23:56:08Z. Feb. 2021. URL: <https://github.com/Kong/kong> (besucht am 21.02.2021).
- [23] *Lambda Preise – Amazon Web Services (AWS)*. de-DE. URL: <https://aws.amazon.com/de/lambda/pricing/> (besucht am 07.01.2021).
- [24] *Lambda concepts - AWS Lambda*. URL: <https://docs.aws.amazon.com/lambda/latest/dg/gettingstarted-concepts.html> (besucht am 07.01.2021).
- [25] *Load testing for engineering teams | k6*. en. URL: <https://k6.io> (besucht am 13.01.2021).
- [26] Nathan Malishev. *AWS Lambda Cold Start Language Comparisons, 2019 edition | by Nathan Malishev | Level Up Coding*. Sep. 2019. URL: <https://levelup.gitconnected.com/aws-lambda-cold-start-language-comparisons-2019-edition-%EF%B8%8F-1946d32a0244> (besucht am 03.01.2021).

- [27] G. McGrath und P. R. Brenner. "Serverless Computing: Design, Implementation, and Performance". In: *2017 IEEE 37th International Conference on Distributed Computing Systems Workshops (ICDCSW)*. ISSN: 2332-5666. Juni 2017, S. 405–410. DOI: [10.1109/ICDCSW.2017.36](https://doi.org/10.1109/ICDCSW.2017.36).
- [28] Metrics. en. URL: <https://k6.io/docs/using-k6/metrics> (besucht am 13. 01. 2021).
- [29] Ian Molyneaux. *The Art of Application Performance Testing, 2nd Edition*. en. 2. Aufl. O'Reilly Media, Inc., Dez. 2014. ISBN: 978-1-4919-0054-3. URL: <https://learning.oreilly.com/library/view/the-art-of/9781491900536/> (besucht am 20. 01. 2021).
- [30] Alessandro V. Papadopoulos, Laurens Versluis, André Bauer, Nikolas Herbst, Jóakim von Kistowski, Ahmed Ali-eldin, Christina L. Abad, José N. Amaral, Petr Tůma und Alexandru Iosup. "Methodological Principles for Reproducible Performance Evaluation in Cloud Computing". In: *IEEE Transactions on Software Engineering* (2019). Conference Name: IEEE Transactions on Software Engineering, S. 1–1. ISSN: 1939-3520. DOI: [10.1109/TSE.2019.2927908](https://doi.org/10.1109/TSE.2019.2927908).
- [31] *Preise für Amazon CloudWatch – Amazon Web Services (AWS)*. de-DE. URL: <https://aws.amazon.com/de/cloudwatch/pricing/> (besucht am 24. 02. 2021).
- [32] *Preise für Elastic Load Balancing – Amazon Web Services*. de-DE. URL: <https://aws.amazon.com/de/elasticloadbalancing/pricing/> (besucht am 09. 02. 2021).
- [33] *Running large tests*. en. URL: <https://k6.io/docs/testing-guides/running-large-tests> (besucht am 21. 01. 2021).
- [34] Joel Scheuner und Philipp Leitner. "Function-as-a-Service performance evaluation: A multivocal literature review". en. In: *Journal of Systems and Software* 170 (Dez. 2020), S. 110708. ISSN: 01641212. DOI: [10.1016/j.jss.2020.110708](https://doi.org/10.1016/j.jss.2020.110708). URL: <https://linkinghub.elsevier.com/retrieve/pii/S0164121220301527> (besucht am 18. 01. 2021).
- [35] *Serverless Architectures with AWS Lambda*. de. Nov. 2017. URL: <https://d1.awsstatic.com/whitepapers/serverless-architectures-with-aws-lambda.pdf> (besucht am 04. 02. 2021).
- [36] *Vertical Pod Autoscaler - Amazon EKS*. URL: <https://docs.aws.amazon.com/eks/latest/userguide/vertical-pod-autoscaler.html> (besucht am 02. 02. 2021).
- [37] M. Villamizar u. a. "Infrastructure Cost Comparison of Running Web Applications in the Cloud Using AWS Lambda and Monolithic and Microservice Architectures". In: *2016 16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*. Mai 2016, S. 179–182. DOI: [10.1109/CCGrid.2016.37](https://doi.org/10.1109/CCGrid.2016.37).
- [38] *Was ist Container-Orchestrierung?* de. URL: <https://www.redhat.com/de/topics/containers/what-is-container-orchestration> (besucht am 31. 12. 2020).

