h_da
HOCHSCHULE DARMSTADT
UNIVERSITY OF APPLIED SCIENCES

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, 26. Februar 2021

Robin Luley

ABSTRACT

Der Betrieb von Anwendungen in der Cloud erfährt steigende Aufmerksamkeit. Public Cloud Anbieter erlauben es Organisationen, schneller und günstiger ihre Anwendungen und Daten zu provisionieren. Um die Vorteile des Cloud Deployments nutzen zu können, haben sich das Serverless Paradigma und die Containerisierung als relevante Entwicklungen herausgebildet. Während Serverless-Anwendungen fast vollständig von den Cloud Anbietern gemanaged werden und theoretisch unbegrenzt skalierbar sind, müssen Container Deployments in der Regel manuell konfiguriert und verwaltet werden.

Ziel der Arbeit ist es, die Performance von Serverless Anwendungen und containerisierten Anwendungen praktisch zu messen und zu vergleichen. Dazu wird eine klassische Beispielanwendung auf beide Arten deployed und der Durchsatz und die anfallenden Kosten mittels Lasttests geprüft. Es wird Amazon AWS als Testplattform verwendet, der Test-Aufbau soll aber auch die Nutzung anderer Cloud Anbieter ermöglichen.

Forschungsfrage
Wie unterscheiden sich der Betrieb von Containerisierten und Serverless (Lambda) Anwendungen in der Cloud (AWS) bezüglich der Performance und Skalierbarkeit?

ZUSAMMENFASSUNG

Kurze Zusammenfassung des Inhaltes in deutscher Sprache. Ungefähr eine Seite...

GANZ zum Schluss schreiben !

INHALTSVERZEICHNIS

ABBILDUNGSVERZEICHNIS

Teil I

THESIS

1 EINLEITUNG

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

1.1 MOTIVATION

Die Nutzung von Services der Public Cloud Anbieter wie Amazon AWS, Microsoft Azure oder Google Cloud Platform ermöglichen es Organisationen, ihre Anwendungen immer schneller, einfacher und kostengünstiger aufzusetzen und zu betreiben. Schon ein einfaches lift-and-shift von lokalen Rechenzentren hin zu virtuell bereitgestellten Instanzen bedeutet einige Vorteile in puncto Agilität, Kosteneinsparungen und Ausfallsicherheit. Durch Containerisierung und Orchestrierungs-Tools wie Kubernetes wurde es möglich, Anwendungen leichter zu skalieren und noch ausfallsicherer zu machen. Doch die von den Anbietern bereitgestellten Services werden immer raffinierter, um das Betreiben von Software noch weiter zu verbessern. Während beim Betrieb von containerisierten Anwendungen noch gewisse Parameter konfiguriert werden müssen, wird bei Serverless aufgestellten Anwendungen der Server an sich weg abstrahiert. Somit muss sich der Programmierer nicht einmal mehr um die Konfiguration eines Servers kümmern, sondern nur seinen Code schreiben, der dann automatisch von einem Server des Cloud Anbieters ausgeführt wird, wenn er soll und nur wenn er soll. Dies verspricht eine theoretisch unendliche Skalierbarkeit, noch einfacheres Deployment und weitere Kosteneinsparungen. Jedoch ist bisher unklar, wie sich die Nutzung von Serverless Anwendungen im Vergleich zu containerisierten Anwendungen verhält.

1.2 ZIELSETZUNG

Ziel der Arbeit ist es, die Performance von containerisierten und Serverless Anwendungen zu vergleichen anhand eines REST-Backends zu vergleichen.

Dazu soll eine Beispielanwendung REST-API für beide Technologien angepasst entwickelt werden und verschiedenen Lasttest unterzogen werden. Des Weiteren werden Optimierungsmöglichkeiten beider Technologien betrachtet und ebenfalls evaluiert.

1.3 GLIEDERUNG

Zunächst wird auf die Grundlagen von Serverless-Funktionen und Containerisierung eingegangen. Als nächstes soll ein Konzept erstellt werden, mit

dem die beiden Technologien evaluiert werden können. Schließlich wird die Evaluation praktisch durchgeführt und die Ergebnisse präsentiert.

2 GRUNDLAGEN

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

2.1 CLOUD COMPUTING

Laut der Definition des National Institute of Standards and Technology (NIST), ist Cloud Computing ein Modell, dass einen einfachen on-demand Zugriff auf eine Menge von EDV Ressourcen ermöglicht, welche schnell und ohne Interaktion eines Service Providers bereitgestellt oder freigegeben werden können[21]. Cloud Computing zeichnet sich demnach durch fünf Charakteristiken aus. Diese sind die bedarfsorientierte Bereitstellung von Ressourcen die durch einen Systembenutzer ausgelöst wird, den einfachen Zugriff über heterogene Plattformen hinweg durch standardisierte Netzwerkmechanismen, das Pooling von Servern und die schnelle und automatisierte Skalierung, scheinbar unendlich verfügbare Ressourcen und das dauerhafte Messen der Service Aktivitäten.
Darüber hinaus gehen aus der Definition verschiedene Servicemodelle hervor, die alle den Suffix „as a service“, also „als Dienstleistung“, gemeinsam haben. In der Definition des NIST werden die grundlegenden Servicemodelle IaaS, PaaS und SaaS genannt. Im weiterführenden ClouNS Referenz Modell[17], das in Abbildung 2.1 zu sehen ist, werden sie um CaaS erweitert.

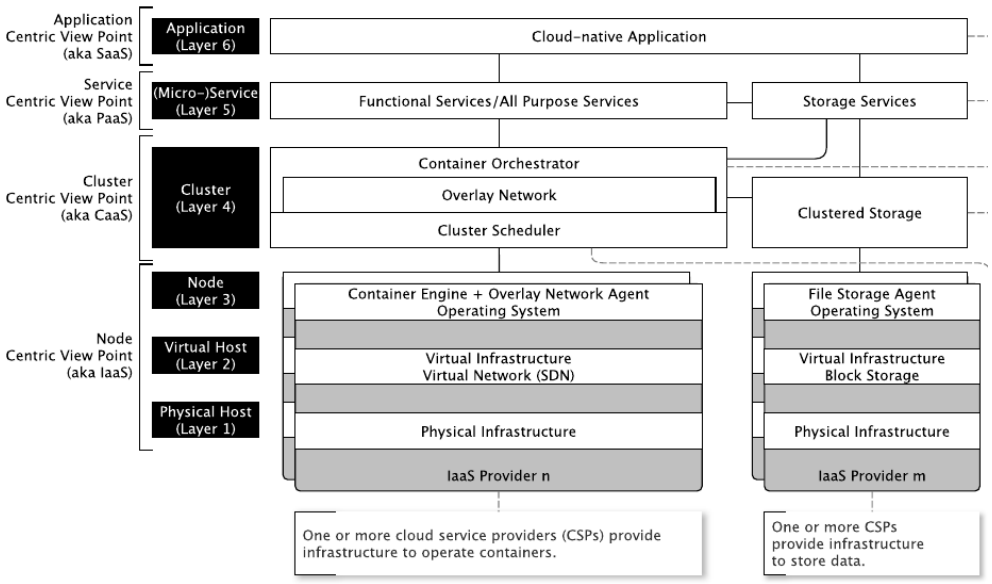

Abbildung 2.1: Der ClouNS Stack[17]

2.1.1 *Infrastructure as a Service (IaaS)*

Infrastructure as a Service bietet die Grundlage des Cloud Native Stacks. Einem Kunden werden hierbei grundlegende zumeist virtualisierte Hardware-Ressourcen wie Prozessorleistung, Speicher oder Netzwerkmöglichkeiten angeboten. Auf diesen kann er seine Software beliebig ausführen. Er hat Kontrolle über Betriebssysteme, Anwendungen und verwendeten Speicher seiner Instanzen[21].

2.1.2 *Cluster-as-a-Service Container-as-a-Service (CaaS)*

Bei Cluster as a Service handelt es sich um eine Schicht über IaaS, die ein Clustering von Containern bietet. Hier agieren Orchestrierungs Tools wie zum Beispiel Kubernetes, die Container auf das Cluster verteilen, skalieren und managen[17].

2.1.3 *Platform as a Service (PaaS)*

PaaS liegt eine Ebene über IaaS oder CaaS. Einem Kunden wird auch hier die Kontrolle über die in der Cloud Infrastruktur auszuführende Anwendung gegeben. Allerdings kann er nicht über verwendete Betriebssysteme, Speichernutzung, Netzwerkkonfiguration usw. entscheiden. Dafür werden ihm vom Anbieter Programmier- oder Laufzeitumgebungen, Tools und Bibliotheken angeboten, die die Entwicklung und Ausführung der Anwendung auf der Infrastruktur ermöglichen[21].

2.1.4 *Software as a Service (SaaS)*

SaaS ist eine höhere Abstraktionsebene über PaaS. Einem Kunden werden hier vom Anbieter explizite auf Cloud Infrastruktur auszuführende Anwendungen bereitgestellt, die dieser verwenden kann. Der Kunde hat weder Einfluss auf die zugrundeliegende Cloud Infrastruktur, noch auf verwendete Betriebssysteme oder Software. Die Ausnahme bilden limitierte Einstellungsmöglichkeiten der vom Anbieter bereitgestellten Anwendung[21].

2.2 CLOUD NATIVE COMPUTING

Cloud Native Technologien oder Cloud Native Computing ist laut der Cloud Native Computing Foundation (CNCF) eine Ansammlung von Technologien, die es Unternehmen ermöglichen „skalierbare Anwendungen in modernen, dynamischen Umgebungen zu implementieren und zu betreiben“ [35]. Bei diesen dynamischen Umgebungen handelt es sich dabei ausschließlich

um Cloud Umgebungen (also private, öffentliche und hybride Clouds). Diese Systeme sollen lose gekoppelt, „belastbar, handhabbar und beobachtbar“ sein und durch Automatisierung schnelle Änderungen ermöglichen[35].

2.3 CONTAINERISIERUNG

Containerisierung ist eine Technik, den Code einer Anwendung mitsamt aller Abhängigkeiten, die diese zur Ausführung benötigt, zu verpacken. Ein solches Paket wird als Container Image bezeichnet. Da alle Abhängigkeiten enthalten sind, werden dadurch die Laufzeitunterschiede auf unterschiedlichen Plattformen minimiert[33]. 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[33].

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 können Container-Images deutlich leichtgewichtiger und schneller als Virtuelle Maschinen ausgeführt werden. Abbildung 2.2 verdeutlicht nochmals den Unterschied beider Technologien.

