

# **Hochschule Darmstadt**

– Fachbereich Informatik –

## **Performance Analyse einer skalierbaren Produktions-Webanwendung**

Abschlussarbeit zur Erlangung des akademischen Grades  
Bachelor of Science (B.Sc.)

vorgelegt von

**Laure Hontabat-Franky**

Matrikelnummer: 756904

Referent : Prof. Dr. Oliver Skroch  
Korreferent : Prof. Dr. Daniel Burda



## ERKLÄRUNG

---

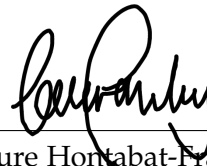
Ich versichere hiermit, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die im Literaturverzeichnis angegebenen Quellen benutzt habe.

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, 01.12.2021*



---

Laure Hontabat-Franky

## ZUSAMMENFASSUNG

---

In Rahmen dieser wissenschaftlichen Arbeit wird die Performance einer skalierbaren Produktions - Webanwendung analysiert, welche mit Amazon Web Services ([AWS](#)) App Runner betrieben wird. Der neue Cloud Service App Runner basiert auf Containerized Deployments, wobei die Bereitstellung einer Anwendung nun schnell und ohne Infrastrukturerfahrung erfolgen soll. Um App Runner zu testen wird ein kleines beispielhaftes REST-Backend entwickelt, welches dann mit App Runner in Betrieb genommen wird. Dazu werden Tests konzipiert die es ermöglichen die Leistung von App Runner zu analysieren. Nach der Konzeption der Tests werden diese für zwei unterschiedliche Konfigurationen von App Runner ausgeführt. Dabei werden auch die entstehenden Kosten der Anwendung festgehalten. Zum Schluss werden diese Ergebnisse analysiert, wobei sich herausstellte, dass der Skalierungsalgorithmus von App Runner die Leistung beeinflussen kann. Daher ist die Nutzung von App Runner für zeitkritische Anwendungen nicht geeignet. Indessen wurden Anwendungsfälle erarbeitet die eine optimale Nutzung von App Runner erzielen könnten. Die aufgestellten Kostenmodelle zeigten auf, dass die Kosten für App Runner zwar rapide zunehmen können, jedoch sehr von der Art und Belastung der Anwendung abhängen.

## ABSTRACT

---

In this scientific work, the performance of a scalable production web application is analyzed, which is operated with [AWS](#) App Runner. The new cloud service App Runner is based on a containerized deployment and promises a quick deployment without much experience with infrastructure. To test App Runner, a small example REST backend will be developed, which will then be put into operation with App Runner. Tests will be designed to analyze the performance of App Runner. After designing these tests, they will be executed for two different configurations of App Runner. The costs incurred by the application will also be recorded. Finally, these results were analyzed, showing that App Runner's scaling algorithm can affect its performance. Therefore, the use of App Runner is not suitable for time-critical applications. However, use cases were developed that could achieve optimal use of App Runner. The cost models that were created showed that the costs for App Runner can increase rapidly, however also very dependent on the type of application and the amount of load the application has to handle.

# INHALTSVERZEICHNIS

---

## I THESIS

1	EINLEITUNG	2
1.1	Motivation . . . . .	2
1.2	Ziel der Arbeit . . . . .	2
1.3	Methodik . . . . .	3
2	STAND DER TECHNIK	4
2.1	Cloud-Computing . . . . .	4
2.2	Containerisierung . . . . .	4
2.3	App Runner . . . . .	5
2.3.1	Wichtige Begriffe . . . . .	6
2.3.2	Funktionsweise der Skalierung . . . . .	7
2.3.3	Anforderungen an Anwendung . . . . .	10
3	TESTANWENDUNG	11
4	TESTKONZEPTION	13
4.1	Methodisches Vorgehen . . . . .	13
4.1.1	Black-Box Test . . . . .	13
4.1.2	Test Prinzipien . . . . .	14
4.2	Formulierung der Tests . . . . .	15
4.2.1	Testing-Tool . . . . .	15
4.2.2	Metriken . . . . .	16
4.2.3	Test-Typen . . . . .	17
4.2.4	Ablauf der Tests . . . . .	18
5	TESTERGEBNISSE	19
5.1	Testkonfiguration . . . . .	19
5.2	Pipe-Clean Tests . . . . .	19
5.3	Instanzenminimum von 1 . . . . .	20
5.3.1	Stress Test . . . . .	20
5.3.2	Last Test . . . . .	26
5.4	Instanzenminimum von 2 . . . . .	29
5.4.1	Stress Test . . . . .	29
5.4.2	Last Test . . . . .	32
6	ANALYSE DER ERGEBNISSE	35
6.1	Analyse der Leistungsaspekte . . . . .	35
6.1.1	Verfügbarkeit . . . . .	35
6.1.2	Antwortszeiten . . . . .	37
6.2	Kosten . . . . .	42
6.3	Implikationen für die Praxis . . . . .	45
7	ZUSAMMENFASSUNG UND AUSBLICK	47
	LITERATUR	48

II	APPENDIX	
A	REPOSITORY	51
A.1	Quell-Code der Testanwendung . . . . .	52

## ABBILDUNGSVERZEICHNIS

---

Abbildung 2.1	Aufbau der Containerisierung . . . . .	5
Abbildung 2.2	Modell von App Runner . . . . .	6
Abbildung 2.3	Skalierung bei Instanzenminimum ohne Anfragen . . .	8
Abbildung 2.4	Skalierung bei Instanzenminimum mit 30 Anfragen . .	9
Abbildung 2.5	Skalierung bei Instanzenminimum mit 50 Anfragen . .	9
Abbildung 5.1	Lastanstieg eines Stress Tests, 2000 gleichzeitige Virtual User (VU) . . . . .	21
Abbildung 5.2	Antwortzeiten von A (links) und B (rechts) im Vergleich; Stress Test mit 1500 VU und Instanzenminimum 1 . . . . .	22
Abbildung 5.3	Antwortzeiten von C; Stress Test mit 2000 gleichzeitige VU und Instanzenminimum 1 . . . . .	23
Abbildung 5.4	Fehler-Code-Quote der Stress Tests für Instanzenminimum 1 . . . . .	24
Abbildung 5.5	Antwortzeiten von A; Stress Test mit 4000 gleichzeitige VU und Instanzenminimum 1 . . . . .	25
Abbildung 5.6	Lastanstieg eines Last Tests, 2000 gleichzeitige VU . . .	26
Abbildung 5.7	Antwortzeiten von B; Last Test mit 4000 VU und Instanzenminimum 1 . . . . .	28
Abbildung 5.8	Fehler-Code-Quote der Last Tests für Instanzenminimum 1 . . . . .	28
Abbildung 5.9	Antwortzeiten von B (links) und C (rechts) im Vergleich; Stress Test mit 1500 VU und Instanzenminimum 2 . . . . .	30
Abbildung 5.10	Antwortzeiten von C; Stress Test mit 2000 gleichzeitige VU und Instanzenminimum 2 . . . . .	31
Abbildung 5.11	Antwortzeiten von A; Stress Test mit 4000 VU und Instanzenminimum 2 . . . . .	32
Abbildung 5.12	Antwortzeiten von C; Last Test mit 2000 VU und Instanzenminimum 2 . . . . .	33
Abbildung 5.13	Antwortzeiten von B; Last Test mit 4000 gleichzeitige VU und Instanzenminimum 2 . . . . .	34



## TABELLENVERZEICHNIS

---

Tabelle 3.1	Routen der Testanwendung . . . . .	11
Tabelle 4.1	Leistungsbeeinflussende Konfigurationen . . . . .	13
Tabelle 4.2	Anwendungsfälle der Tests . . . . .	18
Tabelle 5.1	Pipe-Clean Ergebnisse . . . . .	20
Tabelle 5.2	Instanzenminimum 1 - Ergebnisse der Stress Tests . . .	21
Tabelle 5.3	Instanzenminimum 1 - Ergebnisse der Last Tests . . .	26
Tabelle 5.4	Instanzenminimum 2 - Ergebnisse der Stress Tests . . .	29
Tabelle 5.5	Instanzenminimum 2 - Ergebnisse der Last Tests . . .	32
Tabelle 6.1	Konfiguration (Virtual CPU ( <a href="#">vCPU</a> ) und Speicher) . . .	43

## ABKÜRZUNGSVERZEICHNIS

---

AWS	Amazon Web Services
PaaS	Platform as a Service
ECS	Elastic Container Service
VU	Virtual User
vCPU	Virtual CPU
TT	Think Time
P <sub>95</sub>	95. Quantil
P <sub>99</sub>	99. Quantil
KBr	Kosten einer bereitstehenden Instanz
KAk	Kosten einer aktiven Instanz

Teil I

THESIS

## EINLEITUNG

---

Im folgenden Kapitel wird die Motivation dieser Arbeit erläutert. Dafür wird kurz auf die Entwicklungen von Cloud-Computing eingegangen und wie Services wie [AWS](#) App Runner daraus entstanden sind. Im Anschluss wird das Ziel beschrieben, welches in dieser Arbeit verfolgt wird und welche Aspekte bei der Analyse genau untersucht werden. Zum Schluss wird auf die Methodik eingegangen die während der Erstellung dieser Arbeit verwendet wird. Die kennzeichnenden Erkenntnisse werden indessen zusammengefasst dargestellt.

### 1.1 MOTIVATION

Cloud-Computing ist eine IT-Infrastruktur, die sich im letzten Jahrzehnt außerordentlich weiterentwickelt hat. Public Cloud Anbieter wie [AWS](#) oder Microsoft Azure bieten IT-Ressourcen als Dienstleistung an, um Unternehmen zu ermöglichen, Anwendungen schneller, günstiger und mit höherer Ausfallsicherheit in Betrieb nehmen zu können. Einer der Entwicklungen, die Cloud-Computing voran getrieben hat, ist die Nutzung von Containerisierung. Containerisierung ermöglicht es, alle notwendigen Komponenten, wie Bibliotheken, Frameworks und andere Abhängigkeiten einer Anwendung, in Container zu verpacken. Die Anwendung kann dann innerhalb dieses Containers ausgeführt werden, unabhängig von der bestehenden Umgebung, wie z.B. dem Betriebssystem. Ein Nachteil der Containerisierung ist jedoch die manuelle Konfigurationen der Container. Diese setzt gewisse Vorkenntnisse zum Thema Container voraus. App Runner baut auf der Containerisierung auf, ist jedoch in der Nutzung vereinfacht und benötigt kaum Konfigurationen. Mit App Runner ist es möglich, direkt aus einem Code-Repository Quell-Code in Betrieb zu nehmen. Die Nutzung und Inbetriebnahme von Anwendungen soll mit App Runner simpel und ohne Vorkenntnisse von Containerisierung möglich sein. Weiterhin werden die Container von [AWS](#) automatisch verwaltet, dazu gehören die Skalierung, das Load-Balancing und die Ende-zu-Ende-Verschlüsselung. [16]

### 1.2 ZIEL DER ARBEIT

Im Rahmen dieser Arbeit wird die Leistung des neuen Cloud Services App Runner analysiert. Untersucht werden die möglichen Nutzen für Entwickler, sowie die Grenzen in welchen App Runner Anwendung findet. Dafür werden anhand einer Testanwendung Leistungstests durchgeführt, sowie Kosten berechnet und untersucht. Die Ergebnisse der Tests und der Kostenberechnung werden dann analysiert, um die Nutzen und Grenzen von App

Runner spezifizieren zu können. Hierbei werden folgende Fragen beantwortet:

1. Wie der Skalierungsalgorithmus die Leistung beeinflusst.
2. Wie sich die Leistung in verschiedenen Situationen verändert.