- [39] *Was ist Kubernetes?* de. URL: <https://kubernetes.io/de/docs/concepts/overview/what-is-kubernetes/> (besucht am 31. 12. 2020).
- [40] *Was ist Serverless?* de. Dez. 2016. URL: <https://serverless-stack.com/chapters/de/what-is-serverless.html> (besucht am 31. 12. 2020).
- [41] *What is Load Testing? How to create a Load Test in k6.* en. URL: <https://k6.io/docs/test-types/load-testing> (besucht am 10. 02. 2021).
- [42] *What is Stress Testing? How to create a Stress Test in k6.* en. URL: <https://k6.io/docs/test-types/stress-testing> (besucht am 10. 02. 2021).
- [43] *What is a Container? | App Containerization | Docker.* en. URL: <https://www.docker.com/resources/what-container> (besucht am 31. 12. 2020).
- [44] *expressjs/express.* original-date: 2009-06-26T18:56:01Z. Feb. 2021. URL: <https://github.com/expressjs/express> (besucht am 13. 02. 2021).
- [45] *pandas - Python Data Analysis Library.* URL: <https://pandas.pydata.org/> (besucht am 25. 01. 2021).
- [46] *serverless/serverless: Serverless Framework – Build web, mobile and IoT applications with serverless architectures using AWS Lambda, Azure Functions, Google CloudFunctions & more! –.* URL: <https://github.com/serverless/serverless> (besucht am 13. 02. 2021).

Teil II

APPENDIX

TASK-DEFINITIONEN

Folgende Fargate Task-Definitionen wurden für die Container-Tests verwendet. In der Sektion „containerDefinitions“ sind alle definierten Container, deren Container-Image, Arbeitsspeicher („memory“) und die Anzahl der vCPUs („cpu“) definiert. Außerhalb dieser Sektion sind die allgemeinen Werte dieses Tasks aufgeführt.

A.1 128MB CONTAINER

```
{
  "ipcMode": null,
  "executionRoleArn": "arn:aws:iam::734730000614:role/ecsTaskExecutionRole",
  "containerDefinitions": [
    {
      "dnsSearchDomains": null,
      "environmentFiles": null,
      "logConfiguration": {
        "logDriver": "awslogs",
        "secretOptions": null,
        "options": {
          "awslogs-group": "/ecs/notes-express-task-definition",
          "awslogs-region": "eu-central-1",
          "awslogs-stream-prefix": "ecs"
        }
      },
      "entryPoint": [],
      "portMappings": [
        {
          "hostPort": 80,
          "protocol": "tcp",
          "containerPort": 80
        }
      ],
      "command": [],
      "linuxParameters": null,
      "cpu": 128,
      "environment": [],
      "resourceRequirements": null,
      "ulimits": null,
      "dnsServers": null,
      "mountPoints": [],
```

```

    "workingDirectory": null,
    "secrets": null,
    "dockerSecurityOptions": null,
    "memory": 128,
    "memoryReservation": null,
    "volumesFrom": [],
    "stopTimeout": null,
    "image": "734730000614.dkr.ecr.eu-central-1.amazonaws.com
      /notes-express:latest",
    "startTimeout": null,
    "firelensConfiguration": null,
    "dependsOn": null,
    "disableNetworking": null,
    "interactive": null,
    "healthCheck": null,
    "essential": true,
    "links": [],
    "hostname": null,
    "extraHosts": null,
    "pseudoTerminal": null,
    "user": null,
    "readonlyRootFilesystem": null,
    "dockerLabels": null,
    "systemControls": null,
    "privileged": null,
    "name": "custom"
  }
],
"placementConstraints": [],
"memory": "512",
"taskRoleArn": null,
"compatibilities": [
  "EC2",
  "FARGATE"
],
"taskDefinitionArn": "arn:aws:ecs:eu-central-1:734730000614:task-definition
  /notes-express-task-definition:2",
"family": "notes-express-task-definition",
"requiresAttributes": [
  {
    "targetId": null,
    "targetType": null,
    "value": null,
    "name": "com.amazonaws.ecs.capability.logging-driver.awslogs"
  },
  {