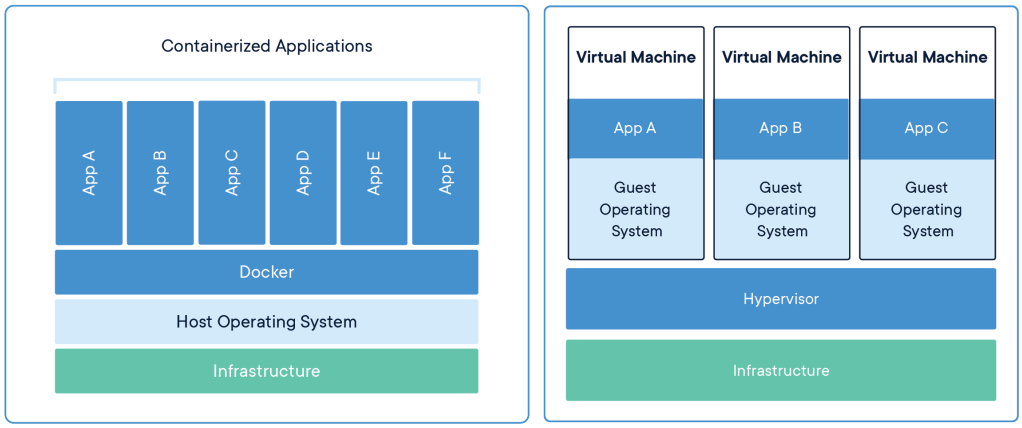

Abbildung 2.2: Vergleich zwischen Containern und Virtuellen Maschinen[33]

2.4 CONTAINER-ORCHESTRIERUNG

Container-Orchestrierung beschreibt die Automatisierung der „Bereitstellung, Verwaltung, Skalierung und Vernetzung von Containern“[30]. Mit Orchestrierungs-Tools lassen sich Container einfach auf Computer-Clustern provisionieren, skalieren und überwachen. Wie im ClouNS Modell (Abb. 2.1) zu sehen ist, agieren Container Orchestrierungs Tools auf der CaaS Ebene. Dadurch erlauben sie das Betreiben von infrastrukturunabhängigen und flexiblen Anwen-

dungen auf Basis von Containern[31]. Beispiele für bekannte Orchestrierungs-Tools sind Kubernetes oder AWS Elastic Container Service (ECS).

2.5 SERVERLESS

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

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[16]. Beispiele für BaaS sind Amazon AWS Simple Storage Service (S3) als Speicher von Dateien oder AWS DynamoDB als NoSQL-Datenbank.

In dieser Arbeit sollen ausschließlich Functions-as-a-Service betrachtet werden. Dabei 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 eintrifft. Serverless verspricht dadurch eine effizientere Entwicklung und Kosteneinsparungen, da bei den Anbietern nur der tatsächlich laufenden Code abgerechnet wird[32]. Im ClouNS Modell (Abb. 2.1) ist FaaS also eine Ebene über der von CaaS einzustufen, da es eine Plattform zur Entwicklung von Software darstellt, die Container und deren Orchestrierung als Basis nutzen.

2.6 AWS LAMBDA

Die Serverless Anwendung dieser Arbeit soll auf AWS Lambda ausgeführt werden. Dabei handelt es sich um ein FaaS Angebot von Amazon AWS. Die Funktionsweise von Lambda lässt sich wie folgt beschreiben[6]:

1. Es tritt ein Event ein, dass die Ausführung der Lambda-Funktion anfordert.
   Dabei kann es sich um alle mögliche Ereignisse handeln. Zum Beispiel ein Dateiupload in S3, einen CRON-Job oder im Falle eines Web Backends eine API-Aufruf mittels AWS 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 von Anwender konfigurieren:

a) Die Runtime: Dabei handelt es sich um die verwendete Programmiersprache bzw. Framework. Als Optionen bietet Lambda beispielsweise Node.js 12.x, Java, Python oder Go.

b) Die Arbeitsspeichergröße: Wie viel Speicher der ausgeführten Funktion bereitsteht. Die Mindestgröße beträgt hier 128MB; maximal sind etwas mehr als 10GB möglich. Proportional zur Speichergröße bestimmt Lambda ebenfalls die CPU Leistung, 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 (Speicher, CPU) erstellt, der Code der Funktion geladen und entpackt und der Initialisierungscode der Lambda Funktion (nicht die Funktion an sich) ausgeführt. Dort 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.

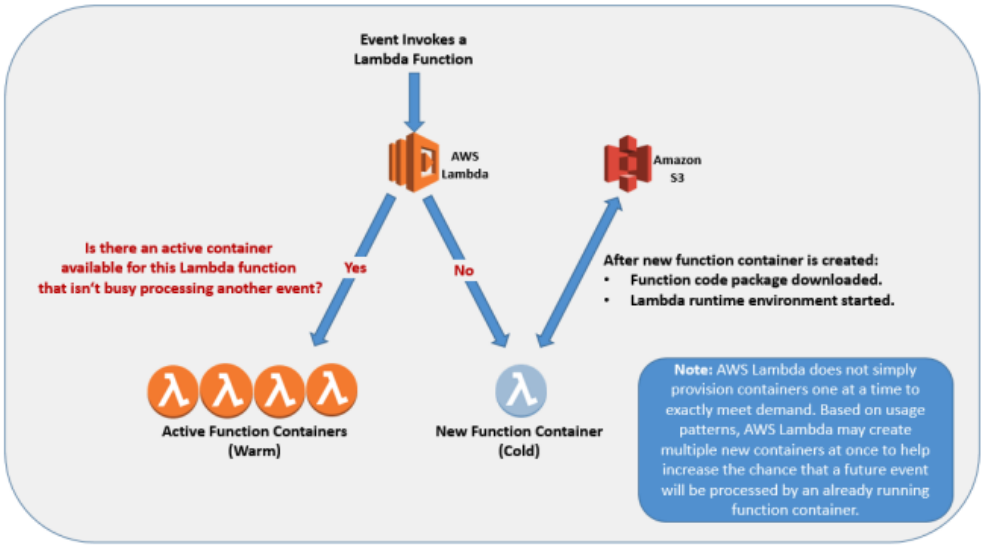

Abbildung 2.3: AWS Lambda Funktionsweise[27]

2.6.1 *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 erst die Init-Phase ausgeführt werden, wird dies als Coldstart bezeichnet. Aufgrund des Mehraufwandes, erweist sich ein Coldstart als deutlich langsamer als ein Warmstart. Abbildung 2.3 verdeutlicht den Prozess.

2.6.2 *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 1000 nebenläufige Funktionen pro AWS Region, dieser Wert lässt sich allerdings nach Absprache mit AWS erhöhen.

Durch die Nebenläufigkeit wird die Stärke von Lambda Funktionen bei der Skalierung deutlich. Gibt es einen Anfragesturm, werden automatisch neue Instanzen aufgesetzt, die diese Bearbeiten können. Bereits initialisierte Umgebungen werden wiederverwendet. Lässt die Nachfrage nach, werden Umgebungen automatisch wieder heruntergefahren.

2.7 RELATED WORK

Serverless ist allgemein ein noch junges Thema, das erst mit der Einführung von AWS Lambda im Jahre 2014 viel Aufmerksamkeit erfuhr. Es gibt bereits einige Studien zu der Performance von Serverless-Anwendungen, jedoch wurde in den meisten Studien nur Micro-Benchmarks betrachtet, d.h Tests, die nur einen Aspekt untersuchen, bspw. die CPU-Floating-Point Performance und keine realistischen Applikationen untersuchen[26]. Vergleiche mit der Performance von Container-Anwendungen sind nur sehr wenige zu finden.

McGrath und Brenner untersuchten die Performanz von Serverless-Funktionen durch mehrere verschiedene Tests. 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. 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 Skalierverhalten für diverse Cloud-Anbieter; bei AWS Lambda wurde ein weitestgehend linearer Anstieg festgestellt. Des Weiteren werden für zukünftige Forschung noch andere die Performanz 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 CPU-Allokation (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 laufendem Docker-Containers[13]. 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 auf die der Last verteilt wurde, während bei Beanstalk alle Anfragen von einer einzigen Instanz bearbeitet wurde, obwohl alle Einstellungen getroffen wurden, damit Beanstalk so schnell wie möglich hochskalieren kann. Hier zeigen sich deutlich die automatischen Skalierungsvorteile einer FaaS Applikation. Während es bei Beanstalk 20 verschiedene Einstellungsmöglichkeiten gäbe, übernimmt 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 10 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[29]. Die Anwendung

wurde als Monolith und als Microservices mit und ohne AWS Lambda betrieben. Dabei zeigte sich, dass die Kosten für den Betrieb bei einer auf Lambda basierten Microservices Architektur deutlich (mehr als 70 Prozent) geringer ausfallen können, als bei einem Monolith oder wenn die Microservices von den Entwicklern selbst gemanaged werden. 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 Performanz untersucht. 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 Backends miteinander verglichen werden. Die Ergebnisse der Arbeit zeigten dennoch, dass ECS mit Fargate deutlich besser performte als AWS Lambda und sogar als ECS mit Elastic Compute Cloud (EC2). 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 der von AWS Lambda[10]. 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 die 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 Performanz von Containern und Serverless-Funktionen anhand mehrerer Konfigurationen und mit verschiedenen Performance-Tests 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.

3 KONZEPTION

In diesem Kapitel wird die grundlegende Konzeption der Arbeit vorgestellt.

3.1 METHODISCHES VORGEHEN

Es soll eine Beispielanwendung sowohl Containerisiert als auch Serverless entwickelt werden. Die Konzeption dieser beiden Anwendungen wird in den folgenden Abschnitten genauer beschrieben. Im Anschluss werden Performance Tests gegen beide 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. Dazu wurden Annahmen (H1 bis H4) formuliert:

RQ1 Wie viel Load kann eine einzige Container-Instanz tragen und wie vielen nebenläufigen Lambda-Funktionen entspricht das?

RQ2 Wie ist die Performance bei normalem Load oder bei Spike Load?
H1: Die Performance einer Lambda Funktion stagniert unter Spike Load stark, während sie bei einem Container gleich bleibt.

RQ3 Welchen Einfluss hat die Nutzung größerer CPU / RAM Werte?
H2: Ein Container kann bei einer Verdopplung doppelt so viele Anfragen verarbeiten.

RQ4 Welchen Einfluss hat die Nutzung mehrerer Container Instanzen auf die Performance?
H3: Die Performance steigt linear mit jedem neuen Container an

RQ5 Welchen Einfluss hat die ein Use-Case, der mehreren Endpunkten anfragt
H4: Bei Lambda großer Einfluss, da jede Funktion einzeln einen Coldstart machen muss

Das methodische Vorgehen für den Aufbau der Test-Umgebung basiert dabei grundlegend auf den von Papadopoulus et. al. vorgestellten methodologischen Prinzipien für reproduzierbare Performanz-Evaluation im Cloud Computing Umfeld, die im folgenden beschrieben werden[23]:

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 Performanz (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 und Zeit nicht genauer betrachtet, daher hier der Hinweis, dass die Ergebnisse dieser Arbeit eventuell nicht signifikant sind.

P7 Einheiten (Measurement units): Für alle Messungen die zugehörige Einheit angeben

P8 Kosten (Cost): Kostenmodell, Ressourcennutzung und abgerechnete Kosten angeben.

Das Prinzip P4 wird erfüllt durch die Nutzung eines öffentlichen GitHub Repositories unter `https://github.com/rolule/ba`. Dort finden sich die Skripte die zur Generierung der Tests verwendet wurden und die Output-Artefakte der Tests.

Im Anschluss an die Performance-Tests wird noch auf Möglichkeiten der Skalierung und die Kosten der beiden Technologien eingegangen.

3.2 KONZEPTION DER TESTANWENDUNG

Als Testanwendung wird ein einfaches REST-Backend am Beispiel eines Notiz-Applikation entwickelt, wie es in der Praxis häufig verwendet wird. 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[22]. 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 verwendet[38]. 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, 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.