Somit kann eine Bewertung über die Leistungsstärke dieses Dienstes getroffen werden und der Nutzen von App Runner für Entwickler klar definiert werden. Mit Hilfe der Ergebnisse dieser Arbeit können potenzielle Nutzer beurteilen für welche Anwendungsfälle App Runner genutzt werden kann.

### 1.3 METHODIK

Zunächst wird App Runner in den Cloud-Computing Kontext eingeordnet. Hierfür werden die Begriffe Cloud-Computing und Containerisierung kurz erläutert und definiert. Im Anschluss wird die Funktionsweise von App Runner erklärt und die Anforderungen an eine App Runner Anwendung dargelegt. Nachdem diese Anforderungen definiert wurden, wird die Testanwendung beschrieben. Diese dient der kurzen Veranschaulichung des Deployment mit App Runner, um anschließend die Leistung von App Runner zu testen. Im Weiteren erfolgt die Testkonzeption; dabei werden sowohl das methodische Vorgehen als auch die verschiedenen Test Typen, sowie Metriken und das Testing Tool vorgestellt. Anschließend wird der Ablauf der Tests beschrieben, um diese daraufhin durchzuführen. Zuletzt werden die Ergebnisse unter Betrachtung verschiedener Leistungsaspekte analysiert, um das vorgestellte Ziel der Arbeit zu erreichen.

## STAND DER TECHNIK

---

Im folgenden Kapitel werden die Begriffe Cloud-Computing, Containerisierung und App Runner erläutert und dessen Funktionsweise beschrieben. Das Verständnis dieser Technologien und Begriffe helfen die Funktionsweise von App Runner genauer zu verstehen.

### 2.1 CLOUD-COMPUTING

Cloud-Computing ist der Oberbegriff für eine Vielzahl an Diensten, welche in vielen verschiedenen Bereichen der Informatik verwendet werden. Diese Dienste bilden verschiedene Arten des Cloud-Computings. Allgemein kann man Cloud-Computing jedoch als die Bereitstellung von dynamisch skalierbarer Rechenressourcen definieren, wie zum Beispiel: Server, Speicher oder Anwendungen [15] (Seite 2).

Die Arten, in die Cloud-Computing eingeordnet werden kann, bilden drei verschiedene Ebenen von Diensten. Es handelt sich dabei um die sogenannten „Infrastructure as a Service“, Platform as a Service (PaaS) und „Software as a Service“. Wir gehen im folgenden auf PaaS genauer ein, da App Runner eine solche Lösung ist.

Unter dem Begriff der PaaS ist die Bereitstellung benötigter Hardware zu verstehen. Hierbei können Entwickler die Anzahl an CPU-Kernen, Arbeitsspeicher und Speicherplatz von Servern, abhängig der unterschiedlichen Anforderungen anpassen und verändern. Dazu können die gebotenen Server abhängig von der momentanen Auslastung automatisch hoch skalieren und unterstützen auf demselben physischen Server mehrere Betriebssysteme. [15] (Seite 2) Es ist jedoch zu beachten, dass der Nutzer selber für die Auswahl der Anwendungen, die auf der Plattform seiner Wahl ausgeführt werden sollen, verantwortlich ist. Das hat zur Folge, dass der Nutzer für die Sicherheitsaspekte im Zusammenhang mit seiner ausgewählten Anwendung verantwortlich ist [22] (Seite 24).

### 2.2 CONTAINERISIERUNG

Die Grundidee der Containerisierung ist es, die Applikation mitsamt ihrer Umgebung, Frameworks und Bibliotheken in ein so genanntes Container-Abbild zu verpacken, welches dann unabhängig von der bestehenden Infrastruktur ausgeführt werden kann. Container-Abbilder sind Code-Pakete dessen Aufbau und Struktur standardisiert sind, was es Entwicklern ermöglicht, diese überall laufen zu lassen [10] (Seite 2). Wenn ein solches Container-Abbild ausgeführt wird, nennt man es einen Container. Es ist möglich mehrere Container auf demselben Rechner gleichzeitig laufen zu lassen, da diese

den Betriebssystemkern des Rechners gemeinsam nutzen und als isolierte Prozesse laufen [10] (Seite 2).

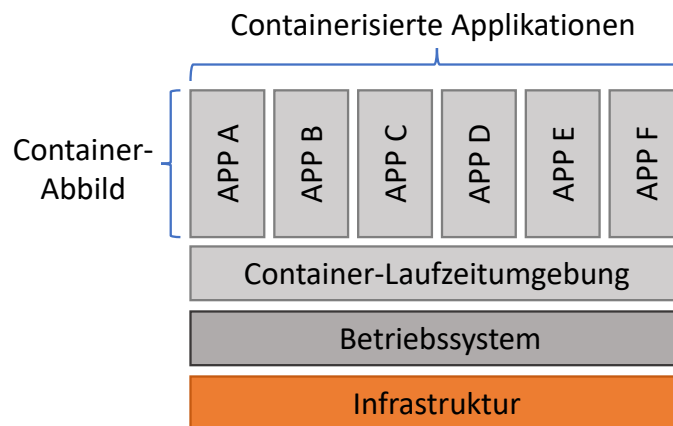


Abbildung 2.1: Aufbau der Containerisierung, angelehnt an: [23]

Durch Containerisierung wird die Einrichtung der Infrastruktur reduziert auf die Installation eines Betriebssystems und einer Container-Laufzeit-umgebung. Die Container-Laufzeitumgebung dient zur Verwaltung aller Container auf einem Server. Dadurch erhalten Entwickler die Möglichkeit die Container-Abbilder lokal auf einem Rechner laufen zu lassen. Dies gestattet es die Anwendung in produktionsähnlicher Umgebung zu simulieren und zu testen, ohne alle notwendigen Abhängigkeiten installieren zu müssen. Weiterhin sind Entwickler in der Lage, sich mehr auf die Entwicklung, Ausführung und Überwachung von Anwendungen zu konzentrieren. [14] (Seite 12)

Diese Form des Betriebs einer Anwendung wurde in die Welt des Cloud-Computings als eine PaaS Lösung eingeführt und ermöglichte es Nutzern ihre Anwendung schneller und agiler in Betrieb zu nehmen. Die vom Nutzer erstellten Container-Abbilder konnten, mithilfe der bestehenden Infrastruktur und Betriebssysteme, in der Cloud genutzt und abhängig von der Auslastung automatisch skaliert werden.

### 2.3 APP RUNNER

Der von AWS entwickelte Service App Runner baut auf dem Prinzip der Containerisierung auf. Bei der Nutzung von App Runner werden dabei jedoch bestimmte Aspekte für den Nutzer abstrahiert. Diese Aspekte bilden die Besonderheiten die den Dienst ausmachen. Diese werden von App Runner verwaltet und können vom Entwickler nicht beeinflusst werden. Die einzigen Aspekte, die von dem Entwickler noch verändert werden können, sind der Quell-Code der Anwendung, sowie die Einstellung einzelner Konfigurationen die von App Runner angeboten werden.

In der Abbildung 2.2, sind die einzelnen Aspekte zu sehen die von App

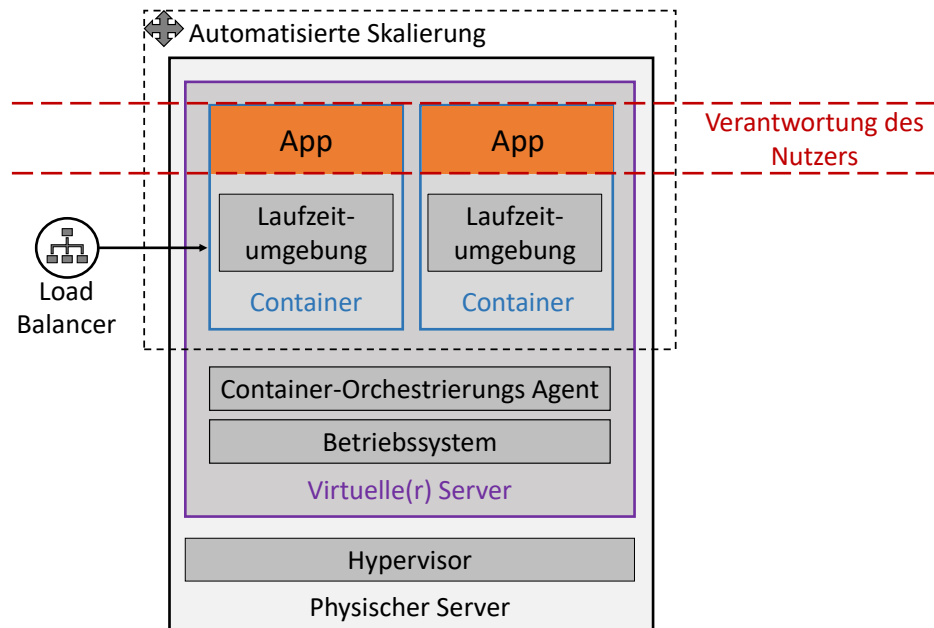


Abbildung 2.2: Modell von App Runner, angelehnt an: [20]

Runner verwaltet werden. Dabei handelt es sich einerseits um die Hardware, also den physischen Server, andererseits um Systeme, wie zum Beispiel einem Hypervisor, der es ermöglicht mehrere virtuelle Server bereitzustellen. Diese virtuellen Server besitzen ein Betriebssystem und einen Container-Orchestrierungs Agenten, welcher die Automatisierung der Bereitstellung, Verwaltung, Skalierung und Vernetzung von Containern übernimmt. Mithilfe dieser Systeme können innerhalb des virtuellen Servers Container erstellt werden, welche von dem Container-Orchestrierungs Agenten verwaltet werden. Dazu gehört das Load-Balancing der Container, also die effiziente Verteilung der Last zwischen den Containern, sowie die automatisierte Skalierung der Container. Das bedeutet, dass abhängig von der aktuellen Last neue Container bereitgestellt oder bestehende aufgelöst werden, um die Last zwischen diesen Container zu verteilen.

Diese Container, auch Instanzen genannt, werden von App Runner automatisch erstellt, um die Applikation eines Nutzers in Betrieb zu nehmen. Hierbei muss der Nutzer Einstellungen bezüglich der Applikation vornehmen, damit die Container korrekt erstellt werden können. Zu diesen Einstellungen gehören die Laufzeitumgebung, sowie die Erstellungs- und Startbefehle der Anwendung [16] (Seite 41). Diese Einstellungen liegen in der Verantwortung des Nutzers, alle anderen in diesem Absatz genannten Aspekte werden von App Runner verwaltet.

### 2.3.1 Wichtige Begriffe

Im Folgenden werden kurz wichtige Begriffe erläutert, wie Instanzen und deren Zustände, die Einstellungen von App Runner zu dem automatischen



Skalieren, sowie die Definition eines Kaltstarts im Kontext von App Runner. Instanzen sind laufende Container der Applikation die mit App Runner in Betrieb genommen wurde. Diese Instanzen können HTTP-Anfragen entgegennehmen und bearbeiten. Weiterhin können sich Instanzen in drei Zuständen befinden. [20]

**AKTIV:** Aktive Instanzen bearbeiten Anfragen und haben vollen Zugriff auf die ihnen zugeteilten Ressourcen.

**UNTÄTIG:** Eine untätige Instanz wartet auf Anfragen. Sie besitzt zwar Ressourcen um Anfragen zu beantworten, diese sind jedoch gedrosselt und benötigen eine gewisse Zeit um aktiviert zu werden.

**IN BEREITSTELLUNG:** Instanzen die sich in Bereitstellung befinden, sind Instanzen die für die Bearbeitung von Anfragen mit Ressourcen vorbereitet werden. Sie sind nicht in der Lage Anfragen anzunehmen.