```

```

        "targetId": null,
        "targetType": null,
        "value": null,
        "name": "ecs.capability.execution-role-awslogs"
    },
    {
        "targetId": null,
        "targetType": null,
        "value": null,
        "name": "com.amazonaws.ecs.capability.ecr-auth"
    },
    {
        "targetId": null,
        "targetType": null,
        "value": null,
        "name": "com.amazonaws.ecs.capability.docker-remote-api.1.19"
    },
    {
        "targetId": null,
        "targetType": null,
        "value": null,
        "name": "ecs.capability.execution-role-ecr-pull"
    },
    {
        "targetId": null,
        "targetType": null,
        "value": null,
        "name": "com.amazonaws.ecs.capability.docker-remote-api.1.18"
    },
    {
        "targetId": null,
        "targetType": null,
        "value": null,
        "name": "ecs.capability.task-eni"
    }
],
"pidMode": null,
"requiresCompatibilities": [
    "FARGATE"
],
"networkMode": "awsvpc",
"cpu": "256",
"revision": 2,
"status": "ACTIVE",
"inferenceAccelerators": null,
"proxyConfiguration": null,

```

```

    "volumes": []
}

```

A.2 256MB CONTAINER

```

{
  "ipcMode": null,
  "executionRoleArn": "arn:aws:iam::734730000614:role/ecsTaskExecutionRole",
  "containerDefinitions": [
    {
      "dnsSearchDomains": null,
      "environmentFiles": null,
      "logConfiguration": {
        "logDriver": "awslogs",
        "secretOptions": null,
        "options": {
          "awslogs-group": "/ecs/notes-express-task-definition",
          "awslogs-region": "eu-central-1",
          "awslogs-stream-prefix": "ecs"
        }
      },
      "entryPoint": [],
      "portMappings": [
        {
          "hostPort": 80,
          "protocol": "tcp",
          "containerPort": 80
        }
      ],
      "command": [],
      "linuxParameters": null,
      "cpu": 256,
      "environment": [],
      "resourceRequirements": null,
      "ulimits": null,
      "dnsServers": null,
      "mountPoints": [],
      "workingDirectory": null,
      "secrets": null,
      "dockerSecurityOptions": null,
      "memory": 256,
      "memoryReservation": null,
      "volumesFrom": [],
      "stopTimeout": null,
      "image": "734730000614.dkr.ecr.eu-central-1.amazonaws.com/notes-express:latest",
    }
  ]
}

```

```

    "startTimeout": null,
    "firelensConfiguration": null,
    "dependsOn": null,
    "disableNetworking": null,
    "interactive": null,
    "healthCheck": null,
    "essential": true,
    "links": [],
    "hostname": null,
    "extraHosts": null,
    "pseudoTerminal": null,
    "user": null,
    "readonlyRootFilesystem": null,
    "dockerLabels": null,
    "systemControls": null,
    "privileged": null,
    "name": "custom"
  }
],
"placementConstraints": [],
"memory": "512",
"taskRoleArn": null,
"compatibilities": [
  "EC2",
  "FARGATE"
],
"taskDefinitionArn": "arn:aws:ecs:eu-central-1:734730000614:task-definition
/notes-express-task-definition-256:1",
"family": "notes-express-task-definition-256",
"requiresAttributes": [
  {
    "targetId": null,
    "targetType": null,
    "value": null,
    "name": "com.amazonaws.ecs.capability.logging-driver.awslogs"
  },
  {
    "targetId": null,
    "targetType": null,
    "value": null,
    "name": "ecs.capability.execution-role-awslogs"
  },
  {
    "targetId": null,
    "targetType": null,
    "value": null,