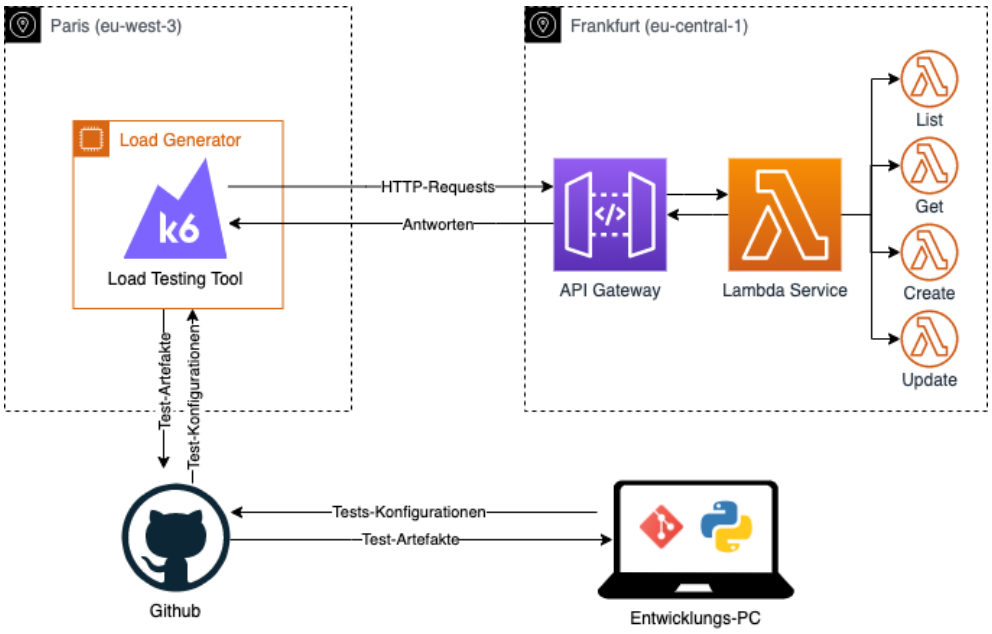

Abbildung 3.1: Testing-Architektur der Serverless Notiz-Anwendung

Der von Serverless bzw. CloudFormation erstellte Anwendungs-Stack beinhaltet ein Amazon API-Gateway, das über die Routen der REST-Schnittstelle

verfügt. Trifft eine HTTP-Anfrage 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 vom Benutzer im HTTP-Request übergebenen Parametern aufruft. Nach Beendigung der Funktion wird ihr Rückgabewert über das API-Gateway zurück an den Benutzer 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 Amazon S3-Bucket hochgeladen und bei Bedarf vom Lambda-Service angefordert.

Durch die Nutzung des Frameworks wird die Erstellung der Anwendung vereinfacht. Die Konfiguration des API-Gateways und des S3-Buckets werden automatisch erledigt. Zusätzlich minimiert das Framework den geschriebenen JavaScript Code mittels Webpack und erhöht so die Performance der Lambda-Funktion.

3.2.2 *Container*

Für die Entwicklung der containerisierten Anwendung wird das Express Framework verwendet, das mit über 51.000 Github-Stars ein weit verbreitetes Web-Frameworks für Node.js ist[36]. Die Anwendung wird unter Nutzung der Docker-Engine 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 der AWS Elastic Container Service (ECS) Plattform, einem von AWS bereitgestelltem Orchestrations-Tool. Es wird der Starttyp Fargate benutzt, welcher die Erstellung eines Server-Clusters automatisch durchführt 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.

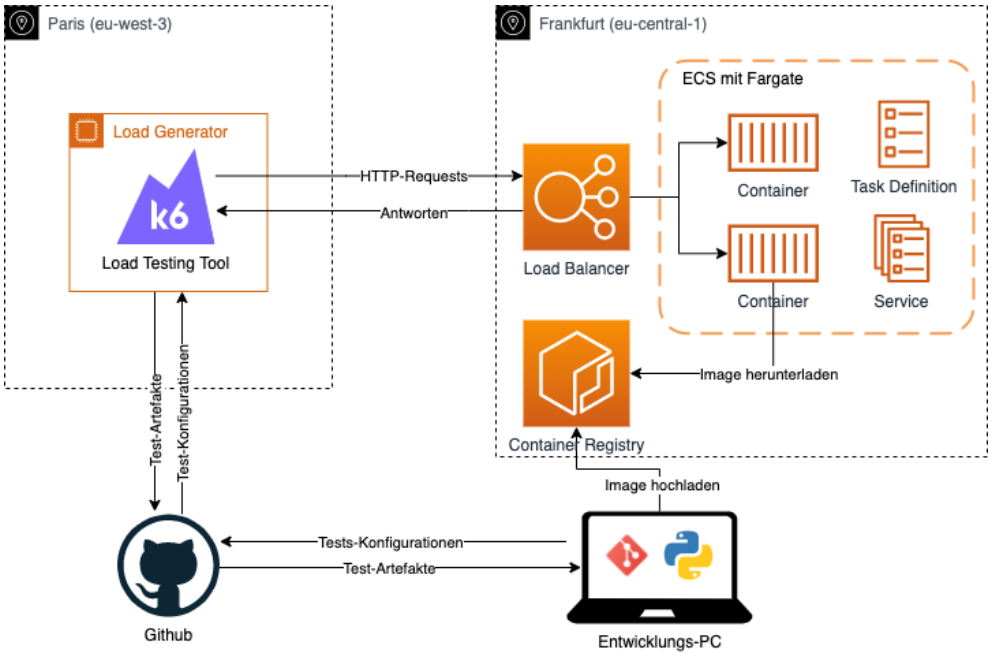

Abbildung 3.2: Testing-Architektur der Container Notiz-Anwendung

Bei der Konfiguration des Container-Clusters muss für Fargate eine Task-Definition erstellt werden. Diese beinhaltet 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. AWS ECS erlaubt nur bestimmte Paare von CPU und Speicher-Einstellungen für einen Container. Für die Tests mit 128MB und 256MB Speicher wird das Einstellungs-Paar mit 512MB Speicher und 0.25vCPU 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 Speicher und 0.5vCPU verwendet, der ein hartes Limit von 512MB Speicher zugewiesen bekommt. Ein vCPU (virtual CPU) beschreibt laut AWS hierbei einen Thread eines CPU-Kerns[7]. Die Task-Definitionen sind im Anhang A zu finden.

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 [19] verwendet. Durch eine optimierte CPU Auslastung ermöglicht es ohne die Notwendigkeit einer verteilen Ausführung, etliche virtuelle Benutzer zum Testen einer Anwendung erstellen zu lassen[25]. Die Benutzer arbeiten parallel die in Sektion 3.3.4 vorgestellten Use-Cases ab, bei denen verschiedene Requests an die REST-APIs der Services geschickt werden.

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[25]. Da die verwendete t3.xlarge Instanz 16 Gigabyte RAM beinhaltet, sollten theoretisch mehr als 3200 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.

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.

3.3.2 *Metriken*

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

1. VUS (virtual users): Die Anzahl der virtuellen Benutzer zum aktuellen Zeitpunkt
2. Antwortzeit (Response Time): Die Zeit vom Abschicken eines Requests bis zum Erhalt der Antwort in Millisekunden (ms). Dabei wird Aufwand für eventuelle DNS Lookups nicht mit einberechnet.
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[11].
   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[12].
4. Request Rate: Die durchschnittliche Anzahl der Requests pro Sekunde (req / s)

Für Analysen und Vergleiche mehrerer Test-Ausführungen untereinander, werden die Ergebnisse jedes k6-Tests als CSV-Text-Datei exportiert. Im Anschluss können diese Dateien mit dem Data-Science Framework Pandas[37] 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.

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[33].

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[34].

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 evaluiert. Die nach einem Request 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[22]. Um möglichst realitätsnahe Ergebnisse zu erzielen, wird davon ausgegangen, dass der Benutzer etwa eine Sekunden 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 -> 1s

- Use-Case B: Notiz bearbeiten
  Erst alle Notizen, dann eine einzelne Notiz abrufen und bearbeiten.
  1. GET /notes -> 1s
  2. GET /notes/1 -> 3s
  3. PUT /notes/1 -> 1s

- Use-Case C: Notiz falsch erstellt
  Der Benutzer ruft alle Notizen ab. Er entscheidet sich eine Notiz 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 -> 1s + 3s = 4s
  2. POST /notes -> 1s
  3. GET /notes/1 -> 3s
  4. PUT /notes/1 -> 1s

3.3.5 *Ablauf der Tests*

Alle Tests werden in Abhängigkeit von der Konfiguration der Systeme bezüglich des Speichers oder der Anzahl der Container-Instanzen durchgeführt. Nach Molyneaux sollte für jede Konfiguration zunächst ein Pipe-Clean Test durchgeführt werden[22]. 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 AWS Lambda ein von Natur automatisch skalierendes System ist, während ein Container alleine nicht skalieren kann. Um eine zu hohe Auslastung des Containers während der Last-Tests zu vermeiden, wird der Stress-Test durchgeführt, um das VU-Limit für die aktuelle Container-Konfiguration zu erkennen. Im Anschluss wird mit diesem ermittelten VU-Limit ein Last-Test gegen beide System 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. Der Last-Test mit dem schnellen Anstieg der Benutzer wird im folgenden als Spike-Test bezeichnet, um den Unterschied der beiden Tests deutlicher zu machen.

Zwischen jedem Lambda-Test wurde eine Pause von mindestens 10 Minuten eingelegt, um warmen Funktionen Zeit zu geben vom Garbage-Collector des Lambda-Services beendet zu werden.

4 ANALYSE

In diesem Kapitel werden die in Kapitel 4 vorgestellten Systeme analysiert.

4.1 128MB KONFIGURATIONEN

Zunächst werden alle Tests gegen die 128MB Konfigurationen ausgeführt. In den 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. Der Test wurde mit einer Dauer von zehn Minuten mit einem aktiven Benutzer für jeden Use-Case durchgeführt. Jeder Test wurde drei mal ausgeführt und die Metriken über die Gesamtmenge der Daten aggregiert. Bei den Requests wurde die Summe gebildet, bei den durchschnittlichen Requests pro Sekunde das arithmetische Mittel.

	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 aufgrund des Tests-Tools mit der Englischen 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. Während die maximale Antwortzeit bei ersterer nur etwa 10,7% über dem Median liegt, ist sie bei der Lambda-Funktion etwa 630% über ihrem Median. Dass die Lambda-Anwendung eine überaus große Varianz in ihren Antwortzeiten aufweist, wird auch in ihrer Standardabweichung von bis zu 72ms deutlich, währende sie bei der Container Anwendung in allen drei Use-Cases unter 1ms liegt. Interessanterweise liegt die Antwortzeit der Lambda-

Anwendung im Median in allen drei Use-Cases bei ähnlichen Werten um ca. 100ms. Andererseits nimmt die Standardabweichung und das P(95) Quantil der Antwortzeit bei Use-Case B im Vergleich zu Use-Case A deutlich zu. Und auch bei Use-Case C ist ein deutlicher Anstieg der Werte zu beobachten. Vermutlich ist dies auf die größere Anzahl an angefragten Endpunkten zurückzuführen, da dadurch mehr Coldstarts durchgeführt werden müssen. Auch bei der Container-Anwendung ist keine Veränderung der Median-Antwortzeit um ca. 60ms festzustellen. Die Standardabweichung der Antwortzeit erhöhte sich nur leicht von 0,42 auf 0,66. 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 liegt an der größeren Think-Time von 3 Sekunden, wie sie bei den Use-Cases mit Benutzer-Eingaben zum Einsatz kommt.