Diese Instanzen können konfiguriert werden. Zu diesen Konfigurationen gehören die Anzahl vCPU und Speicher, sowie Einstellungen zu der automatisierten Skalierung, wobei diese Konfigurationen für alle Instanzen gelten. Es gibt drei Werte die eingestellt werden können um den Skalierungsalgorithmus von App Runner zu beeinflussen.

Der erste Wert ist die „maximum Concurrency“. Dieser Wert beschreibt die maximale Anzahl an gleichzeitig laufenden Anfragen innerhalb einer Instanz. Wenn diese Anzahl erreicht wurde, kann diese Instanz keine weiteren Anfragen annehmen und App Runner skaliert den Dienst hoch, das bedeutet, dass eine weitere Instanz aktiviert wird [16] (Seite 56). Der nächste Wert ist das Instanzenminimum. Es ist die minimale Anzahl an Instanzen welche allzeit bereitgestellt sein müssen und somit untätig auf HTTP-Anfragen warten. Mit diesem Wert wird eine Reduzierung der Instanzen auf null verhindert [16] (Seite 56). Zuletzt gibt es den Wert des Instanzenmaximums. Diese Zahl gibt an wie viele Instanzen maximal gleichzeitig aktiv sein dürfen [16] (Seite 56). Diese Einstellungen geben dem Skalierungsalgorithmus von App Runner Basiswerte, um das Skalierungsverhalten spezifisch für die Applikation anzupassen.

Zum Schluss wird kurz der Begriff eines Kaltstarts im Kontext von App Runner erläutert. Zu einem sogenannten Kaltstart kommt es in App Runner wenn alle aktiven Instanzen mit der „maximum Concurrency“ ausgelastet sind und weitere Anfragen ankommen. Hierbei müssen neue Instanzen bereitgestellt und aktiviert werden. Die Anfragen die nicht von einer Instanz bearbeitet werden können, werden in eine Warteschlange gestellt. Die wartenden Anfragen erleben einen Kaltstart und werden verzögert oder gar nicht bearbeitet.

### 2.3.2 Funktionsweise der Skalierung

Hier wird kurz die Funktionsweise der Skalierung beschrieben. Dazu wird die folgende Konfiguration verwendet:

Das Instanzenminimum liegt bei 2 Instanzen; das heißt es sind immer mindestens zwei Instanzen bereitgestellt, die untätig auf Anfragen warten. Das Instanzenmaximum liegt bei 5 Instanzen; somit können nie mehr als fünf Instanzen gleichzeitig aktiv sein. Die „maximum Concurrency“ wird auf 20 Anfragen gesetzt. Somit kann jede Instanz maximal 20 Anfragen gleichzeitig bearbeiten. Grundsätzlich sind folgende Szenarien möglich: [20]

1. Keine Anfragen: Die minimale Anzahl an Instanzen, in diesem Beispiel zwei Instanzen, sind bereit und warten untätig auf Anfragen (siehe Abbildung 2.3). In diesem Zustand sind die Instanzen zwar gedrosselt, jedoch in der Lage ankommende Anfragen abzuarbeiten.

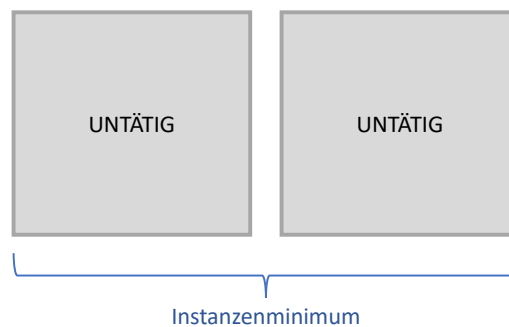


Abbildung 2.3: Skalierung bei Instanzenminimum ohne Anfragen, angelehnt an: [20]

2.  $\text{Max. Concurrency} * \text{Inst. Min.} > \text{Anfragen}$ : In Abbildung 2.4 ist der Fall zu sehen bei dem die ankommenden Anfragen unter der summierten Concurrency des Instanzenminimums liegt. Hier werden 30 Anfragen abgeschickt, welche gleichmäßig auf die zwei Instanzen verteilt werden. Falls die Instanzen untätig waren werden sie nun aktiv und beginnen die Anfragen abzuarbeiten. Sollte die Anzahl an Anfragen hoch bleiben, kann es hierbei dazu kommen, dass der Skalierungsalgorithmus bereits im Hintergrund eine weitere Instanz bereitstellt. Dies wird vom Skalierungsalgorithmus automatisch eingeleitet, um bei weiter ansteigenden Last einen Kaltstart zu vermeiden.
3.  $\text{Max. Concurrency} * \text{Inst. Min.} < \text{Anfragen}$ : Werden mehr Anfragen geschickt als die bereits aktiven Instanzen bearbeiten können, kommt es zu Kaltstarts. Wenn z.B. 50 Anfragen gleichzeitig geschickt werden, können 40 Anfragen auf die bereitgestellten Instanzen verteilt werden. Die übrig gebliebenen Anfragen werden in eine Warteschlange gestellt. Diese Warteschlange ist ein Buffer in dem Anfragen temporär abgespeichert werden können, um später erneut gesendet zu werden. Wie in Abbildung 2.5 zu sehen, stellt App Runner eine neue Instanz bereit,

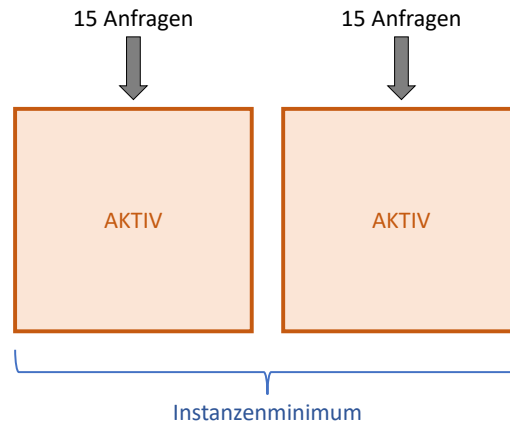


Abbildung 2.4: Skalierung bei Instanzenminimum mit 30 Anfragen, angelehnt an: [20]

welche die Anfragen in der Warteschlange abarbeitet, sobald diese aktiv ist. Die wartenden Anfragen erleben einen Kaltstart und haben eine längere Antwortzeit. Sobald die neue Instanz aktiv ist, werden alle kommenden Anfragen wieder gleichmäßig verteilt.

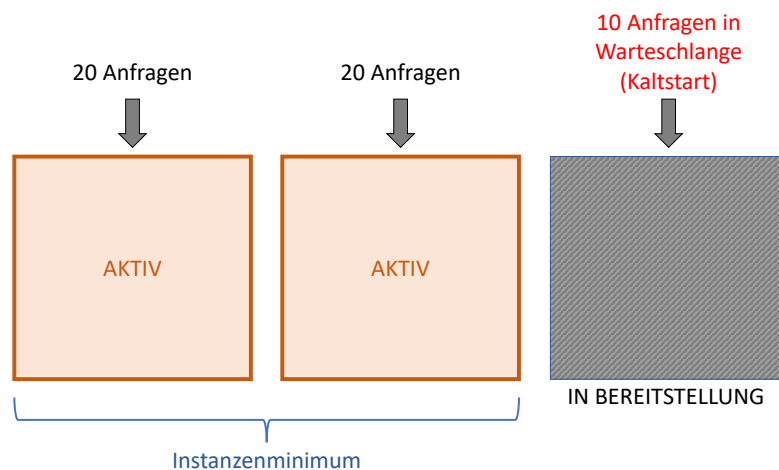


Abbildung 2.5: Skalierung bei Instanzenminimum mit 50 Anfragen, angelehnt an: [20]

4.  $\text{Max. Concurrency} * \text{Inst. Max.} < \text{Anfragen}$ : Zuletzt gibt es noch den Fall, dass die Anzahl an Anfragen höher ist als die „maximum Concurrency“ des Instanzenmaximums. Hierbei kann nicht weiter hochskaliert werden, da alle verfügbaren Instanzen bereits aktiv sind. Hier kommt es zu einer Überlastung des Systems und viele Anfragen werden nicht beantwortet.

### 2.3.3 Anforderungen an Anwendung

App Runner unterstützt die Full Stack Entwicklung, einschließlich Frontend- und Backend-Webanwendungen, darunter fallen API-Dienste, Backend-Webdienste oder Websites. Jedoch müssen, laut des Developer Guides für App Runner, Applikationen folgende Anforderungen erfüllen [16].

Grundsätzlich können alle Anwendungen betrieben werden die auf Web-Anfragen basieren, also ein HTTP-Anfrage-Antwort basierter Dienst sind. Zudem müssen einzelne Instanzen der Anwendung in der Lage sein, während ihrer Lebensdauer, mehrere Anfragen gleichzeitig zu bedienen. Weiterhin müssen Anwendungen die mit App Runner betrieben werden zustandslos sein, da App Runner nach Durchführung einer Anfrage keine Zustände persistent abspeichert. Das bedeutet, dass alle Anfragen isoliert und unabhängig von vorherigen oder kommenden Anfragen bearbeitet werden müssen. App Runner besitzt nur einen flüchtigen Speicher. Es werden daher nach Durchführung einer Anfrage keine Daten persistent abgespeichert. Das hat zur Folge, dass Anfragen nicht von lokalen Daten abhängig sein dürfen. Um Daten persistent abzuspeichern müsste daher eine externe Datenbank an die Anwendung gekoppelt werden. Zuletzt müssen bestimmte Umgebungsvariablen angegeben werden. Eine wichtige Umgebungsvariable ist der Port auf dem die Anwendung läuft. Es können jedoch weitere Umgebungsvariablen für die Applikation eingefügt und abgespeichert werden.

## TESTANWENDUNG

In diesem Kapitel wird der Aufbau und die Funktionsweise der Testanwendung beschrieben, mit der im Anschluss die Leistungstests durchgeführt werden. App Runner bietet zwei Möglichkeiten seine Anwendung in Betrieb zu nehmen [16] (Seite 21). Die erste Möglichkeit ist ein bereits existierendes Container-Image mit der Anwendung, ohne Erstellung seitens App Runner, in Betrieb zu nehmen. Die zweite Möglichkeit ist den Quell-Code der Anwendung über GitHub mit App Runner zu verknüpfen. Hierbei muss App Runner die Anwendung direkt aus dem Quell-Code heraus erstellen und in Betrieb nehmen. Um dies zu ermöglichen muss eine yaml-Datei erstellt werden, in der die Runtime, sowie der Build und Start Befehl spezifiziert sind. Des Weiteren muss in dieser Datei der Port angegeben werden in dem die Anwendung später laufen soll. Diese Datei wird dann bei Inbetriebnahme von App Runner ausgelesen und zum Erstellen der Anwendung genutzt. Bei der Entwicklung dieser Anwendung wurden die Anforderungen an App Runner Anwendungen von Kapitel 2.3.3 erfüllt. Es handelt sich bei der Anwendung um ein einfaches HTTP-Backend für eine kleine Koch-Rezepte-Seite. Auf dieser Rezepte-Seite kann man Rezepte aufrufen, verändern, oder erstellen kann. Es sind insgesamt 40 Beispiel-Rezepte in einer eigenen Datei ausgelagert, welche in dieser als JSON-Objekte abgespeichert wurden. Um diese Aktionen durchzuführen werden die Routen, wie sie in Tabelle 3.1 zu finden sind, angeboten. Es wurde auf die Nutzung einer externen Daten-

HTTP-Verb	Nutzung	Pfad
GET	Alle Rezepte aufrufen	/recipes
GET	Ein Rezept aufrufen	/recipes/{id}
PATCH	Ein Rezept verändern	/recipes/{id}
POST	Ein Rezept erstellen	/recipes/

Tabelle 3.1: Routen der Testanwendung

bank verzichtet, um einen simplen App Runner Anwendungsfall zu simulieren. Des Weiteren wurde kein Autorisierung-Mechanismus verwendet, um eine konstante Benutzer-Rate zu erhalten. Hierbei wurde zur Umsetzung die Runtime Node.js 12 und für die Entwicklung der Anwendung das Express Framework in der Version 4.17.1 verwendet. Zuletzt kann man noch die Konfiguration der vCPU und des Speichers einstellen. AWS beschreibt vCPUs als einzelne Threads eines CPU-Kerns [19] (Seite 576). Hierbei hat man die Wahl zwischen 1vCPU mit entweder 2, 3 oder 4 Gigabyte, oder 2vCPU mit 4 Gigabyte. Aufgrund der beachtlichen Größen der auswählbaren Konfigurationskombinationen wird für die folgenden Tests nur die Kombinati-

on einer **vCPU** mit 2 Gigabyte Speicher betrachtet. Die exakte Konfiguration bezüglich des Testens wird später in 5.1 dargestellt. Dort werden die einzelnen Konfigurationen nochmal beschrieben und anschließend getestet. Code der Anwendung ist zu finden unter <https://github.com/laura-franky/bachelor-project>.