```



```

        "name": "com.amazonaws.ecs.capability.ecr-auth"
    },
    {
        "targetId": null,
        "targetType": null,
        "value": null,
        "name": "com.amazonaws.ecs.capability.docker-remote-api.1.19"
    },
    {
        "targetId": null,
        "targetType": null,
        "value": null,
        "name": "ecs.capability.execution-role-ecr-pull"
    },
    {
        "targetId": null,
        "targetType": null,
        "value": null,
        "name": "com.amazonaws.ecs.capability.docker-remote-api.1.18"
    },
    {
        "targetId": null,
        "targetType": null,
        "value": null,
        "name": "ecs.capability.task-eni"
    }
],
"pidMode": null,
"requiresCompatibilities": [
    "FARGATE"
],
"networkMode": "awsvpc",
"cpu": "256",
"revision": 1,
"status": "ACTIVE",
"inferenceAccelerators": null,
"proxyConfiguration": null,
"volumes": []
}

```

A.3 512MB CONTAINER

```

{
    "ipcMode": null,
    "executionRoleArn": "arn:aws:iam::734730000614:role/ecsTaskExecutionRole",
    "containerDefinitions": [

```

```

{
  "dnsSearchDomains": null,
  "environmentFiles": null,
  "logConfiguration": {
    "logDriver": "awslogs",
    "secretOptions": null,
    "options": {
      "awslogs-group": "/ecs/notes-express-task-definition",
      "awslogs-region": "eu-central-1",
      "awslogs-stream-prefix": "ecs"
    }
  },
  "entryPoint": [],
  "portMappings": [
    {
      "hostPort": 80,
      "protocol": "tcp",
      "containerPort": 80
    }
  ],
  "command": [],
  "linuxParameters": null,
  "cpu": 512,
  "environment": [],
  "resourceRequirements": null,
  "ulimits": null,
  "dnsServers": null,
  "mountPoints": [],
  "workingDirectory": null,
  "secrets": null,
  "dockerSecurityOptions": null,
  "memory": 512,
  "memoryReservation": null,
  "volumesFrom": [],
  "stopTimeout": null,
  "image": "734730000614.dkr.ecr.eu-central-1.amazonaws.com
    /notes-express:latest",
  "startTimeout": null,
  "firelensConfiguration": null,
  "dependsOn": null,
  "disableNetworking": null,
  "interactive": null,
  "healthCheck": null,
  "essential": true,
  "links": [],
  "hostname": null,

```

```

        "extraHosts": null,
        "pseudoTerminal": null,
        "user": null,
        "readonlyRootFilesystem": null,
        "dockerLabels": null,
        "systemControls": null,
        "privileged": null,
        "name": "custom"
    }
],
"placementConstraints": [],
"memory": "1024",
"taskRoleArn": null,
"compatibilities": [
    "EC2",
    "FARGATE"
],
"taskDefinitionArn": "arn:aws:ecs:eu-central-1:734730000614:task-definition
    /notes-express-task-definition-512:1",
"family": "notes-express-task-definition-512",
"requiresAttributes": [
    {
        "targetId": null,
        "targetType": null,
        "value": null,
        "name": "com.amazonaws.ecs.capability.logging-driver.awslogs"
    },
    {
        "targetId": null,
        "targetType": null,
        "value": null,
        "name": "ecs.capability.execution-role-awslogs"
    },
    {
        "targetId": null,
        "targetType": null,
        "value": null,
        "name": "com.amazonaws.ecs.capability.ecr-auth"
    },
    {
        "targetId": null,
        "targetType": null,
        "value": null,
        "name": "com.amazonaws.ecs.capability.docker-remote-api.1.19"
    },
    {

```

```

        "targetId": null,
        "targetType": null,
        "value": null,
        "name": "ecs.capability.execution-role-ecr-pull"
    },
    {
        "targetId": null,
        "targetType": null,
        "value": null,
        "name": "com.amazonaws.ecs.capability.docker-remote-api.1.18"
    },
    {
        "targetId": null,
        "targetType": null,
        "value": null,
        "name": "ecs.capability.task-eni"
    }
],
"pidMode": null,
"requiresCompatibilities": [
    "FARGATE"
],
"networkMode": "awsvpc",
"cpu": "512",
"revision": 1,
"status": "ACTIVE",
"inferenceAccelerators": null,
"proxyConfiguration": null,
"volumes": []
}

```

REPOSITORY

In diesem Kapitel wird der Aufbau des Code-Repositories erklärt. Es ist unter der URL <https://github.com/rolule/ba> auffindbar.