4.1.2 *Stress-Tests*

Um RQ1 zu beantworten, muss zunächst die maximale Auslastung eines einzelnen Containers ermittelt werden. Dazu wurden mehrere Stress-Tests durchgeführt. Ein Stress-Test dient dazu, die Grenzen des SUTs herauszufinden. Da Lambda ein von Natur aus automatisch horizontal skalierendes System ist, werden die Test zunächst für die Container-Anwendung durchgeführt und im Anschluss die Lambda-Anwendung mit der gleichen Test-Konfiguration getestet. Die Anzahl der VUs wird bei den Stress-Tests wie von Molyneaux[22] empfohlen immer stufenweise erhöht. Das bedeutet, dass nach jedem Ramp-Up (ein linearer Anstieg) 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 nach den vollen 600 Benutzern wird hier nicht eingelegt, da das Skalierungsverhalten nicht untersucht wird.

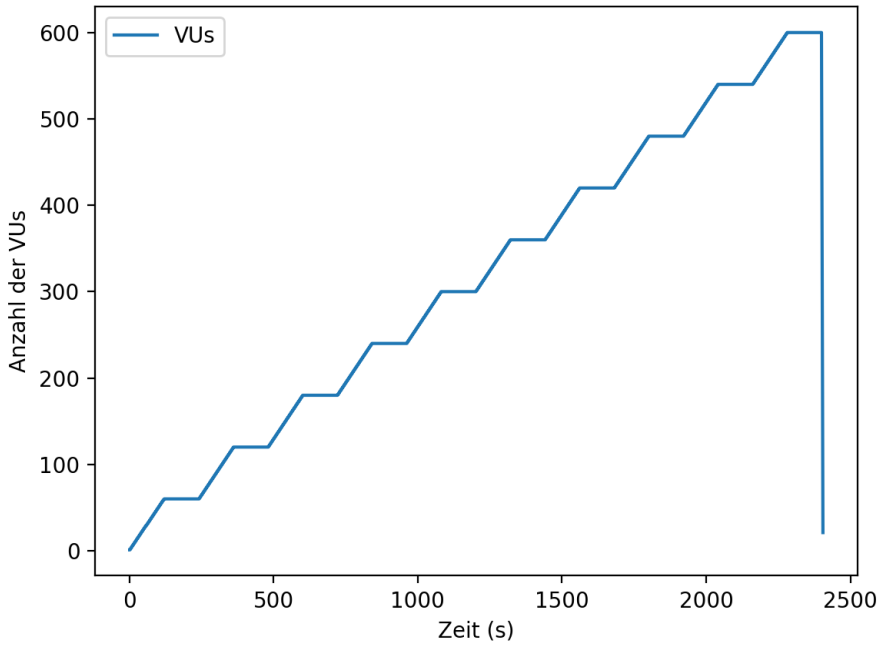

Abbildung 4.2: Beispiel-Anstieg der gleichzeitigen VUs für einen Stress Test

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 getesteten Use-Cases und die Stress-Test Konfiguration.

	Use-Case A			Use-Case B			Use-Case C		
Metrik \ Max. VUs	**200**	**600**	**1000**	**200**	**600**	**1000**	**200**	**600**	**1000**
Requests	231548	712394	1128646	142374	438345	730188	106660	328172	547148
Durchschn. Requests / s	96.43	296.68	470.00	59.19	182.24	303.58	44.27	136.20	227.08
Durchschnittliche Antwortzeit (ms)	60.21	60.55	116.18	60.28	59.95	61.65	60.07	19.36	60.26
Minimale Antwortzeit (ms)	57.70	29.08*	12.45*	57.80	57.66	11.73*	8.59	60.05	11.52*
Maximale Antwortzeit (ms)	364.93	492.27	935.66	276.58	277.11	1033.86	274.96	275.01	408.75
Median Antwortzeit (ms)	59.87	59.66	60.97	59.92	59.70	59.81	59.95	59.83	59.74
P(90) Quantil d. Antwortzeit (ms)	60.79	60.74	254.95	60.89	60.61	61.49	60.82	60.75	60.70
P(95) Quantil d. Antwortzeit (ms)	61.59	63.01	332.07	61.81	60.99	69.47	61.14	61.19	61.24
Standardabweichung d. Antwortzeit	3.13	6.45	93.36	2.79	2.76	12.07	1.81	2.21	4.68
Variationskoeffizient	0.05	0.11	0.8	0.05	0.05	0.2	0.03	0.04	0.08
Maximale CPU-Auslastung	31.78	84.93	100%	21.63	54.82	93.14	8.96	48.8	71.34
Fehler	0	3x502	31x502	0	0	2x502	1x502	1x502	1x502

Abbildung 4.3: Fargate 128MB Stress-Test Vergleich

Der erste Test wurde mit bis zu 200 Benutzern durchgeführt. Dabei konnte jedoch nur bis zu 32% der CPU-Auslastung des Container-Clusters erreicht werden; In Use-Case B waren es sogar nur fast 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, bspw. bei Use-Case A von 60,46ms auf 364,93ms, deutlich erhöht. Dies ist allerdings nur vereinzelt der Fall, da immer noch 95% aller Anfragen innerhalb von ca. 62ms beantwortet werden. Der Variationskoeffizient stieg 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 zwar bis zu 84,93 Prozent CPU-Auslastung erreicht werden und die maximale Antwortzeit stieg erneut an, auf fast 500ms. Trotz dessen konnten von dem einzelnen Container 95% der Anfragen innerhalb von knapp 63ms beantwortet werden. Problematisch zeigten sich hier dennoch drei Requests, die vom Server nicht mehr beantwortet werden konnten und den 502 Fehlercode zurücksendeten. Dies entspricht jedoch nur 0,00042% aller in diesem Stress-Test durchgeführten Requests.

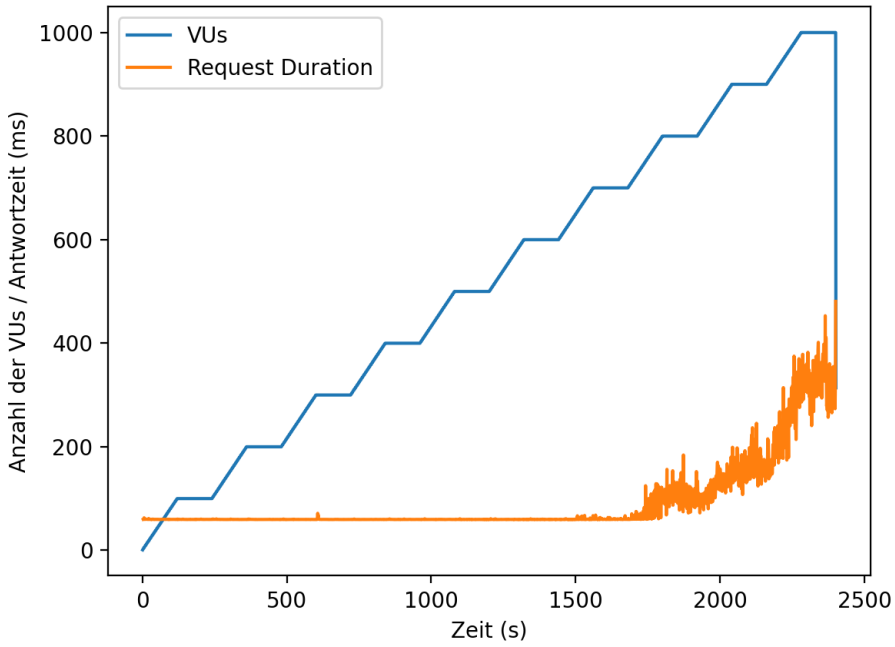

Abbildung 4.4: 128MB Container Stress-Test 1000VUs Use-Case A

Der dritte Stress-Test für den 128MB Container wurde mit bis zu 1000 VUs durchgeführt. Abbildung 4.4 zeigt den zeitlichen Verlauf dieses 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 knapp unter 100% die bei dieser Nutzerzahl gemessen wurde. In den 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 leicht herunter. Während des Tests wurden trotz einer CPU-Auslastung von teilweise 100% nur 31 HTTP 502 Fehlercodes vom Server ausgelöst, was einer

Quote von 0,0027% entspricht. Die Antwortzeit konnte im Median weiterhin bei etwa 60ms gehalten werden; die durchschnittliche Antwortzeit stieg mit 116,8ms auf fast das doppelte des Grundwertes an. Bei Use-Case B lag die erreichte Prozessor-Auslastung bei 93,14%. So wurden keine deutlichen Abweichungen ausgelöst. Der Variationskoeffizient stieg mit 0,2 leicht an, liegt aber immer noch deutlich unter dem Wert des Pipe-Clean Tests der 128MB Lambda Funktion. Dieser Wert könnte allerdings auch durch die von den aufgetretenen Fehler verursachten geringen Antwortzeiten gestiegen sein. Diese sind in allen Tabellen-Abbildungen mit einem Stern (*) gekennzeichnet. Um die maximale Auslastung des Containers für die anderen beiden Use-Cases zu erreichen, wurden noch weitere Stress-Tests mit bis zu 2.000 VUs durchgeführt. Bei Use-Case B konnte das Limit bei ca. 1.100 Benutzern erreicht werden. Für Use-Case C liegt es bei ca. 1.500 Benutzern.

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. 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 von 41,74ms auf 16.92ms gesunken (Use-Case A). Für Use-Case A lässt sich erkennen, dass die Standardabweichung bei 200 und 600 VUs noch um die 19 beträgt, während bei 1000 Usern auf fast 55 ansteigt. Dies könnte allerdings an den extrem langen maximalen Antwortzeiten von bis zu 29.037ms liegen, die durch die Fehler verursacht werden. Die großen Fehler-Latenzen stehen im Gegensatz zu der Container Anwendung, bei der die maximale Antwortzeit immer noch knapp unter einer Sekunde lag. Auffallend ist, dass Fehler mit HTTP Statuscode 500 etwa 10s Antwortzeit aufweisen, während Fehler mit HTTP Statuscode 504 die fast 30s lange Latenz erzeugen. Die bei der Container-Anwendung erzeugten HTTP 502 Fehlercodes, führen stattdessen zu Antwortzeiten von 12ms bis zu 260ms.

	Use-Case A			Use-Case B			Use-Case C		
Metrik \ Max. VUs	**200**	**600**	**1000**	**200**	**600**	**1000**	**200**	**600**	**1000**
Requests	225074	694547	1159820	139893	430878	718971	105180	323960	540256
Durchschn. Requests / s	93.73	289.24	483.01	58.16	179.12	298.89	43.65	134.46	224.22
Durchschnittliche Antwortzeit (ms)	90.36	87.64	86.09	91.36	89.94	88.19	91.84	90.09	89.57
Minimale Antwortzeit (ms)	71.59	70.59	70.30	71.56	49.48*	70.56	72.47	71.77	27.83*
Maximale Antwortzeit (ms)	742.47	1171.70	29037.43*	689.92	6708.68	1102.13	636.98	29018.74*	1000.73
Median Antwortzeit (ms)	83.65	79.04	77.89	86.43	82.68	79.98	85.60	82.09	83.27
P(90) Quantil d. Antwortzeit (ms)	112.18	104.50	100.62	112.30	110.52	107.02	111.83	108.95	109.16
P(95) Quantil d. Antwortzeit (ms)	121.23	117.71	114.44	122.23	120.75	118.10	123.21	120.47	119.60
Standardabweichung d. Antwortzeit	19.69	19.22	54.70	20.81	23.22	19.20	21.12	93.66	19.78
Variationskoeffizient	0.22	0.22	0.64	0.23	0.26	0.22	0.23	1.04	0.22
Maximale Funktions-Dauer (ms)	576.02	562.88	514.99	624.16	574.59	861.59	508.72	555.66	448.42
Maximale Cold-Start Dauer (ms)	256	281	300	338	264	274	260	268	275
Maximale Nebenläufigkeit	26	59	93	32	61	86	30	56	74
Fehler	0	0	6x500 3x504	0	1x500	0	0	2x500 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 83.65ms, während sie bei bis zu 1000 VUs auf 77.89ms sank. Die Werte für Use-Case B scheinen einem ähnlichen Prinzip zu unterliegen. Es ist jedoch aktuell nicht bekannt, warum dies der Fall ist. Vermutlich provisioniert AWS die Funktionen schneller je mehr Anfragen sie erhalten. Dies würde auch den Verlauf aus Abbildung 4.6 erklären, bei dem die drei Use-Cases für den Test mit bis zu 1.000 Benutzern abgebildet sind.