## TESTKONZEPTION

Kapitel 4 widmet sich der Konzeption der Tests. Hierbei wird zuerst das methodische Vorgehen dieser Arbeit spezifisch vorgestellt. Im Anschluss werden die Tests formuliert, dazu werden das Testing-Tool, die verwendeten Metriken, die Test-Typen und der Ablauf der Tests erläutert.

### 4.1 METHODISCHES VORGEHEN

Ziel der Tests ist es die Performance von App Runner zu erfassen. Zur erfolgreichen Umsetzung werden zuerst die Tests genauer definiert. Mithin wird zunächst die jeweilige Art der Tests näher beleuchtet, welche Aspekte der Performance sie umfassen und wie diese gestaltet werden, um reproduzierbare Ergebnisse zu liefern.

#### 4.1.1 *Black-Box Test*

Allgemein wird sich bei den kommenden Tests um Systemtests handeln, da das gesamte System getestet wird [21](Seite 79). Weiter handelt es sich um Black-Box Tests, da der innere Aufbau von App Runner nicht einsehbar ist. Bei Black-Box Tests können nur Ein- und Ausgaben erfasst und analysiert werden, wobei nur die Eingaben kontrollierbar sind [21](Seite 141). Im Falle von App Runner handelt es sich bei diesen Eingaben einerseits um die entwickelte Testanwendung, andererseits um die Konfigurationen welche bei Inbetriebnahme vorgenommen werden können. Alle weiteren Vorgänge können vom Anwender nicht beeinflusst werden.

Bei den Konfigurationen die App Runner anbietet beeinflussen nur gewisse Einstellungen die Performance der resultierenden Applikation. Dazu gehören Konfigurationen bezüglich der Skalierung, der CPU und des Speichers. Wie bereits beobachtet sind die Konfigurationen bezüglich CPU & Speicher für die entwickelte Anwendung zu mächtig, aufgrund dessen wird nur eine Konfiguration betrachtet. Diese Eingabewerte können noch weiter

Leistungsaspekt	Konfigurationsmöglichkeit
CPU & Speicher	Anzahl vCPU & Größe des Speichers
Skalierung	Instanzenminimum Instanzenmaximum „Maximum Concurrency“

Tabelle 4.1: Leistungsbeeinflussende Konfigurationen

eingeschränkt werden, um die zu testenden Kombinationsmöglichkeiten ge-

ring zu halten. Zum Beispiel sollte der Wert der „maximum Concurrency“ möglichst hoch gesetzt werden, ohne dass die Applikation überlastet. Dies ermöglicht App Runner die Instanzen effizient auszunutzen und nur bei Notwendigkeit neue bereitstellen zu müssen. Generell bedeutet das, dass die „maximum Concurrency“ von der Effizienz des Codes der Applikation abhängig ist und in den kommenden Tests für alle Konfigurationen gleichgesetzt wird. Ein weiterer Wert, der in den Konfigurationen nicht untersucht wird, ist das Instanzenmaximum. Da es sich bei diesem Wert um die maximale Anzahl aktiver Instanzen handelt, also die maximale Last die die Applikation aushalten soll, wird hier ein Wert im Vorhinein festgelegt und für alle Konfigurationen verwendet. Somit bleiben die Einstellungen bezüglich des Instanzenminimum als Eingabewert für den Black-Box Test übrig.

#### 4.1.2 Test Prinzipien

Nun können aus den herausgearbeiteten veränderbaren Werten für die Tests genaue Aspekte der Leistung untersucht werden. Zum Beispiel welchen Einfluss die Anzahl an stetig verfügbaren Instanzen auf die Leistung hat. Da die Eingabemöglichkeiten des Black-Box Tests bereits eingeschränkt wurden ist es möglich alle Kombinationsmöglichkeiten zu betrachten. Um ein möglichst genaues Bild der Leistung zu erhalten sollten möglichst viele Kombinationsmöglichkeiten abgedeckt und getestet werden [12](Seite 61). Hierbei wird nach bestimmten Prinzipien gearbeitet. Anhand dieser Prinzipien soll eine reproduzierbare Test-Umgebung erstellt werden. Die Prinzipien lauten wie folgt: [13](Seiten 2 - 6)

- P1: WIEDERHOLTE VERSUCHE Es wird empfohlen Tests für eine Konfiguration mehrmals durchzuführen um statistische Ungewöhnlichkeiten erkennen zu können. Aus diesem Grund werden in dieser Arbeit alle Tests einer Konfiguration fünfmal ausgeführt.
- P2: ABDECKUNG VON ARBEITSLAST UND KONFIGURATION Die Tests sollen eine Vielzahl an möglichen Szenarien abdecken. Deswegen werden verschiedene Werte-Kombinationen von Instanzenminima getestet. Des Weiteren werden drei Test-Typen durchgeführt und pro Test drei verschiedene Anwendungsfälle getestet.
- P3: BESCHREIBUNG DES VERSUCHSAUFBAUS Es sollten alle Variablen, welche Teil der Tests sind und Einfluss auf diese haben, genau beschrieben werden. Dazu gehören die Testanwendung, die Konfigurationen der einzelnen Tests, sowie das Testing-Tool, den genauen Ablauf und das Ziel der Tests.
- P4: OFFEN ZUGÄNGLICHE ARTEFAKTE Verwendete Artefakte die für die Durchführung der Tests notwendig waren, sowie aus ihnen entstandenen Protokolle sollten offen zugänglich sein. Hierfür wird ein öffentliches GitHub Repository verwendet.



- P5: **ERGEBNISBESCHREIBUNG DER GEMESSENEN LEISTUNG** Die Ergebnisse der Tests sollten angemessen visualisiert und beschrieben werden. Dazu sollten die Ergebnisse aussagekräftig und allgemein sein, weswegen zum Beispiel für die Antwortzeiten Quantile zur Beschreibung verwendet werden. Die Visualisierung und Beschreibung der Tests erfolgt in Kapitel 5.
- P6: **STATISTISCHE AUSWERTUNG** Schlussfolgerungen die aus den gemessenen Daten gemacht werden sollten statistisch bewertet werden, um die Signifikanz der erzielten Ergebnisse beurteilen zu können. Für diese Arbeit kann jedoch nicht gewährleistet werden, dass die Ergebnisse dieser Arbeit eine Signifikanz haben, weswegen dieses Prinzip nicht genauer betrachtet wird.
- P7: **MESSEINHEITEN** Es soll und wird immer die zugehörige Messeinheit eines Ergebnisses angegeben.
- P8: **KOSTEN** Es wird ebenfalls empfohlen immer das Kostenmodell, sowie die abgerechneten Kosten eines Services anzugeben. Aufgrund dessen werden die Kosten aller Tests im Kapitel 5.4 errechnet und festgehalten.

## 4.2 FORMULIERUNG DER TESTS

Nun werden die Tests genauer erläutert. Dabei werden zuerst Testing-Tool, verwendete Metriken und die Test-Typen vorgestellt. Im Anschluss wird der Ablauf der Tests präsentiert, indessen werden die Anwendungsfälle geschildert, die mit den verschiedenen Konfigurationen getestet werden.

### 4.2.1 *Testing-Tool*

Für die kommenden Performance Test wird die Pro-Version des Testing-Tools Artillery.io verwendet. Artillery.io ist ein Open-Source Performance-Toolkit für Leistungstest [3]. Es werden eine Reihe unterschiedlicher Tests angeboten, darunter Stress und Last Tests. Hierbei muss eine yaml-Datei angelegt werden, welche die Tests spezifiziert. Inhalt dieser Datei ist aufgebaut wie folgt:

**KONFIGURATION:** Die Konfiguration beinhaltet die Load Phasen, welche pro Test mehrfach definiert werden können. In einzelnen Load Phasen wird die Dauer der Phase in Sekunden, die Anzahl an **VU** die pro Sekunde erstellt werden, die maximale Anzahl an **VU** pro Phase und den Namen der Phase festlegen. Des Weiteren kann man eine steigenden Anzahl an **VU** festlegen. Hierbei wird dann die Anzahl ankommender Benutzer während der Phase stetig erhöht, bis die angegebene Anzahl erreicht worden ist. Dies ermöglicht eine exponentielle Steigerung der Last. Generell ist jedoch zu beachten, dass die summierte Dauer der Load Phasen nicht die Dauer des Tests angibt, sondern die Zeitspanne

in der die [VU](#) erstellt werden. Die tatsächliche Dauer der Tests ist die benötigte Zeit, bis alle erstellten [VU](#) ihr Szenario abgeschlossen haben. [4] Innerhalb einer Konfiguration kann man mehrere Load-Phasen zu einem Environment zusammenfassen. Dieses Environment kann dann unabhängig von anderen Environments in einer Konfiguration ausgeführt werden.

**SZENARIEN:** Bei einem Szenario handelt es sich um einen Anwendungsfall für die Tests. Ein Szenario beinhaltet den Namen des Szenarios, die einzelnen Schritte die ein [VU](#) abarbeitet, alias Flow, sowie die „Think Time ([TT](#))“, die nach jedem Schritt eingefügt wird. [4] Pro Anwendungsfall wurde ein Szenario geschrieben.

Für jeden Test-Typ wird eine Datei angelegt, welche die Konfiguration der Tests beinhaltet. Des Weiteren wird für jeden Anwendungsfall eine Datei angelegt, welche das Szenario beinhaltet. Die Ergebnisse der Tests werden alle 10 Sekunden auf der Konsole ausgegeben. Dazu werden nach kompletter Beendigung des Tests alle Ergebnisse aggregiert auf der Konsole ausgegeben und als eine JSON-Datei abgespeichert. Für die Durchführung der Tests wird von Artillery-Pro ein Elastic Container Service ([ECS](#)) Cluster in [AWS](#) als Load Generator benötigt [1]. Es handelt sich dabei um ein Fargate Cluster, von dem aus die in den Tests definierte Last an die Anwendung gesendet wird. Dieses Cluster wurde in der [AWS](#) Region eu-central-1, Frankfurt in Betrieb genommen. Die laufende Anwendung hingegen in der [AWS](#) Region us-east-1 (North Virginia). Um das vierte Prinzip einzuhalten werden die Dateien in dem [GitHub Repository](#) abgespeichert. Dort sind neben dem Code der Anwendung auch die Konfigurations Dateien der Tests im Ordner „artillery“ und deren Ergebnisse im Ordner „reports“ beinhaltet.