Das Repository ist in zwei Ordner unterteilt. Im Ordner „thesis“ ist der LaTeX-Source Code dieser Arbeit zu finden. Der Ordner „code“ enthält alle für diese Arbeit verwendeten Quellcode-Dateien, Test-Skripte und Test-Artefakte. Zusätzlich ist auf der Wurzel-Ebene des Repositories die PDF-Version dieser Arbeit zu finden. Im folgenden wird sich ausschließlich auf den Ordner „code“ bezogen.

B.1 ANWENDUNGS-QUELLCODE

B.1.1 *Container*

Der Quellcode der Container-Anwendung ist im Unterordner „container/notes-express“ zu finden. Er enthält die zum Bauen des Container-Images benötigte Dockerfile, welche mit Hilfe des Shell-Skriptes „build.sh“ gebaut werden kann. Zum Bauen des Abbildes wird ein laufender Docker-Daemon benötigt. Die Anwendung kann nach dem Bauprozess mit dem Skript „run.sh“ auf Port 80 gestartet werden. Der Server sollte beim Aufrufen der URL <http://localhost/notes> im Web-Browser eine in JSON formatierte Liste von Notizen anzeigen.

B.1.2 *Serverless*

Der Quellcode der Serverless-Anwendung ist im Unterordner „serverless/notes-serverless“ zu finden. Zum Deployen und Ausführen der Anwendung ist das Serverless Framework und ein eingeloggter [AWS](#) Benutzer notwendig. Sie wurde aber so konzipiert, dass sie sich genau so wie der Container verhält. Der Code der einzelnen Funktionen ist in den Dateien „get.js“, „list.js“, „update.js“ und „create.js“ zu finden. Die Datei „serverless.yml“ enthält die Konfiguration des Serverless-Frameworks.

B.2 TEST DATEIEN

Der Ordner „tests“ enthält alle Dateien die zur Ausführung und Auswertung der Tests notwendig sind. Auch die Test-Artefakte sind hier enthalten. Die im folgenden beschriebenen Dateien befinden sich alle innerhalb dieses Ordners. Um die Skripte in diesem Ordner ausführen zu können, werden einige Abhängigkeiten benötigt. Da auch die Load-Generatoren diese benötigen, wurde ein Shell-Skript geschrieben, welches diese installiert und konfigu-

riert. Es ist unter dem Dateinamen „setup.sh“ zu finden und kann auf einer [EC2](#) Instanz ausgeführt werden, um die Tests-Umgebung zu initialisieren.

B.2.1 *Test Skripte und Konfiguration*

Das Skript „test.js“ wurde erstellt, um die Tests auf den Load-Generatoren leichter ausführen zu können. Es wurde in Node.js geschrieben und um es auszuführen, muss Node.js und der Node Paket Manager (npm) installiert sein und die Abhängigkeiten mittels des Befehls `npm install` installiert worden sein. Zur einfacheren Ausführung wurde ein Shell-Skript mit dem Namen „test.sh“ geschrieben, dass die Node-Applikation mit allen übergebenen Parametern aufruft. Das Test-Skript dient der Konfiguration und Ausführung jedes einzelnen durchgeführten Tests. Es bittet den Benutzer um eine Eingabe des ausgewählten Services (Lambda oder Fargate), die konfigurierte Größe (128MB, 256MB, 512MB), den Use-Case (a, b, c) und den Test-Strategien (pipe, stress, load, spike). Zusätzlich wird für Fargate die Anzahl der Tasks, also der Container-Instanzen abgefragt. Die Use-Case Skripte befinden sich im Unterordner „use-cases“ und werden in der von dem Testing-Tool k6 verstandenen Syntax (in JavaScript) geschrieben. Die Konfigurationen der einzelnen Test-Strategien sind im Unterordner „strategies“ zu finden und werden k6 bei der Ausführung als Optionen übergeben.