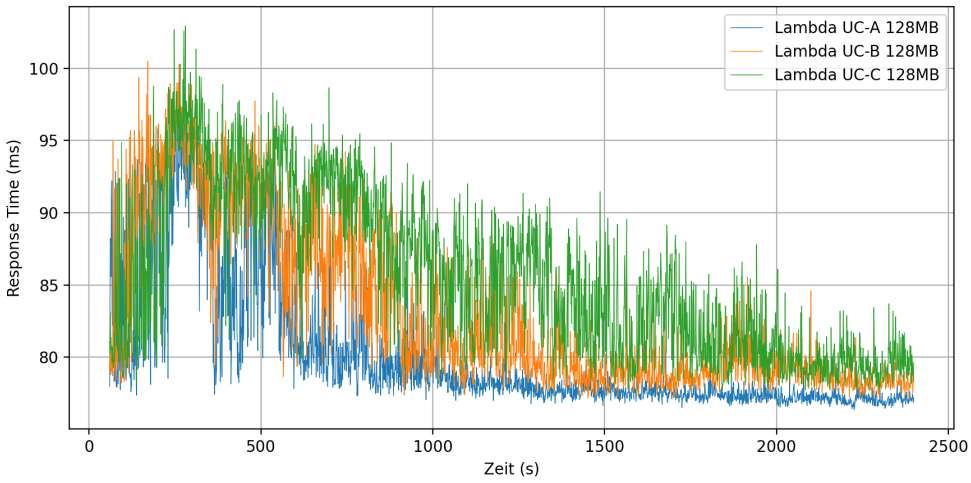

Abbildung 4.6: Lambda 128MB Stress-Test Verlauf Vergleich

Aus Gründen der Übersichtlichkeit wurde der Graph erst ab einer Minute dargestellt. Es ist zu erkennen, dass die Antwortzeit bei allen Use-Cases am Anfang ansteigt und nach ca. 300 Sekunden ihren Höhepunkt bei in etwa 95ms bis 100ms erreicht. Danach flachen alle Kurven langsam ab, bis sie ca. 80ms erreichen. Dabei scheint die Kurve von Use-Case A jedoch schneller als die von Use-Case B abzuflachen. Und auch dieser befindet sich schneller auf einem niedrigeren Niveau als der dritte Use-Case.

Zwischen den Use-Cases wird ebenfalls ein Unterschied in der Anzahl der nebenläufigen (concurrent) Lambda-Funktionen deutlich. Während bei 200 und 600 VUs für Use-Case B noch mehr Funktionen bereitgestellt wurden als für Use-Case A, zeigt sich für 1000 VUs ein umgekehrtes Bild. In diesem Fall, übertrifft der erste Use-Case den zweiten um 7 nebenläufige Funktionen. Dies ist interessant gegenüber RQ5, da es der Hypothese widerspricht, dass bei einer einzelnen angefragten Funktion weniger Coldstarts durchgeführt werden müssen als bei mehreren Funktionen (H5). Grund dafür ist vermutlich jedoch die geringere Anzahl der Requests pro Sekunde bei Use-Case B.

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.

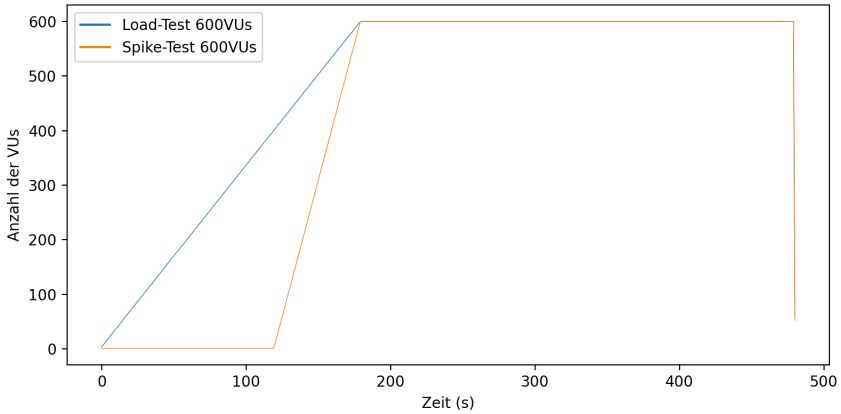

Abbildung 4.7: Load- und Spike-Test Vergleich

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 erreichen innerhalb von drei Minuten die maximale Grenze. Für den Spike-Test soll nach einer zwei-minütigen Warte-Phase, in der keine Benutzer auftreten, ein schneller Anstieg innerhalb von einer Minute auf die 600 Virtuelle Benutzer erfolgen. 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 Skalierungsverhaltens der Lambda-Funktionen 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.

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.

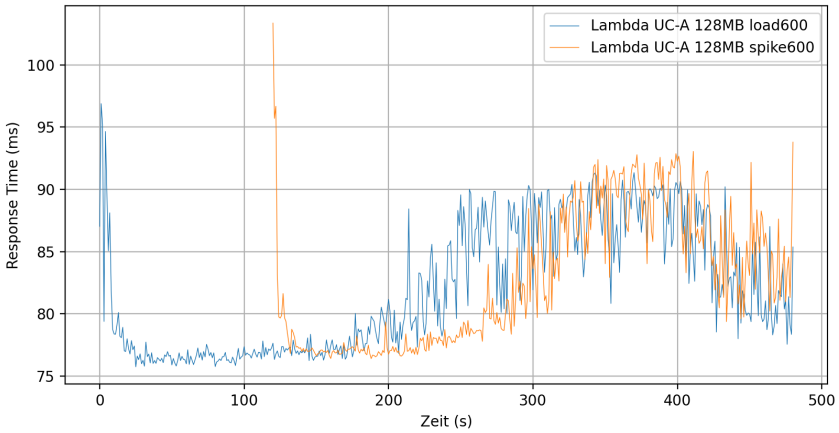

Abbildung 4.8: Lambda Use-Case A Load- und Spike-Test Vergleich

Abbildung 4.8 zeigt ein Vergleich zweier Load- und Stress-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 75ms - 80ms einpendelt. Ab 180 Sekunden (3 Minuten) ist 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 allerdings 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; die Antwortzeit liegt mit ca. 93ms nur leicht über der des Load-Tests.

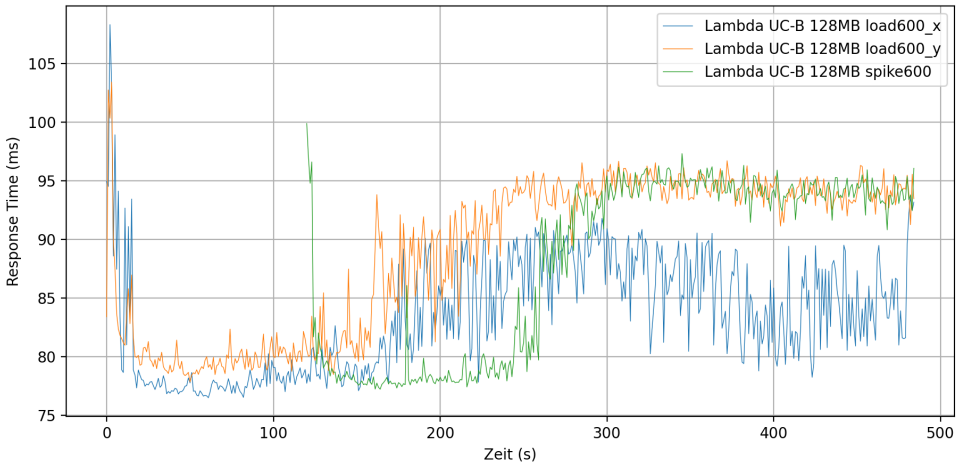

Abbildung 4.9: Lambda Use-Case B Load- und Spike-Test Vergleich

Für Use-Case B (Abbildung 4.9) wurden zwei 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 Antwortzeit von ca 95ms, womit dessen Verlauf stärker dem zweiten Load Test ähnelt. Es wird deutlich, wie unterschiedlich die Verläufe eines identischen Tests bei der Lambda-Anwendung ausgeprägt sein können. Interessant ist außerdem, 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.

Da die Antwortzeiten des Spike-Test trotz des dreimal schnelleren Anstiegs dennoch äußerst nah bei denen der beiden Load-Tests liegen, lässt sich entgegen H1 annehmen, 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.

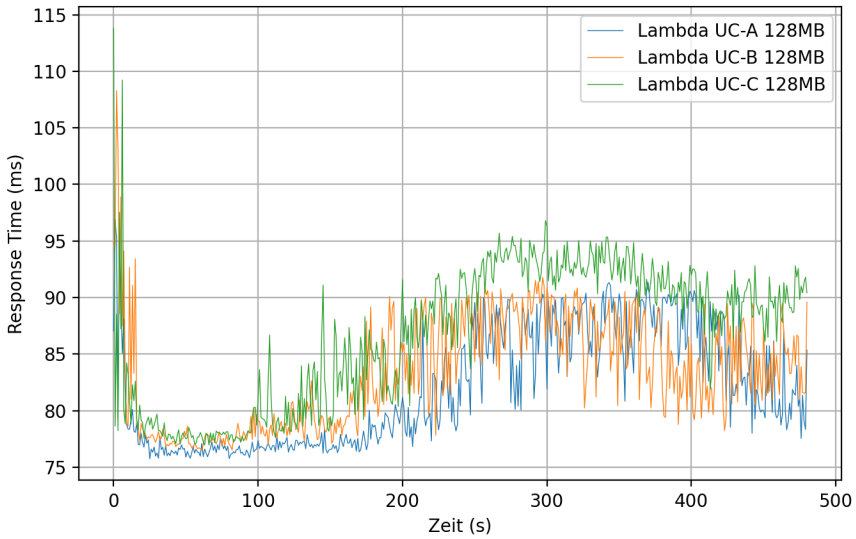

Abbildung 4.10: Vergleich der 600VU Load Tests für Lambda

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 aber eindeutig über der des ersten Use-Case, übertrifft diesen sogar gegen Ende des Vergleichs. Use-Case C mit den meisten Endpunkten weist auch die höchsten Antwortzeiten auf und erreicht mit über 95ms in etwa 5ms höhere Zeiten 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 mit ca. 0,01 auch kaum eine Veränderung. Abbildung 4.11 zeigt die Pipe-Clean Tests der unterschiedlich großen Konfigurationen der Lambda Funktion für die Use-Cases A und B.