#### 4.2.2 Metriken

Nach Durchführung eines Performance Tests gibt Artillery.io eine vollständige Zusammenfassung der Ergebnisse der Tests in einer JSON-Datei aus. Folgende Metriken sind in diesen Ergebnissen beinhaltet: [4]

1. Dauer: Die Zeit die benötigt wurde bis alle gestarteten [VU](#) ihre Szenarien abgeschlossen haben. Somit die Dauer des Tests insgesamt.
2. Abgeschlossene Anfragen: Alle erfolgreich abgeschlossenen Anfragen während des Tests. Erfolgreich bedeutet hier ein 200-HTTP-Code.
3. Antworten pro Sekunde: Die durchschnittliche Anzahl erfolgreicher Antworten pro Sekunde.
4. Antwortzeit (in ms): Die Antwortzeit wird gemessen vom Abschieken einer Anfrage bis zum Erhalt der Antwort. Dabei werden mehrere Werte festgehalten: Minimale, Maximale und Mediane Antwortzeit, sowie das 95. Quantil ([P95](#)) und 99. Quantil ([P99](#)).

## 5. HTTP Status-Codes, wie zum Beispiel:

- a) 200: OK. Die Anfrage konnte erfolgreich bearbeitet werden. [5]
- b) 429: Too Many Requests. Es wurden zu viele Anfragen in kurzer Zeit gesendet [6]. AWS drosselt alle API-Anforderungen auf eine maximale API-Anfrage-Rate pro Endpunkt [18]. Diese maximale API-Anfrage-Rate ist hierbei die „maximum Concurrency“ pro Instanz. 429-Codes werden gesendet wenn diese Schwelle überschritten wurde und hochskaliert werden muss.
- c) 502: Bad Gateway. Der Server hat eine ungültige Antwort erhalten [7].
- d) 503: Service Unavailable. Der Server ist vorübergehend nicht in der Lage, die Anfrage zu bearbeiten. Dies kann darauf zurückzuführen sein, dass der Server überlastet ist [8].

## 6. Weitere Fehler:

- a) ETIMEDOUT: Wenn eine Antwort auf eine Anfrage länger als 10 Sekunden benötigt, bricht Artillery die Anfrage ab und meldet diesen Fehler [2].
- b) ECONNRESET: Hierbei handelt es sich um eine abrupte Unterbrechung der Verbindung von Server Seite aus.

Anhand dieser JSON-Datei kann eine HTML-Datei erstellt werden, welche die Metriken in Graphen darstellt. Diese Ergebnisdateien und -graphen werden in dem [GitHub Repository](#) abgespeichert, zur Betrachtung aller Tests einzeln. Metriken die nur in AWS betrachtet werden können, wie zum Beispiel die Anzahl aktuell laufender Instanzen, kann nicht als Artefakt abgespeichert werden, da diese nur auf der AWS App Runner Konsole abgelesen werden können.

## 4.2.3 Test-Typen

Um möglichst viele Aspekte der Performance von App Runner zu testen werden drei verschiedene Test-Typen verwendet.

1. Pipe-Clean Test: Zuerst wird ein sogenannter Pipe-Clean Test ausgeführt. Dieser Test soll einerseits als funktionsprüfendes Beispiel dienen und andererseits eine Leistungsbasislinie für die weiteren Tests darstellen. Hierbei wird die Performance der Anwendung bei nur einem Benutzer pro Sekunde gemessen. [11] (Seite 50)
2. Stress Test: Ein weiterer Test ist der Stress Test. Dieser Test dient zum Testen der Stabilität der Anwendung und zum Ausreizen der Grenzen. Dazu wird die Anwendung einer anfangs normalen bis hin zu einer unüblich hohen Last ausgesetzt mit dem Ziel die obere Grenze zu überschreiten [9] (Seite 13).

3. Last Test: Zum Schluss wird ein Last Test durchgeführt, hierbei wird ein Normalbetrieb simuliert indem eine exponentiell steigende Zahl **VU** erstellt wird. Der Fokus liegt auf dem Verhalten der Leistung bezüglich Antwortzeit und Anzahl der Fehler. Es wird das maximale Volumen, welches das System aushalten kann ermittelt [9] (Seite 13).

#### 4.2.4 Ablauf der Tests

Zunächst werden die Anwendungsfälle vorgestellt, welche von allen Test-Typen abgearbeitet werden. Für alle Anwendungsfälle ist eine „**TT**“ eingeplant, welche notwendig ist um ein realitätsnahes Szenario zu simulieren [11] (Seite 54). Hierbei sind für das Betrachten aller Rezepte, sowie für die Erstellung eines Rezepts zwei Sekunden vorgesehen, für das Auswählen und die Bearbeitung eines Rezepts eine Sekunde. Anwendungsfall A: Aufrufen aller Rezepte. Hierbei wird die URL der Anwendung aufgerufen und alle Rezepte werden zurückgegeben. Anwendungsfall B: Rezept bearbeiten. Hier wird erneut die URL aufgerufen um diesmal ein bestimmtes Rezept auszuwählen und dieses danach zu bearbeiten. Anwendungsfall C: Rezept erstellen. Bei der Erstellung eines Rezepts werden zuerst alle Rezepte aufgerufen, um dann über eine bestimmte Route ein Neues zu erstellen. Danach werden nochmal alle Rezepte aufgerufen um das neu erstellte zu betrachten.

Der Ablauf der Tests ist wie folgt, zuerst wird die System Konfiguration

Name	Anwendungsfall	HTTP-Verb	Pfad	<b>TT</b>
A	Alle Rezepte aufrufen	GET	/recipes	2 Sek.
B	Alle Rezepte aufrufen	GET	/recipes	2 Sek.
	Ein Rezept auswählen	GET	/recipes/{id}	1 Sek.
	Das Rezept bearbeiten	PATCH	/recipes/{id}	1 Sek.
C	Alle Rezepte aufrufen	GET	/recipes	2 Sek.
	Ein Rezept erstellen	POST	/recipes/	2 Sek.
	Alle Rezepte aufrufen	GET	/recipes	2 Sek.

Tabelle 4.2: Anwendungsfälle der Tests

vorgestellt mit der getestet wird. Für diese Konfiguration werden alle eben vorgestellten Test-Typen durchgeführt. Zuerst die Pipe-Clean Tests, um eine Basis Leistung festhalten zu können und die Funktionalität der Anwendung zu prüfen. Diese Ergebnisse dienen als Vergleichsbasis für die kommenden Tests. Daraufhin werden Stress Tests ausgeführt. Die Anzahl der **VU** wird hierbei stufenweise erhöht, bis zu dem Punkt, an dem das System die Last nicht mehr abarbeiten richtig kann. Danach werden Last Tests durchgeführt, wobei die Last exponentiell und schnell zunimmt, bis die von den Stress Test erfasste Grenze erreicht wird. Für jeden Test-Typ werden alle drei Anwendungsfälle getestet und fünfmal durchgeführt. Die resultierenden Ergebnisse werden daraufhin kurz zusammengefasst dargestellt und visualisiert.

## TESTERGEBNISSE

---

Dieses Kapitel befasst sich mit den Ergebnissen der in Kapitel 4 beschriebenen Tests. Zunächst wird die genutzte Konfiguration der Anwendung, sowie die Konfiguration der Tests vorgestellt. Im Anschluss werden die Ergebnisse der Tests präsentiert, beginnend mit den Pipe-Clean Tests. Danach werden die Ergebnisse der Stress und Last Tests beschrieben. Hierbei werden zuerst alle Ergebnisse für das Instanzenminimum von eins dargestellt. Zuletzt dann die Ergebnisse mit dem Instanzenminimum von zwei.

### 5.1 TESTKONFIGURATION

Die genutzte Konfiguration des **vCPU** und des Speichers ist für alle folgenden Tests auf **1vCPU** und 2GB Speicher gesetzt. Dazu ist die „maximum Concurrency“ auf 50 gleichzeitig laufende Anfragen pro Instanz gesetzt und das Instanzenmaximum auf drei. Wie bereits erwähnt werden zwei Instanzenminima getestet. Erst mit einer minimalen Instanz und dann im Anschluss mit zwei minimalen Instanzen.

Die Testkonfiguration der Artillery Tests ist pro Test-Typ unterschiedlich. Für die Pipe-Clean Tests handelt es sich um maximale einem **VU** pro Sekunde. Für die Stress Tests wurden mehrere Testreihen mit unterschiedlichen maximalen **VU** durchgeführt. Es wurden die **VU** Maxima von 1500, 2000 und 4000 gleichzeitige Benutzer gewählt. Um diese Last zu erzeugen wurden 10 Workers genutzt, welche parallel mit jeweils 150, 200 und 400 **VU** die einzelnen Tests durchführten. Für die Last Tests wurden die maximalen **VU** auf 2000 und 4000 gleichzeitige Benutzer gesetzt. Um diese Last zu erzeugen wurden erneut 10 Workers genutzt, welche parallel mit jeweils 200 und 400 **VU** die einzelnen Tests durchführten. Wie in 2.3.3 geschildert wird jeder dieser Tests für jeden Anwendungsfall fünfmal durchgeführt.

Da die Ergebnisse der Tests in den meisten Fällen nur geringe Abweichungen ergaben wurde auf eine Darstellung der einzelnen Testdurchläufe verzichtet. Stattdessen wurden die Werte eines Testdurchlaufs repräsentativ für alle gewählt und tabellarisch dargestellt. Diese Tabellen wurden für jeden Testtyp aufgestellt. Bei größeren Unstimmigkeiten in den Ergebnissen wird in den spezifischen Testbeschreibungen näher darauf eingegangen.

### 5.2 PIPE-CLEAN TESTS

Hier werden die Ergebnisse der Pipe-Clean Tests der beiden Instanzenminima vorgestellt. Dieser Testdurchlauf dient einerseits als Funktionsprüfung der Anwendung, andererseits zur Erfassung von Basiswerten zum Vergleich mit weiteren Tests. Die Testlaufzeit betrug für alle Testdurchläufe 10 Minu-

ten und lieferte folgende Ergebnisse:

Metrik / Anwendungsfall	A		B		C	
Anfragen	200		360		258	
Durchschnitt. Anfragen/Sek.	0,33		0,6		0,43	
Instanzenminimum	1	2	1	2	1	2
Minimale Antwortzeit (ms)	92	91	88	90	88	91
Maximale Antwortzeit (ms)	184	1006	1098	1029	164	1015
Mediane Antwortzeit (ms)	93	93	95	93	95	92,5
95. Quantil (ms)	94	94	95	94,5	95	94
99. Quantil (ms)	99,5	97,5	111	109,8	99	102,4

Tabelle 5.1: Pipe-Clean Ergebnisse

Mit der Testkonfiguration von einem maximalen **VU** pro Sekunde entstanden die in Tabelle 5.1 aufgelistete geringe Anzahl Anfragen und durchschnittlich beantworteter Anfragen pro Sekunde. Jeder **VU** hat eine Sekunde für seine Anfragen, wobei dieser oft die Anfragen beendete bevor ein weiterer **VU** erstellt wurde. Diese Last wurde nicht erhöht, dadurch handelt es sich bei diesen Ergebnissen um die Antwortzeiten einer einzigen aktiven Instanz.