Hat der Benutzer alle Angaben gemacht, wird k6 mit den ausgewählten Test-Einstellungen ausgeführt. Nach Terminierung des Tests, werden dem Benutzer die von k6 ermittelten Metriken angezeigt. Darunter ist die durchschnittliche (avg), minimale (min), maximale (max), mediane Antwortzeit (med) und die 0,90 und 0,95 Quantile. Zusätzlich wird die Gesamtzahl aller Requests dieses Test-Durchlaufs und die Anzahl der durchschnittlichen Requests pro Sekunde ausgegeben. Für weitere Metriken wie die Standardabweichung und den Variationskoeffizienten, werden die Python-Analysertools (siehe [B.2.3](#)) verwendet. Die Metriken der maximalen CPU-Auslastung, maximale Funktions-Dauer, maximale Coldstart-Dauer und maximale Nebenläufigkeit werden von der [AWS](#) CloudWatch Konsole abgelesen.

B.2.2 *Test-Artefakte*

Die Artefakte der Tests sind nach Service aufgeteilt und in den Ordnern „fargate“ und „lambda“ zu finden. Diese sind weiterhin unterteilt in die unterschiedlichen Use-Cases (a, b und c). In den Use-Case Ordnern befinden sich weitere Unterordner, die die Tests in die ausgewählte Konfigurations-Größe (wie viel RAM/CPU) eingeteilt werden. Bei Fargate wird mit dem Präfix 2x darauf hingewiesen, dass hier zwei Tasks verwendet wurden. Eine Ebene tiefer sind die unterschiedlichen Test-Strategien aufgeführt. In deren Subordnern befinden sich die Test-Artefakte, die als `tar.gz` Archive abgespeichert sind. Der Name setzt sich dabei immer aus dem Datum in umgekehrter Schreibweise (also aus dem 26.02. wird 0226) und der Uhrzeit des Tests ohne Doppelpunkt (also aus 14:03 wird 1403), verbunden mit einem Bindestrich,

zusammen (bspw. 0226-1403.tar.gz). Die Archive lassen sich zum Beispiel unter Linux-Systemen mit Hilfe des Befehls `tar xf 0226-1403.tar.gz` entpacken. Ein entpacktes Archiv enthält immer eine gleichnamige [CSV](#) und JSON Datei. Die JSON Datei enthält eine von k6 erstellte Zusammenfassung des Tests. Die [CSV](#) Datei enthält alle von jedem [VU](#) gemessene Metriken, die k6 unterstützt und erlaubt es somit den Testablauf, mit Hilfe der Analyse-Tools, genau zu rekonstruieren. Die Dateien mussten in Archive verpackt werden, da die [CSV](#) Dateien aufgrund mehrerer Tausend Zeilen sonst zu groß für das Repository geworden wären.

B.2.3 *Python Analyse*

Im Unterordner „pandas“ befinden sich die Analyse-Tools, die zur Auswertung in dieser Arbeit verwendet wurden. Es handelt sich um einige Python-Skripte, die die Datenanalyse-Bibliothek „Pandas“ verwenden und die von k6 erstellten [CSV](#) Dateien als Eingabe erwarten. Für die Auswertung aller relevanten von k6 erfassten Metriken (außer der durchschnittlichen Requests pro Sekunde) inklusive Standardabweichung und Variationskoeffizient kann das Skript „stats.py“ verwendet werden. Beispielweise kann mit dem Befehl `„python3 stats.py ../lambda/a/128/pipe/0127-1041.csv“` eine Analyse eines ausgeführten Test der Lambda-Anwendung für Use-Case A, die 128MB Variante und einen Pipe-Clean Tests vom 27. Januar um 10:41 Uhr durchgeführt werden. Wird anstatt der konkreten Datei der ganze „pipe/“ Ordner angegeben, wird die Analyse über alle [CSV](#) Dateien innerhalb des Ordners durchgeführt. Zur Auswertung von Fehlern eines Tests kann das Skript „errors.py“ verwendet werden. Das Skript „time.py“ dient zum Plotten des zeitlichen Verlaufs der Antwortzeit im Median. Identisch dazu ist „vus.py“, welches aber zusätzlich den Verlauf der virtuellen Benutzer anzeigt.

Es ist wichtig, dass die Skripte ausschließlich innerhalb des „pandas“ Ordners ausgeführt werden, da nur so der Pfad des Skriptes erfasst werden kann. Für die Ausführung ist eine Python3 Virtuelle-Umgebung notwendig und die Pakete „pandas“, „matplotlib“ und „openpyxl“ müssen mittels der Python-Paketmanagers „pip“ installiert werden (siehe „setup.sh“ Skript).