	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 nur noch 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*

Unter Annahme von H3 wird davon ausgegangen, dass der Container 256MB Speicher und CPU-Größe doppelt so viele Anfragen verarbeiten kann wie der Container mit 128MB. Da der kleinere Container bis zu 700 Benutzer bedienen konnte, wurde im folgenden ein Stress-Test mit bis zu 1500 Benutzern durchgeführt, für den Fall dass er mehr als doppelt so viel verarbeiten kann. Trotz des doppelt so großen Speichers und vCPUs konnte jedoch keine Verbesserung der maximalen Benutzer für die Container-Instanz erreicht werden. Genau wie der 128MB Container begann die Antwortzeit des 256MB Container bei ca. 700 VUs stark anzusteigen. Die Hypothese H3 kann also für diesen Fall nicht bestätigt werden.

Die Stress-Tests für die Lambda Funktion wurden für diese Konfiguration nicht durchgeführt, da keine Verbesserung gegenüber der 128MB Variante zu vermutet wird.

Im Anschluss wurden die Last- bzw. Spike Tests durchgeführt, um diese mit der 128MB Variante zu vergleichen. Als VU-Limit wurde erneut 600 festgelegt. Dies ermöglicht einen einfachen Vergleich der beiden Varianten.

Auch hier konnte für die Container-Anwendung keine Unterschiede zu der kleineren Konfiguration festgestellt werden.

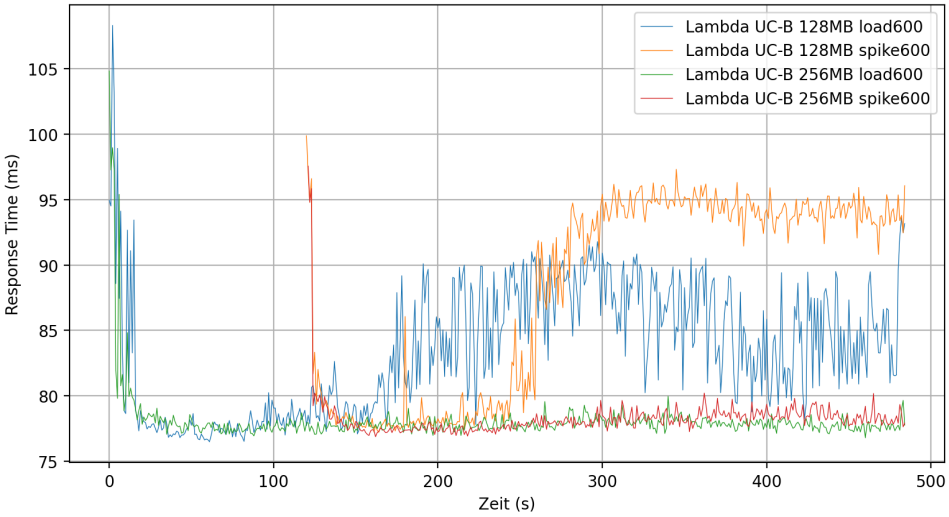

Abbildung 4.12: Lambda 128 und 256MB Vergleich der Load- und Spike-Tests

Für die Lambda-Funktionen 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 4.12 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 auch 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 der Antwortzeit im Median und der Coldstart-Dauer konnte auch hier keine Veränderung festgestellt werden.

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 1500 Benutzern, ergab für Use-Case A, dass die Antwortzeit des Containers ab einer VU-Anzahl von ca. 1400 Benutzern deutlich zunimmt. Mit Eintreten der 1400 Benutzer erreichte das Container Cluster ebenso eine CPU-Auslastung von 100 Prozent. Für Use-Case B musste ein weiterer Stress-Test durchgeführt werden, da bei 1500 Benutzern nur eine CPU-Auslastung von 58% erreicht werden konnte. Der zweite Stress-Test wurde mit bis zu 2400 Benutzern durchgeführt. Bei ca. 2350 VUs wurde die maximale Auslastung des Containers erreicht.

Im Anschluss an die Stress-Tests des Containers wurden diese mit den gleichen Konfigurationen gegen die Lambda-Anwendung durchgeführt.

	Use-Case A (stress1500)		**Use-Case B** (stress2500)	
Metrik \ Konfiguration	Cont. 512MB	Lamb. 512MB	Cont. 512MB	Lamb. 512MB
Requests	1150094	1136282	1495785	1494039
Durchschn. Requests / s	957.45	945.93	1240.80	1239.26
Durchschnittliche Antwortzeit (ms)	69.42	82.26	81.99	83.97
Minimale Antwortzeit (ms)	25.22*	70.28	9.10*	23.91*
Maximale Antwortzeit (ms)	18110.31	1122.4	2728.69	1164.28
Median Antwortzeit (ms)	59.81	76.50	68.38	77.48
P(90) Quantil d. Antwortzeit (ms)	66.60	94.44	117.39	96.57
P(95) Quantil d. Antwortzeit (ms)	76.88	97.90	140.89	100.23
Standardabweichung d. Antwortzeit	297.59	13.98	56.55	14.57
Variationskoeffizient	4.29	0.17	0.69	0.17
Maximale CPU-Auslastung	100%	-	100%	-
Maximale Funktions-Dauer (ms)	-	143.91	-	209.04
Maximale Cold-Start Dauer (ms)	-	222	-	309
Maximale Nebenläufigkeit	-	180	-	336
Fehler	2x502	0	76x502	1x504

Abbildung 4.13: Vergleich der Stress-Tests für 512MB Konfigurationen

Abbildung 4.13 zeigt das Ergebnis der Stress-Tests für die Use-Cases A und B. Der Container konnte dabei erneut, trotz Überlastung, geringfügig mehr Requests verarbeiten als die Lambda-Funktionen. Die Antwortzeit der Lambda-Funktionen pendelte sich, wie schon bei der 256MB Funktion, bei ca. 77ms ein. Auch der Variationskoeffizient lag erneut bei 0,17. Auffällig ist die hohe Fehleranzahl von 76 HTTP-502-Statuscodes, die bei der Container-Anwendung für Use-Case B auftraten. Die maximale Funktions-Dauer war im Vergleich mit den Stress-Tests der 256MB Lambda-Funktionen erneut deutlich niedriger.

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 den gleichen Stress-Tests unterzogen wie die einzelne 512MB Container-Instanz. Abbildung 4.14 zeigt das Ergebnis des Tests.

	Use-Case A (stress1500)		**Use-Case B** (stress2500)		**Use-Case C**	
Metrik \ Konfiguration	2x256MB	512MB	2x256MB	512MB	2x256MB	512MB
Requests	1157420	1150094	1495002	1495785		
Durchschn. Requests / s	963.57	957.45	1240.23	1240.80		
Durchschnittliche Antwortzeit (ms)	63.01	69.42	83.12	81.99		
Minimale Antwortzeit (ms)	16.71*	25.22*	9.11*	9.10*		
Maximale Antwortzeit (ms)	1068.32	18110.31	607.27	2728.69		
Median Antwortzeit (ms)	59.71	59.81	64.76	68.38		
P(90) Quantil d. Antwortzeit (ms)	63.95	66.60	133.88	117.39		
P(95) Quantil d. Antwortzeit (ms)	79.61	76.88	156.25	140.89		
Standardabweichung d. Antwortzeit	21.56	297.59	35.46	56.55		
Variationskoeffizient	0.34	4.29	0.43	0.69		
Maximale CPU-Auslastung (%)	98.66	100%	100%	100%		-
Maximale Funktions-Dauer (ms)	-	-	-	-	-	
Maximale Cold-Start Dauer (ms)	-	-	-	-	-	
Maximale Nebenläufigkeit	-	-	-	-	-	
Fehler	5x502	2x502	29x502	76x502		

Abbildung 4.14: Vergleich einer 512MB Instanz mit zwei 256MB Instanzen

In diesem Test wird kein Unterschied in der Performance einer einzelnen 512MB Instanz zu zwei 256MB Instanzen deutlich.

4.4 KOSTEN

In dieser Sektion sollen die Kostenmodelle der in dieser Arbeit getesteten Technologien untersucht und verglichen werden. Dabei wird sich erneut aufs Amazon AWS beschränkt. Es kann daher bei anderen Public-Cloud Anbietern deutliche Unterschiede geben.

4.4.1 *Container*

Wie schon in der vorherigen Sektion erwähnt gibt es viele verschiedene Wege, Container-Anwendungen in der Public Cloud zu betreiben. Für die Kosten-Schätzung werden deshalb nur der Betrieb der Anwendung mit AWS ECS Fargate-Starttyp und Amazon Elastic Kubernetes Service betrachtet.

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 pro vCPU pro Stunde 0,04656$. Die Formel 4.1 zeigt die Berechnung der Kosten für ein Fargate-Cluster.

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

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:

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

Dieser Wert konnte mit den in dieser Arbeit durchgeführten Tests bestätigt werden, da jeden Tag Kosten von 0,34$ für die Nutzung des Clusters anfielen. Bei 30 Tagen Laufzeit ergeben sich damit tatsächlich Kosten von 10,2$.

Für eine n-fache Anzahl an Tasks ergeben sich auch die n-fachen Kosten pro Monat. Variationen in den Anzahlen der Tasks aufgrund von (Auto-) Skalierung sind schwer vorauszusagen und werden daher nicht berechnet.

Zusätzlich zu den Fargate Kosten sind noch Gebühren für die Nutzung des Load-Balancers 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[24]. 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 Balancer:

- 128MB = 0,5GB und 0,25 vCPU = 10,22$ + 19,44$ (ELB) = 29,66$
- 256MB = 0,5GB und 0,25 vCPU = 10,22$ + 19,44$ (ELB) = 29,66$
- 512MB = 1 GB und 0,5 vCPU = 20,44$ + 19,44$ (ELB) = 39,88$
- 1024MB = 2 GB und 1 vCPU = 40,88$ + 19,44$ (ELB) = 60,32$

Kosten für Skalierung und die Nutzung des Load-Balancers sind hierbei noch zu ergänzen.

4.4.2 *Lambda*

Die Ausführung von Lambda-Funktionen wird nach der Ausführungszeit in Millisekunden abgerechnet. Dabei zählt die Zeit eines Cold-Starts mit in die Berechnung ein. Auch die konfigurierte Arbeitsspeichergröße fließt mit in das Ergebnis ein. Derzeit berechnet AWS in der Region Frankfurt (eu-central-1) Kosten von 0,0000166667 US-Dollar pro Gigabyte-Sekunde (GB-Sekunde) [18]. Zusätzlich müssen pro 1.000.000 Ausführungen 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.

Für die Ausführung der Lambda-Funktionen wird ein Event-Auslöser für REST-Anfragen benötigt. Dazu wurde in dieser Arbeit Amazon API Gateway 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 Aufruf an. Diese werden für jede API insgesamt gerechnet und gelten damit nicht für jede einzelne aufgerufene Lambda-Funktion. Es gibt Vergünstigungen ab 333 Millionen Aufrufen im Monat. Zuzügliche Preise für Caching werden hierbei nicht einberechnet. AWS bietet für das API Gateway kostenloses Nutzenkontingent von einer Millionen Aufrufen im Monat an[5]. Formel 4.2 zeigt die Berechnung der Gesamtkosten für eine Lambda-Funktion.