Die Tabelle 5.1 stellt eine detaillierte Liste der Ergebnisse der PipeClean Tests bei drei verschiedenen Anwendungsfällen dar. Hier ist zu erkennen, dass die Antwortzeiten aller Anwendungsfälle und Instanzen Konfigurationen untereinander geringe Unterschiede aufweisen. Der einzige große Unterschied zwischen den Tests ist die maximale Antwortzeit welche bei 4 von 6 Tests knapp über 1000 ms liegt. Nimmt man das Maximum aus der allgemeinen Betrachtung bewegen sich die Antwortzeiten zwischen 88 und 111 ms, wobei der Median zwischen 92,5 bis 95 ms liegt.

### 5.3 INSTANZENMINIMUM VON 1

Nun werden die Ergebnisse der Tests, von der in 5.1 beschriebenen Konfiguration, mit einem Instanzenminimum von eins beschrieben. Zunächst werden die Stress Tests und im Anschluss die Last Tests dargestellt.

#### 5.3.1 Stress Test

Das Ziel dieser Tests ist es einerseits die Leistung einer Instanz zu bestimmen, andererseits den Punkt zu ermitteln an dem eine einzelne Instanz auf eine zweite Instanz hochskaliert und wie diese Skalierung sich auf die Leistung auswirkt. Zunächst wurde eine Aufwärmphase gestartet. Diese dauerte 30 Sekunden mit einer maximalen Anzahl **VU** von fünf. Nach dieser initialen Aufwärmphase wurde die Anzahl der **VU** in zwanziger Schritten stufenweise erhöht. Hierbei wurde jede Stufe 60 Sekunden lang gehalten. Zuletzt gab es eine vier Minuten lange Phase in der die maximale Last gehalten wurde.

Es gab keine Cool-Down Phase der Tests, da hierbei nicht das Cool-Down Verhalten untersucht werden sollte. In der Abbildung 5.1 ist ein typischer Ablauf eines Stress Tests zu sehen.

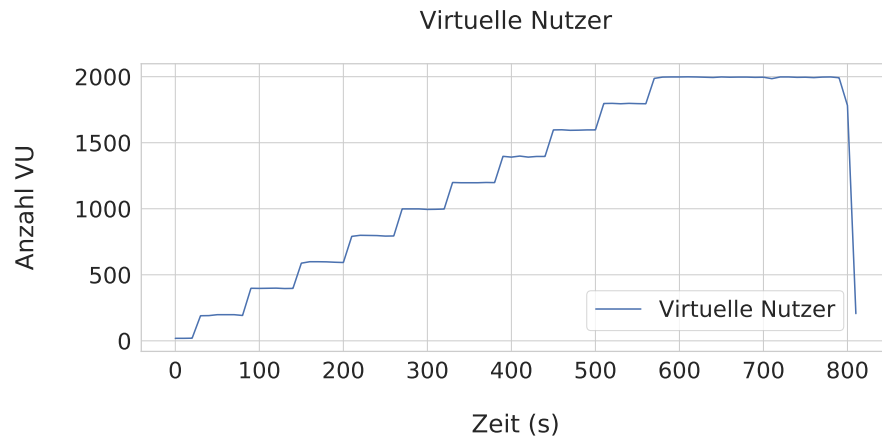


Abbildung 5.1: Lastanstieg eines Stress Tests, 2000 gleichzeitige VU

Instanzenminimum 1	A			B			C		
Metrik \Maximale VUs	1500	2000	4000	1500	2000	4000	1500	2000	4000
Anfragen	293870	394421	1343268	453346	662522	2037131	321159	466179	1463531
Durchschn. Anfragen/Sek.	420,83	480,65	943,01	643,26	776,3	1431,41	456,32	565	1027,34
Min. Antwortzeit (ms)	64	60	2	34	38	4	64	46	10
Max. Antwortzeit (ms)	3141	10174	3717	1797	3589	4281	1628	1878	3759
Mediane Antwortzeit (ms)	95	95	95	95	95	99	95	95	95
95. Quantil (ms)	95	1407	271	239	255	463	95	207	303
99. Quantil (ms)	103	3967	575	607	607	767	107	543	639
Error: ETIMEDOUT	1	1225	16	3	1	25	4	3	14
Error: ECONNRESET	/	/	/	/	/	30	/	/	29
HTTP-Code: 429	/	2713	15097	7198	7912	160672	/	8253	15254
HTTP-Code: 502	/	55	/	/	/	/	/	/	/
HTTP-Code: 503	/	9471	399	93	758	1743	/	4	3458

Tabelle 5.2: Instanzenminimum 1 - Ergebnisse der Stress Tests

1500 CONCURRENT VU: Die Laufzeit des Tests betrug 11 Minuten und 40 Sekunden. Die Antwortzeiten von den Anwendungsfällen A und C mit 1500 VU sind mit den Antwortzeiten der Pipe-Clean Tests vergleichbar. Die Ergebnisse der Tests weichen nur in den minimalen und maximalen Werten voneinander ab. Dazu erhielten die Tests nur 200 HTTP-Codes als Antwort. Dafür kam es jedoch zu Timeout also ETIMEDOUT, und Connection, also ECONNRESET, Fehlern. Diese Fehler tauchten jedoch bei Anwendungsfällen A und C mit ungefähr 300000 Anfragen pro Test nur zweimal auf. Statistisch haben diese Zahlen geringe Auswirkungen jedoch, sollten diese nicht unerwähnt bleiben.

Im Fall von Anwendungsfall B sind die Antwortzeiten, jedoch nicht mit den Ergebnissen der Pipe-Clean Tests vergleichbar. B hat durch den gewählten Testaufbau, die höchste Anzahl an Anfragen pro Test und Anfragen pro Sekunde, somit erzeugt der Anwendungsfall B die



größte Last mit der gleichen Anzahl an VU. Aus diesem Grund ist B als einziger Anwendungsfall auf eine zweite Instanz hochskaliert, was sich in der Anzahl an Fehler-Codes und den höheren Werten der oberen Quantilen der Antwortzeit erkennen lässt. Die Antwortzeiten in den oberen Quantilen wurden eindeutig durch die Skalierung beeinflusst. Für P<sub>95</sub> lagen die Werte zwischen 215 und 319 ms und für P<sub>99</sub> zwischen 575 und 703 ms. Ein großer Unterschied und eine höhere Varianz zu den vorher erfassten Werten von 95 ms für P<sub>95</sub> und 107 ms für P<sub>99</sub>. Des Weiteren erhielt Anwendungsfall B in diesem Testdurchlauf 200, 429 sowie 503 HTTP-Codes, dessen Bedeutung in 4.2.2 genauer erläutert werden. Die 429-Codes sind, wie dort beschrieben, ein Indiz dafür, dass die Instanz überlastet ist und auf eine weitere Instanz hochskaliert werden muss. Bei diesen Tests war durchschnittlich 1,4954% der Anfragen 429-Codes und 0,0301% 503-Codes. Insgesamt konnten somit bei Anwendungsfall B 1,5255% der Anfragen nicht bearbeitet werden. In der Abbildung 5.2 sind die Antwortzeiten von Fall

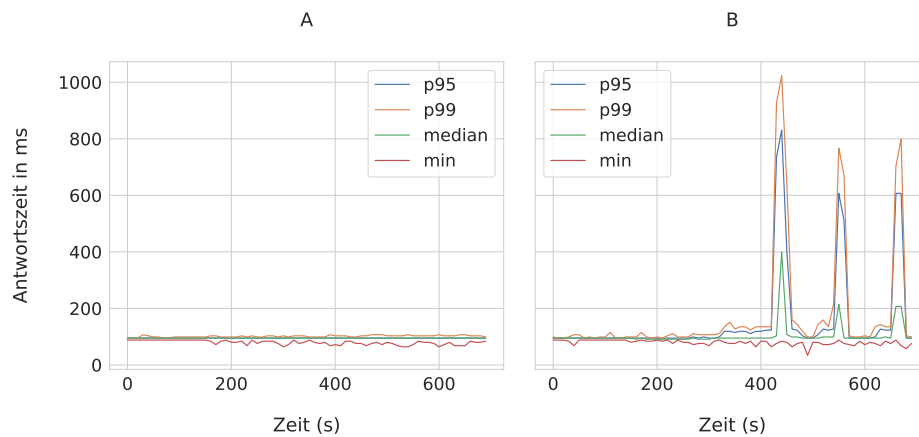


Abbildung 5.2: Antwortzeiten von A (links) und B (rechts) im Vergleich; Stress Test mit 1500 VU und Instanzenminimum 1

A und Fall B im Vergleich. Dort ist der Unterschied in der Antwortzeit klar zu erkennen, da Fall B mehrere Antwortzeit-Spitzen besitzt. Diese Spitzen sind die Zeitpunkte an denen der Skalierungsalgorithmus von AWS eine Skalierung auf zwei Instanzen gestartet hat. Hier sind drei Spitzen zu sehen, da zunächst auf zwei Instanzen hochskaliert wurde, jedoch kurz darauf wieder auf eine herunter. Dies änderte sich jedoch erneut gegen Ende des Tests, als erneut auf zwei Instanzen hochskaliert wurde. Der Auslöser für diese Art der Skalierung wurde nicht weiter untersucht, da es sich bei dem Skalierungsalgorithmus um einen Teil der nicht einsehbaren Black-Box handelt. Daher wurde nur die, aus diesem Verhalten entstandene, Leistung beobachtet. Im Vergleich sind links bei den Antwortzeiten von A keine Spitzen zu erkennen. Auf eine graphische Darstellung der Antwortzeiten für den Anwendungsfall C wurde verzichtet, da diese denen von A stark ähneln.



2000 CONCURRENT VU: Bei nächsten Testdurchlauf wurde die Anzahl gleichzeitig laufender VU auf 2000 erhöht. Dieser Test hatte durch die erhöhte Anzahl an Schritten beim Lastanstieg eine Testlaufzeit von 13 Minuten und 40 Sekunden. Bei diesem Testdurchlauf wurde die Anzahl an aktiven Instanzen bei allen Anwendungsfällen auf zwei erhöht. Was sich erneut in der Anzahl an Fehler-Codes und den Antwortzeiten wieder spiegelt.

Bei diesem Testdurchlauf war der Anwendungsfall A besonders fehlerhaft. Es wurden überdurchschnittlich schlechte Antwortzeiten und viele Fehler beobachtet. Dieses Verhalten zog sich durch alle 5 Testdurchläufe hindurch, weswegen die Ergebnisse dieser Tests weiterhin in der Darstellung bleiben. Den Ergebnissen der Tests wird jedoch nicht so viel Gewichtung zugeteilt wie den Ergebnissen der Fälle B und C, welche untereinander eine hohe Ähnlichkeit aufweisen sowie mit den Ergebnisse von B bei dem 1500VU Test. Wie in Tabellen 5.2 zu erkennen ist die mediane Antwortzeit wieder bei allen Tests bei ungefähr 95 ms. Für Fall B bewegt sich P<sub>95</sub> zwischen 215 und 319 ms und P<sub>99</sub> zwischen 575 und 671 ms. Für Fall C liegt P<sub>95</sub> zwischen 199 und 367 ms und P<sub>99</sub> zwischen 511 und 735 ms. Die Antwortzeiten haben sich für alle Anwendungsfälle in den Quantilen erhöht. In Abbildung 5.3 sind wieder Antwortzeitsspitzen zu erkennen, welche die Skalierung darstellen. Diesmal skalierte das System auf erst drei Instanzen um ungefähr 100 Sekunden danach auf zwei Instanzen herunter zu skalieren. Dieses Verhalten zeigte sich bei mehreren Testdurchläufen, wurde jedoch nicht weiter untersucht. Diesbezüglich wurde nur die Leistung beobachtet.

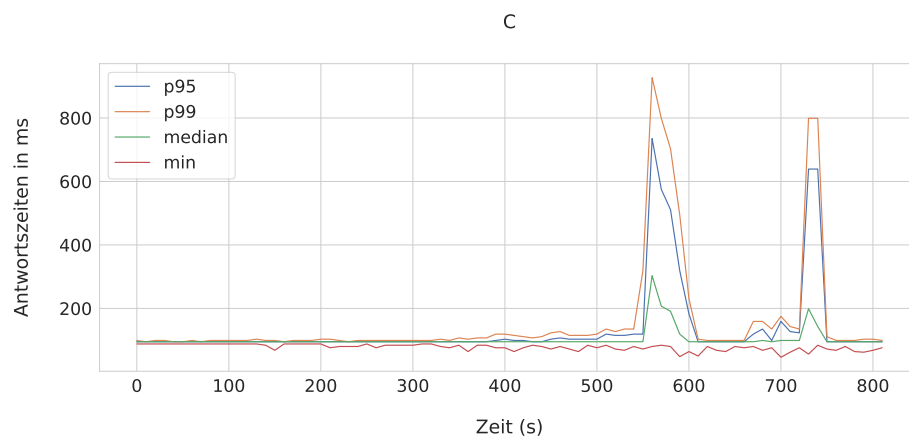


Abbildung 5.3: Antwortzeiten von C; Stress Test mit 2000 gleichzeitige VU und Instanzenminimum 1

Dazu sind die Fehler-Codes ebenfalls für alle Fälle deutlich gestiegen. Fall A hat 200, 429, 503 und 502 HTTP-Codes erhalten mit einer insgesamt durchschnittlichen Fehler-Code Rate von 3,6436% im Vergleich zu den Fällen B und C. Die Fehlerraten von B und C ähnelten eher der

Fehlerrate von B beim 1500VU Test, wie in Abbildung 5.4 erkennbar. B hat eine durchschnittliche Fehler-Code Rate von 1,7912% und Fall C hat durchschnittlich 1,5515% Fehler-Codes. Bei beiden Fällen haben die 429 Codes die Mehrheit der fehlerhaften Antworten gebildet, mit 1,6056% für Fall B und 1,2728% für C. Es kam zu keinen 502-Codes, sondern nur 503-Codes, welche 0,1856% für B und 0,3484% für C ausgemacht haben.

Zuletzt ist noch die Menge Timeout und Verbindungsfehler zu betrachten. Hierbei zeigt sich Anwendungsfall A erneut als ein Ausreißer, durch dessen hohe Anzahl. Es kam bei Fall A zu einer hohen Anzahl an Timeouts, ungefähr 0,2025% aller Anfragen erhielten statt einer Antwort einen Timeout-Error zurück. Bei Fall B waren es nur 0,0028% und bei Fall C 0,0215% aller Anfragen die einen Error erhielten.

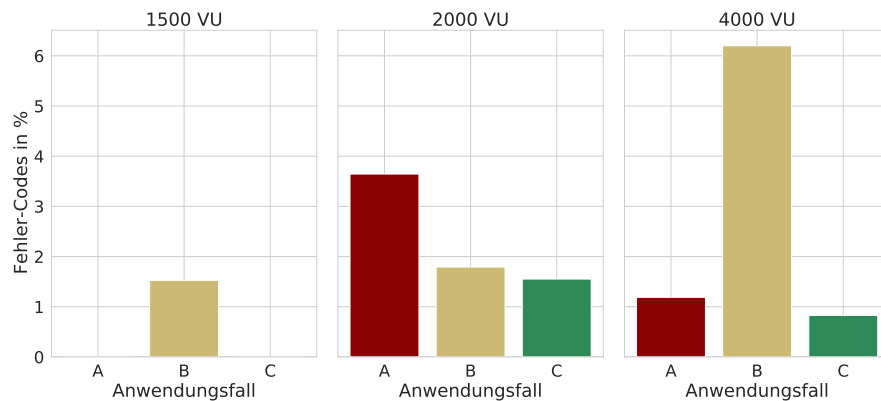


Abbildung 5.4: Fehler-Code-Quote der Stress Tests für Instanzenminimum 1

**4000 CONCURRENT VU:** Für die letzte Testreihe wird die Anzahl der VU auf 4000 gesetzt mit dem Hintergrund die eben entdeckte Grenze von 2000VU zu überschreiten und von zwei auf drei Instanzen zu skalieren. Die Laufzeit der Tests ging bis maximal 23 Minuten und 50 Sekunden. Da es sich bei der Laufzeit, um die Zeit handelt die benötigt wird bis alle Szenarien abgeschlossen sind, ist diese abhängig von den Antwortzeiten welche pro Test variieren können. Wie erwartet wurde bei allen Anwendungsfällen auf drei Instanzen hochskaliert. Auffällig ist bei diesem Test, dass die minimale Antwortzeit bei allen Anwendungsfällen im Gegensatz zu den vorherigen Minima sehr niedrig ist, mit Werten von 2 bis 16 ms. Als Ausgleich dazu sind die maximalen Antwortzeiten höher als in den Tests davor, mit 3376 bis 10995 ms.

In der Abbildung 5.5 sind die Antwortzeitenspitzen zu sehen. Hier wird wieder deutlich, dass der Skalierungsalgorithmus zu mehreren Zeitpunkten skalierte. Die erste Skalierung geschah bei knapp 1800 VU auf drei Instanzen. Danach kam es zu Skalierung nach oben und unten, wobei die meiste Zeit drei Instanzen aktiv waren. Grundsätzlich

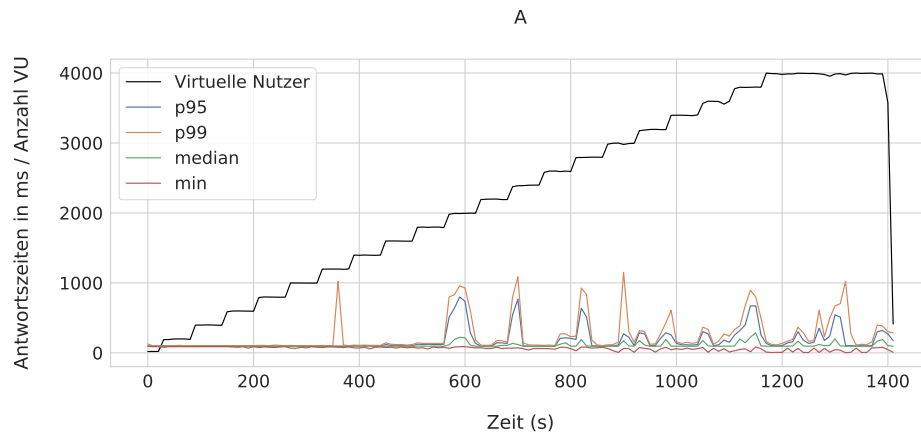


Abbildung 5.5: Antwortszeiten von A; Stress Test mit 4000 gleichzeitige VU und Instanzenminimum 1

ähneln die Quantile für Fälle A und C den Werten vom Test mit 2000VU, mit P<sub>95</sub> zwischen 231 und 287 ms für A und zwischen 271 und 287 ms für C. Im 99. Quantil lagen die Werte für A zwischen 415 und 639 ms, für Fall C waren es 511 bis 831 ms. Wie bei dem Test mit 1500VU sticht Anwendungsfall B hervor. Die minimalen, maximalen und die medianen Antwortszeiten sind vergleichbar mit Fällen A und C, jedoch sind die Quantile ein wenig höher. Das 95. Quantil liegt innerhalb der fünf durchgeführten Tests zwischen 367 und 2047 ms und das 99. Quantil zwischen 639 und 5119 ms, hier ist die Varianz deutlich höher.

Wie in Abbildung 5.4 zu sehen liegt die Fehler-Quote für Anwendungsfall B deutlich höher als bei anderen Tests. Insgesamt 6,1993% Prozent der Anfragen erhielten entweder einen 429-Code oder einen 503-Code als Antwort, wobei die 429-Codes 4,6214% dieser Fehler ausmachen und 503-Codes 1,5779%. Dazu gab es einen großen Anteil Timeout und Connection Fehler, 0,1295% aller abgeschickten Anfragen erhielten gar keine Antwort sondern wurden aus zeitlichen Gründen oder Serverseitig abgebrochen. Somit konnten insgesamt 6,3288% der Anfragen nicht beantwortet werden.

Bei den Fällen A und C sind die Fehlerwerte geringer, bei Fall A waren insgesamt 1,1868% aller Antworten entweder 429 oder 503-Codes. Fall C hatte sogar nur 0,8267% Fehler-Codes. Bei beiden Fällen war die Mehrzahl der Fehler 429-Codes mit 1,0504% für A und 0,6815% für C. Die 503-Codes waren für A bei 0,1713% und für C bei 0,1452%.

Alle Anwendungsfälle haben eine Skalierung auf drei Instanzen ausgelöst, jedoch lässt die Anzahl an Fehlern vermuten, dass Anwendungsfall B die Grenze von drei Instanzen überschritten hat. Das bedeutet, dass selbst drei Instanzen nicht ausgereicht haben um die Last die Fall B ausgelöst hat zu stemmen.

## 5.3.2 Last Test

Ziel dieser Tests ist es App Runner einem schnelleren Lastanstieg auszusetzen und die Veränderungen in der Leistung zu beobachten. Hierbei wurde nach einer Aufwärmphase von einer Minute, die Anzahl an VU innerhalb kurzer Zeit auf die maximale Anzahl VU des Tests erhöht. Nach dieser steilen Last Phase wurde die maximale Last einige Minuten lang gehalten. Hier wurde erneut keine Cool-Down Phase eingestellt.

Instanzenminimum 1	A		B		C	
Metrik \ Maximale VUs	2000	4000	2000	4000	2000	4000
Anfragen	469254	1126208	895569	2080989	663624	1423137
Durchschn. Anfragen/Sek.	548,94	1087,82	1047,16	2007,98	775,7	1370,01
Min. Antwortzeit (ms)	56	2	8	4	48	12
Max. Antwortzeit (ms)	3361	3786	3436	3764	11020	5935
Mediane Antwortzeit (ms)	95	95	95	115	95	95
95. Quantil (ms)	271	303	247	447	247	319
99. Quantil (ms)	639	671	575	703	607	703
Error: ETIMEDOUT	1	8	6	64	15	42
Error: ECONNRESET	/	/	8	33	10	10
HTTP-Code: 429	7134	18301	161	147885	26	23328
HTTP-Code: 503	43	2361	1439	433	219	5029

Tabelle 5.3: Instanzenminimum 1 - Ergebnisse der Last Tests

2000 CONCURRENT VU: Zunächst wurde die maximale VU Anzahl auf 2000 gesetzt, da in den Stress Tests erkennbar war, dass bei dieser Anzahl an VU alle Anwendungsfälle auf eine zweite Instanz hochskalieren mussten. Die Testlaufzeit betrug in allen Anwendungsfällen 14 Minuten und 10 Sekunden, wobei der Verlauf der ankommenden VU für jeden Anwendungsfall unterschiedlich war. Wie in der Abbildung 5.6 zu sehen

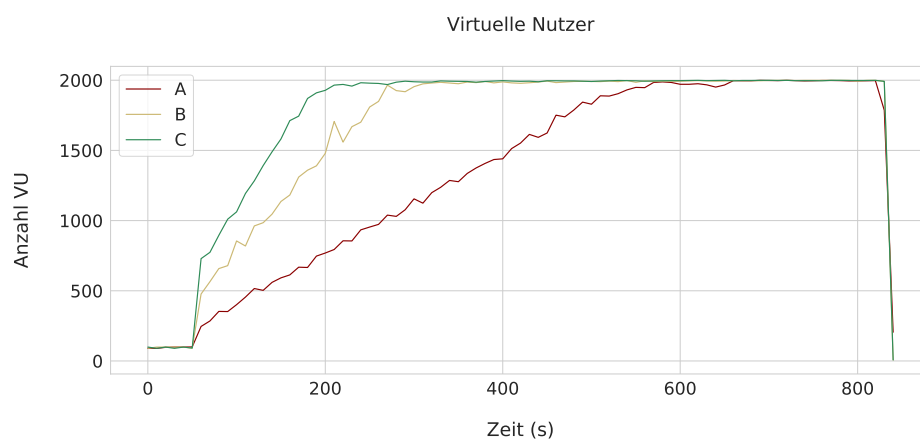


Abbildung 5.6: Lastanstieg eines Last Tests, 2000 gleichzeitige VU

ist der Verlauf der ankommenden VU bei Anwendungsfall A ähnlich

zum VU Verlauf der Stress Test (siehe Abb. 5.1). Bei Anwendungsfall A für diesen Last Test nimmt die Anzahl VU stetig zu und erreicht das Maximum bei ungefähr 550 Sekunden. Bei Anwendungsfall B ist die Kurve der ankommenden VU deutlich steiler als bei Anwendungsfall A. Zunächst gibt es eine kurze starke Steigung auf 500 VU, danach flacht die Steigung etwas ab und die Anzahl VU nimmt in einem linearen Verhalten zu bis bei Sekunde 230 das Maximum von 2000 erreicht wird. Die Kurve von Anwendungsfall C weist ähnliches Verhalten zu der Kurve von B aus. Direkt nach der Aufwärmphase kommt es zu einer starken Steigung auf ungefähr 700 VU. Danach steigt die Anzahl weiter in einem linearen Verhalten an bis es bei Sekunde 200, 2000 VU erreicht.