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

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:

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

Die nachfolgende Tabelle zeigt die Kosten der unterschiedlichen Funktionsgrößen pro 1.000.000 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/Fargate nur die tatsächlich aktive Zeit der Funktion berechnet wird, lässt sich grob abschätzen, wie viele Funktionsaufrufe zu dem gleichen Preis der Fargate Nutzung von einem Container möglich sind (siehe vorherige Sektion). Es wird 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.

$$Aufrufe = \frac{Kosten}{\frac{MedianMs}{1000ms} * \frac{FunktionsMb}{1024Mb} * 0,0000166667\$ + \frac{0,20\$+3,70\$}{1.000.000}} \tag{4.3}$$

Abbildung 4.15 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 4.15: Anzahl der Aufrufe von Lambda-Funktionen dem Monatspreis von Fargate entsprechend

Der jeweils linke Wert in den Tabellenzellen gibt dabei die maximale Anzahl der Aufrufe an, wenn jeder Funktionsaufruf bei 100ms liegt. Der rechte Wert ist der für eine Funktions-Dauer von 1ms und ist damit der maximal mögliche Wert, da Lambda immer mindestens eine Millisekunde abrechnet.

5 DISKUSSION DER ERGEBNISSE

In diesem Kapitel sollen die Ergebnisse aus der vorangegangenen Analyse diskutiert werden. Allgemein ist in den Experimenten deutlich geworden, dass die Performance der Container-Anwendung die der Lambda-Funktionen übertrifft. Die Antwortzeit lag im Median bei fast allen Tests bei ca. 60ms. Nur wenn die CPU-Auslastung eines Container an die 100 Prozent erreichte, stiegen die Response-Times an. Dagegen lagen die Werte der Lambda-Funktionen bspw. bei den Pipe-Clean Tests um den Wert 100ms. Allerdings konnte die Performance von Lambda mit einer größerer Anzahl an Requests verbessert werden. Bei den Load- und Stress-Tests konnte so mediane Antwortzeiten von 76ms erreicht werden. Die kleinste Request-Dauer überhaupt lag bei Lambda, Fehler ausgenommen, bei 69,99ms. Größere Ausreißer gab es bei beiden Technologien. Meist wurden diese durch Fehler verursacht. Während bei Fargate ausschließlich HTTP 502 (Bad Gateway) auftritt, 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 von Lambda sind allerdings mit einer Request-Dauer von bis zu 30 Sekunden als verheerender einzustufen.

Auch in der Varianz der Antwortzeit zeigen sich große Unterschiede der Technologien. Die Fargate Container zeigten schon in den Pipe-Clean Tests eine geringe Abweichung vom Mittelwert. Bei allen Konfigurationen und für alle Use-Cases betrug der Variationskoeffizient ca. 0,01. Bei den Lambda-Funktionen zeigte sich ein anderes Bild. Hier hatte sowohl die Funktionsgröße als auch der Use-Case einen Einfluss auf die Ergebnisse. In den Pipe-Clean Tests zeigte sich, dass Use-Case A einen Variationskoeffizient von ca. 0,23 bis 0,29 aufweist, während er bei Use-Case B zwischen 0,45 bis 0,52 schwankt. Während in den Pipe-Clean Tests keine Unterschiede zwischen den 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), während der Wert bei 256MB für den gleichen Test auf 0,17 schrumpfte. 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 dann aber keine Unterschiede mehr in der Abweichung vom Mittelwert.

5.1 BEANTWORTUNG DER FORSCHUNGSFRAGEN (RQ1 - RQ5)

Im folgenden sollen die in Kapitel 4 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. 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 69 Funktionen festgestellt. Bei der 512MB Konfiguration wurde eine Grenze von etwa 1.400 parallelen Benutzern ermittelt. Bei den Load-Tests mit 1200 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. Das führte darum ebenso zu einer geringeren Nebenläufigkeit mit 60 bzw. 103 bei Use-Case B. Forschungsfrage RQ1 lässt sich also nicht eindeutig beantworten, da sich kein erkennbares Muster abzeichnet und die Zahlen zwischen gleichen Experimenten deutlich abweichen können. 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. Es lässt sich aber für RQ1 sagen, 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. Es wurde vermutet, dass die Performance unter schnellem Anstieg bei beiden Systemen evtl. schlechter als die bei langsamen Anstieg wäre. Für die Performance des Containers konnte in den Experimenten keine Veränderung 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 denen des Load-Tests, bei Use-Case B jedoch deutlich über denen des Load-Tests lagen. Bei den 256MB und 512MB Funktionen konnte jedoch überhaupt kein Unterschied zwischen schnellem und langsamen Unterschied festgestellt werden. Bei größeren Nutzerzahlen könnte dieser Effekt jedoch größer ausfallen.

Die Forschungsfrage RQ3 befasste sich mit der Nutzung größerer RAM und CPU-Werten für sowohl die Container-Anwendung als auch die Lambda-Funktionen. Vermutet wurde, dass bei einer Verdopplung der Container-CPU auch doppelt so viele Anfragen verarbeitet werden können. Dies konnte teilweise bestätigt werden. Bei einem Wechsel von 128MB auf 256MB konnte keine Veränderung der maximalen Benutzerzahlen festgestellt werden. Beide Container konnten 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 jedoch 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 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 die Hypothese (H2) bestätigt 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 1400 VUs, also in etwa doppelt so viele Benutzer vor Steigen der Antwortzeit bedient werden. Daher kann die Hypothese teilweise bestätigt werden.

Für die Forschungsfrage RQ4 wurde die Performance von mehreren Container-Instanzen zu der von einzelnen Containern in Beziehung gestellt. Die Hypothese H3 war, dass die Anzahl der möglichen Requests mit jeder hinzugefügten Instanz linear ansteigt. Dies konnte mit Stress-Tests von zwei 256MB Instanzen und einer 512MB Instanz gezeigt werden. Die Hypothese kann also bestätigt werden.

Die letzte Hypothese war, dass sich ein Use-Case mit mehreren Endpunkten aufgrund der vermehrten Coldstarts negativ auf die Antwortzeiten der Lambda-Anwendung auswirkt (H4). Wie erwartet konnte bei der Container-Anwendung keine Veränderung festgestellt werden. Bei der Lambda-Funktion trat in den Pipe-Clean Tests kein Unterschied zwischen den Use-Cases auf. Anders sieht 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 sinken die Antwortzeiten der Funktionen mit mehreren Endpunkten langsamer auf das untere Niveau ab als die der mit einem Endpunkt. Und bei den Load-Tests erreichen die Use-Cases eine allgemein höhere Antwortzeit. Die Hypothese H4 kann also bestätigt werden.

5.2 KOSTEN

In Sektion 4.4 wurden die Kostenmodelle der beiden 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. Im Falle eines Containers bezahlt man also gleich viel wenn er mehr oder weniger ausgelastet ist. Vor allem zu Zeiten in denen die Auslastung gering ist, bspw. Nachts, ist die Nutzung eines Containers ein großer Nachteil. Der Vorteil von Lambda liegt darin, dass bei wenig Nutzung auch nur wenig zu zahlen ist. Bei einer hohen Auslastung kann ein Container allerdings 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 Auslastung der Anwendung ab. Wie in Sektion 4.4.2 gezeigt wurde, lässt sich ungefähr bestimmen, wie viele monatliche Requests mit Lambda möglich sind, bevor es teurer wird als ein einzelner Container. Andererseits hat ein Container eine Grenze, wie viele Requests er pro Sekunde verarbeiten kann, bevor ein weiterer Container hinzugezogen werden (skaliert werden) muss. Für jeden weiteren Container fallen dann auch weitere Kosten an, allerdings nur so lange er benötigt wird. Dafür müssen aber auch unter Umständen komplizierte Skalierungs-Konfigurationen erstellt werden, was bei Lambda nicht unbedingt notwendig ist. Welche Technologie das bessere Preis-Leistungs-Verhältnis aufweist, lässt sich also nur im Einzelfall bestimmen.

Insgesamt fielen für die Tests dieser Arbeit laut dem AWS Cost Explorer ca. 80 Euro für die Nutzung des API Gateways an. Für Lambda waren es nur ca. 4 Euro. Die Gesamtkosten der Serverless-Anwendung berufen sich also auf ungefähr 85 Euro. Im Gegensatz dazu wurden für die Container Anwendung nur ca. 15 Euro fällig, davon ca. fünf Euro für ECS mit Fargate und ca. zehn Euro für den Elastic Load Balancer. Zusätzlich müssen noch Kosten in Höhe von ca. 25 Euro für die EC2-Instanzen, die als Load-Generator fungierten, hinzugerechnet werden. Die Durchführung aller Tests in dieser Arbeit, veranschlagt also insgesamt Kosten von ca. 124 Euro.

5.3 IMPLIKATIONEN FÜR DIE PRAXIS

Durch die in dieser Arbeit durchgeführten Experimente wurde deutlich, dass die Performance eines Container-Backends auf AWS Fargate zwar der einer Serverless-Anwendung auf AWS Lambda in Bezug auf die Antwortzeit überlegen ist. Allerdings liegt der Unterschied der beiden Technologien nur im Bereich von einigen Millisekunden. Wird eine zeitkritische Anwendung benötigt, also z.B. eine Banking- oder Trading-Plattform, sollte dennoch eher die Verwendung einer Container-Architektur erwägt werden.

Lambda bietet ein stabiles Skalierungsverhalten, das auch bei Spitzenlasten gut funktioniert. Auch Container lassen sich skalieren, jedoch ist ein nicht zu vernachlässigender Aufwand der konkreten Implementierung des Skalierungsverhaltens zu beachten. Wird ECS ohne Fargate oder ein Kubernetes-Service verwendet, muss sich zusätzlich um die Konfiguration des Clusters gekümmert werde. All das wird von Lambda vollständig übernommen, was zu enormen Einsparungen an Projektkosten führen könnte.

Zusätzlich muss bei den Service-Kosten für den konkrete Anwendungsfall abgewogen werden, welche Technologie 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 wirtschaftlicher erweist. Bei nur kurzzeitig hoher Spitzenlast oder geringer Nutzung des Services ist in Bezug auf die Gebühren vermutlich die Nutzung der FaaS-Anwendung geeigneter, da sonst für nicht genutzte Kapazität des Containers bezahlt werden muss. Es ist also empfehlen, kontinuierlich 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 zu sparen.

5.4 AUSBLICK

Es gibt viele verschiedene Variablen die Einfluss auf die Performance von Serverless- und Containerisierten-Anwendungen nehmen können. Beispielsweise könnte das in dieser Arbeit genutzte 50ms Timeout variiert werden 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 die sowohl die Größen der Lambda-Funktionen als auch der Container-Instanzen noch auf mehrere Gigabyte erweitern. Container lassen sich auf viele Wege deployen und betreiben, z.B. auf einem selbst gemanagten EC2 Cluster, mit ECS, einem EKS Cluster oder Elastic Beanstalk. Dazu gehört ebenfalls das unter Umständen komplexe Skalierungsverhalten einer Container-Anwendung.

Da Container auf jedem Computer mit Unterstützung einer Container-Runtime laufen können, bieten sich vielfältige Wege an eine Container-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 manuelle Verteilen der Container auf Computer-Clustern zu vermeiden. Bei Kubernetes gibt es den Horizontal Pod Autoscaler (HPA), der die automatische Skalierung von Pods in Abhängigkeit der CPU-Auslastung ermöglicht[15]. Auch bei Nutzung des AWS Elastic Kubernetes Service (EKS) ist ein automatische Skalierung möglich[14]. Auch die Nutzung eines Vertical Pod Autoscaler (VPA) ist möglich, der automatisch die Ressourcen einzelner Container skaliert[28]. Es kann auch ein Cluster Autoscaler verwendet werden, um zusätzlich zu der Anzahl der Container auch die Anzahl der Kubernetes-Knoten hoch zu skalieren[9]. Auf die Performance-Tests all dieser Konfigurationen kann in dieser Arbeit unmöglich eingegangen werden, da es zu diesem Thema zu viele Möglichkeiten und auch 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 Performanz sinkt.

In dieser Arbeit wurde AWS ECS mit dem Fargate-Starttyp verwendet, um die Einstellung eines Clusters nicht vornehmen zu müssen. Fargate verwaltet die Knoten des Clusters automatisch. Der ordinäre Weg der Cluster-Erstellung wäre es, mehrere Virtuelle Maschinen zu starten, zu konfigurieren und jede dem Cluster zuzuweisen. Durch die Nutzung von Fargate kann die Konfiguration von Container-Clustern also erheblich vereinfacht werden.

ECS und Fargate bieten mit Auto-Scaling auch eine 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.

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

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. Nach Absprache mit AWS kann sich dieses Intervall auf eine Minute senken lassen. Dies ist allerdings, verglichen mit den Skalierungsmöglichkeiten von AWS Lambda, noch immer zu langsam für Anwendungen die hochverfügbar sein müssen.

Wichtig für eine schnelle Skalierung ist vor allem auch die Größe des Containers. Da bei jeder neu hinzugefügten Container-Instanz das Container-Image von der Registry heruntergeladen werden muss, ist es notwendig, die Image-Größe möglichst zu minimieren. Zum Einsatz kommende Techniken sind hierbei beispielsweise die Nutzung eines 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 Docker-Layer minimiert wird. In Zukunft können also noch viele Aspekte des Skalierungsverhaltens und der Optimierung von Container-Anwendungen mit Performance Tests untersucht werden. Auch das Skalierungsverhalten von Lambda-Funktionen kann mit Autoscaling und provisionierter Nebenläufigkeit variiert werden. Lambda ist ein von Natur aus horizontal skalierendes System, daher müssen vom Entwickler theoretisch keine Einstellungen vorgenommen werden, um eine hochverfügbare Anwendung zu schaffen. Um auch Cold-Starts möglichst zu vermeiden, lässt sich die gewollte Nebenläufigkeit allerdings auch schon vor einem Benutzer-Ansturm definieren (Provisioned Concurrency). Lambda startet dann 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.

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“[8], 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 Docker-Container, hat auch die Größe einer Lambda-Funktion Einfluss auf die Startzeit. Denn bei jedem Coldstart muss der Funktions-Code in den Lambda-Container geladen werden. Bei einigen Sprachen wie Java oder .NET macht ebenfalls die Konfiguration des Lambda-Arbeitsspeichers einen großen Unterschied[20].

Immer häufiger werden auch Mischformen von Container und Serverless. Beispielsweise bietet AWS seit Ende 2020 an, Container über AWS Lambda verfügbar zu machen[4]. Darüber 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. Es lassen sich aber für jede beliebige Anwendung Performance-Tests durchführen. In Zukunft könnte das für diese Arbeit genutzte Testing und Analyse-System daher ausgebaut werden, damit es für jede beliebige Anwendung nutzbar ist. Dies kann Organisationen dabei helfen, die Nutzung ihrer Technologien besser zu evaluieren. Es könnten ebenfalls noch weitere Metriken aus AWS CloudWatch, wie z.B. die tatsächlichen Funktions-Kosten, in die Analyse integriert werden und somit den Vergleich beider Technologien noch weiter verbessern. Auch andere Services wie AWS EKS oder Elastic Beanstalk könnten mit diesem System getestet werden.

6 FAZIT

Im Rahmen dieser Bachelor-Thesis wurden Serverless-Funktionen mit AWS Lambda und Container-Anwendungen am Beispiel von AWS ECS mit Fargate auf ihre Performance untersucht und Kosten und Skalierungsverhalten analysiert.

Teil II

APPENDIX

A

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-e
  "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-e
  "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-e
  "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": []
}
```

B

REPOSITORY

In diesem Kapitel wird das Code-Repository erklärt.

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 re:Invent 2017: Become a Serverless Black Belt: Optimizing Your Serverless Appli (SRV401) - YouTube*. URL: https://www.youtube.com/watch?v=oQFORsso2go&feature=emb_title (besucht am 07. 01. 2021).

[5] *Amazon API Gateway – Preise | API-Management | Amazon Web Services*. Amazon Web Services, Inc. URL: https://aws.amazon.com/de/api-gateway/pricing/ (besucht am 02. 02. 2021).

[6] Amazon AWS. *AWS Lambda - Developer Guide*. 2020. URL: https://docs.aws.amazon.com/lambda/latest/dg/lambda-dg.pdf.

[7] *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).

[8] Alex Casalboni. *alexcasalboni/aws-lambda-power-tuning*. original-date: 2017-03-27T15:18:12Z. 5. Jan. 2021. URL: https://github.com/alexcasalboni/aws-lambda-power-tuning (besucht am 06. 01. 2021).

[9] *Cluster Autoscaler - Amazon EKS*. URL: https://docs.aws.amazon.com/eks/latest/userguide/cluster-autoscaler.html (besucht am 02. 02. 2021).

[10] Alex DeBrie. *AWS API Performance Comparison: Serverless vs. Containers vs. API Gateway integration*. 20. Feb. 2019. URL: https://alexdebrie.com/posts/aws-api-performance-comparison/ (besucht am 22. 01. 2021).

[11] *Error 502 Bad Gateway: Wo liegt das Problem?* IONOS Digitalguide. URL: https://www.ionos.de/digitalguide/hosting/hosting-technik/was-bedeutet-502-bad-gateway-erklaerung-loesung/ (besucht am 10. 02. 2021).

[12] *HTTP 504 (Gateway Timeout): So lässt sich der 504-Error beheben*. IONOS Digitalguide. URL: https://www.ionos.de/digitalguide/hosting/hosting-technik/504-gateway-timeout-so-loesen-sie-das-problem/ (besucht am 10. 02. 2021).

[13] Scott Hendrickson, Stephen Sturdevant, Tyler Harter, Venkateshwaran Venkataramani, Andrea C Arpaci-Dusseau und Remzi H Arpaci-Dusseau. *Serverless Computation with OpenLambda*. 2017. URL: https://www.usenix.org/system/files/conference/hotcloud16/hotcloud16_hendrickson.pdf.

[14] *Horizontal Pod Autoscaler - Amazon EKS*. URL: https://docs.aws.amazon.com/eks/latest/userguide/horizontal-pod-autoscaler.html (besucht am 02. 02. 2021).

[15] *Horizontal Pod Autoscaler*. Kubernetes. URL: https://kubernetes.io/de/docs/tasks/run-application/horizontal-pod-autoscale/ (besucht am 02. 02. 2021).

[16] Ken Owens, Sarah Allen, Ben Browning, Lee Calcote, Amir Chaudhry, Doug Davis und Louis Fourie. *CNCF WG-Serverless Whitepaper v1.0*. 2018. URL: https://github.com/cncf/wg-serverless/blob/master/whitepapers/serverless-overview/cncf_serverless_whitepaper_v1.0.pdf (besucht am 08. 01. 2021).

[17] N. Kratzke und R. Peinl. "ClouNS - a Cloud-Native Application Reference Model for Enterprise Architects". In: *2016 IEEE 20th International Enterprise Distributed Object Computing Workshop (EDOCW)*. 2016 IEEE 20th International Enterprise Distributed Object Computing Workshop (EDOCW). ISSN: 2325-6605. Sep. 2016, S. 1–10. DOI: 10.1109/EDOCW.2016.7584353.

[18] *Lambda Preise – Amazon Web Services (AWS)*. Amazon Web Services, Inc. URL: https://aws.amazon.com/de/lambda/pricing/ (besucht am 07. 01. 2021).

[19] *Load testing for engineering teams | k6*. URL: https://k6.io (besucht am 13. 01. 2021).

[20] Nathan Malishev. *AWS Lambda Cold Start Language Comparisons, 2019 edition | by Nathan Malishev | Level Up Coding*. 4. Sep. 2019. URL: https://levelup.gitconnected.com/aws-lambda-cold-start-language-comparisons-2019-edition-%EF%B8%8F-1946d32a0244 (besucht am 03. 01. 2021).

[21] Peter Mell und Timothy Grance. *The NIST Definition of Cloud Computing*. Sep. 2011. URL: https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-145.pdf (besucht am 29. 12. 2020).

[22] Ian Molyneaux. *The Art of Application Performance Testing, 2nd Edition*. 2. Aufl. O'Reilly Media, Inc., 2014. ISBN: 978-1-4919-0054-3. URL: https://learning.oreilly.com/library/view/the-art-of/9781491900536/ (besucht am 20. 01. 2021).

[23] 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.

[24] *Preise für Elastic Load Balancing – Amazon Web Services*. Amazon Web Services, Inc. URL: https://aws.amazon.com/de/elasticloadbalancing/pricing/ (besucht am 09. 02. 2021).

[25] *Running large tests*. URL: https://k6.io/docs/testing-guides/running-large-tests (besucht am 21. 01. 2021).

[26] Joel Scheuner und Philipp Leitner. "Function-as-a-Service performance evaluation: A multivocal literature review". In: *Journal of Systems and Software* 170 (Dez. 2020), S. 110708. ISSN: 01641212. DOI: 10.1016/j.jss.2020.110708. URL: https://linkinghub.elsevier.com/retrieve/pii/S0164121220301527 (besucht am 18. 01. 2021).

[27] *Serverless Architectures with AWS Lambda*. Nov. 2017. URL: https://d1.awsstatic.com/whitepapers/serverless-architectures-with-aws-lambda.pdf (besucht am 04. 02. 2021).

[28] *Vertical Pod Autoscaler - Amazon EKS*. URL: https://docs.aws.amazon.com/eks/latest/userguide/vertical-pod-autoscaler.html (besucht am 02. 02. 2021).

[29] 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)*. 2016 16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid). Mai 2016, S. 179–182. DOI: 10.1109/CCGrid.2016.37.

[30] *Was ist Container-Orchestrierung?* URL: https://www.redhat.com/de/topics/containers/what-is-container-orchestration (besucht am 31. 12. 2020).

[31] *Was ist Kubernetes?* Kubernetes. URL: https://kubernetes.io/de/docs/concepts/overview/what-is-kubernetes/ (besucht am 31. 12. 2020).

[32] *Was ist Serverless?* Serverless Stack. 23. Dez. 2016. URL: https://serverless-stack.com/chapters/de/what-is-serverless.html (besucht am 31. 12. 2020).

[33] *What is Load Testing? How to create a Load Test in k6*. URL: https://k6.io/docs/test-types/load-testing (besucht am 10. 02. 2021).

[34] *What is Stress Testing? How to create a Stress Test in k6*. URL: https://k6.io/docs/test-types/stress-testing (besucht am 10. 02. 2021).

[35] *cncf/toc*. GitHub. URL: https://github.com/cncf/toc (besucht am 29. 12. 2020).

[36] *expressjs/express*. original-date: 2009-06-26T18:56:01Z. 13. Feb. 2021. URL: https://github.com/expressjs/express (besucht am 13. 02. 2021).

[37] *pandas - Python Data Analysis Library*. URL: https://pandas.pydata.org/ (besucht am 25. 01. 2021).

[38] *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).