P<sub>95</sub> war bei allen Fällen zwischen 123 und 319 ms, wobei P<sub>99</sub> zwischen 495 und 767 ms liegt. Besonders auffällig ist jedoch die Anzahl an Fehler-Codes die erhalten wurden, wie in Abbildung 5.8 zu erkennen ist. Fall A hat insgesamt eine Fehler-Code Rate von 1,3507%, wobei 429-Codes 1,3173% der Fehler ausmachen und 503-Codes 0,0334%. Die Fehler-Code Raten sind für die Fälle B und C dazu noch besser als Fall A. Bei Fall B gab es insgesamt eine Fehler-Code Rate von 0,9155%, mit 0,8133% 429-Codes und 0,1022% 503-Codes. Für Fall C war es eine Fehler-Code Rate von nur 0,4215%, mit 0,3947% 429-Codes und 0,0267% 503-Codes. Die Anzahl an Timeout Fehlern ist in diesem Testdurchlauf erneut sehr gering, mit durchschnittlich 0,0005% für Anwendungsfall A, 0,0008% für Fall B und 0,0011% für Fall C.

4000 CONCURRENT VU: Als nächstes wurde die maximale Anzahl auf 4000 gesetzt, um das Grenzverhalten bei schnell ansteigender Last zu beobachten. Hierbei wurde bei jedem Anwendungsfall auf drei Instanzen hochskaliert. Die Testlaufzeit betrug in allen Anwendungsfällen 17 Minuten und 10 Sekunden, wobei jedoch der Verlauf der ankommenden VU für jeden Anwendungsfall unterschiedlich war. Der Verlauf des Lastanstiegs ist äquivalent zu dem Verlauf des Lastanstiegs pro Anwendungsfall des 2000VU Tests (siehe 5.6). Das Maximum wurde zu dem gleichen Zeitpunkt erreicht wie bei 2000VU, was bedeutet, dass die Steigung steiler und somit der Lastanstieg schneller war.

Bei diesem Test kam es wieder zu vielen Fehlern und einer Skalierung auf drei Instanzen, wobei Fall B, wie bei den Stress Tests, die höchste Fehlerrate und Antwortzeiten aufwies. Wie in Abbildung 5.7 erkennbar wiesen die Antwortzeiten häufig Spitzen auf und es kam oft zu Skalierungen nach oben und unten. Jedoch waren vor Allem gegen Ende drei Instanzen aktiv. Dennoch kam es zu erhöhten Antwortzeiten und einer erhöhten Anzahl an Fehlern, durch die Überlastung der Instanzen. Die minimale Antwortzeit ist, wie beim Stress Test mit 4000VU, in einem sehr niedrigen Bereich, zwischen 4 und 12 ms. Der Median ist erneut für A und C bei 95 ms. Für Fall B hat sich dieser auf 115 ms verlangsamt. Die Antwortzeiten von A und C sind für das 95.

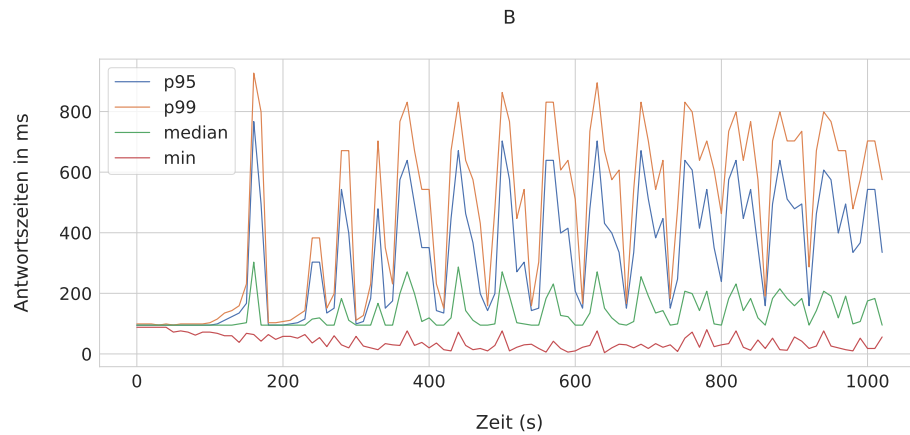


Abbildung 5.7: Antwortszeiten von B; Last Test mit 4000 VU und Instanzenminimum 1

Quantil zwischen 271 und 399 ms, für B liegt es zwischen 399 und 495 ms. Im Vergleich zu dem 4000VU Stress Test sind die Antwortszeiten von A und C um die 100 ms langsamer geworden. Für Fall B sind die Werte in diesem Test besser, da es weniger Varianz in den Antwortszeiten gibt. Ähnlich ist es auch in dem 99. Quantil. Für A und C ist das 99. Quantil zwischen 607 und 703 ms, wobei B sich im Bereich zwischen 703 und 767 ms befindet. Die Werte weisen im direkten Vergleich zu dem 4000VU Stress Tests eine niedrigere Varianz auf und bewegen sich in einem ähnlichen Wertebereich. In beiden 4000 VU Tests ist klar erkennbar das Anwendungsfall B langsamere Antwortszeiten hat als die anderen Anwendungsfälle. Wie in der Abbildung 5.8 zu sehen hat B

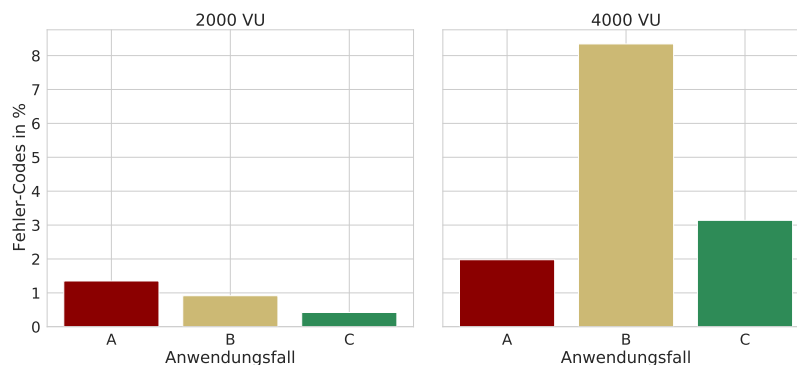


Abbildung 5.8: Fehler-Code-Quote der Last Tests für Instanzenminimum 1

erneut die höchste Fehler-Code Rate von allen Anwendungsfällen, mit einer Fehlerrate von insgesamt 8,3434%. Hierbei machen 429-Codes 8,1555% dieser Fehler aus und die 503-Codes 0,1879%. A und C hingegen haben eine Fehlerrate von 1,9774% und 3,1398% respektive, auch hier machen die 429-Codes die meisten Fehler aus. Verglichen mit dem

4000 **VU** Stress Test sind die Fehler Werte deutlich schlechter trotz gleicher Anzahl an **VU** und ähnlichem Skalierungsverhalten.

#### 5.4 INSTANZENMINIMUM VON 2

Nun wird die Konfiguration mit zwei minimalen bereitstehenden Instanzen getestet. Es werden wieder die Stress und Last Tests für diese Konfiguration durchgeführt.

##### 5.4.1 Stress Test

Für diese Stress Tests werden erneut die Einstellungen aus Abschnitt 5.1 genutzt. Bei diesem Stress Test ist das Ziel die Leistung von zwei Instanzen zu bestimmen. Dazu soll die Leistung beobachtet werden wenn eine zweite bereitstehende Instanz aktiviert wird.

Instanzenminimum 2	A			B			C		
Metrik \Maximale VUs	1500	2000	4000	1500	2000	4000	1500	2000	4000
Anfragen	293909	428649	1342712	455617	667002	2071409	321081	466052	1027,26
Durchschn. Anfragen/Sek.	420,36	521,43	942,49	645,76	809,23	1454,02	455,6	565,11	1027,26
Min. Antwortzeit (ms)	46	50	4	54	40	4	52	48	4
Max. Antwortzeit (ms)	1931	3518	6791	1814	4112	3969	1752	1895	7943
Mediane Antwortzeit (ms)	95	91	95	95	95	95	95	95	95
95. Quantil (ms)	95	143	287	183	191	367	95	207	303
99. Quantil (ms)	103	447	607	463	431	639	103	575	607
Error: ETIMEDOUT	/	1	22	3	10	58	/	/	26
Error: ECONNRESET	/	/	/	10	9	8	/	9	11
HTTP-Code: 429	/	41	25307	195	32	64013	/	21	26612
HTTP-Code: 503	/	/	136	/	175	1000	/	51	115

Tabelle 5.4: Instanzenminimum 2 - Ergebnisse der Stress Tests

1500 CONCURRENT VU: Zuerst wird der Test mit 1500 **VU** und einer Testlaufzeit von 11 Minuten und 40 Sekunden durchgeführt. Nur im Anwendungsfall B wurde auf eine zweite Instanz skaliert. Dies spiegelte sich in den Fehler-Codes und langsameren Antwortzeiten wieder (siehe 5.4. Die Fälle A und C weisen ähnliche Werte zu den Pipe-Clean Tests auf und wurden erfolgreich bearbeitet. Es wurden nur 200-Codes erhalten. Für Fall B sind die Ergebnisse anders. Die minimale, maximale und mediane Antwortzeit ist denen von den Fällen A und C sehr ähnlich, jedoch ist das 95. und 99. Quantil langsamer. Die Werte für das 95. Quantil befinden sich zwischen 119 und 215 und für das 99. Quantil zwischen 415 und 607 ms. Die Anzahl an Fehler-Codes liegt für Fall B durchschnittlich bei einer Fehler-Code Rate von 0,0435%, wobei 429-Codes 0,0151% ausmachen und 503-Codes 0,0284%.

In der Abbildung 5.9 sind zwei Antwortsspitzen zu erkennen. Die erste Spitze ist die erste Skalierung, wobei hier auf drei Instanzen skaliert wurde, um dann 50 Sekunden später auf zwei Instanzen herunter zu

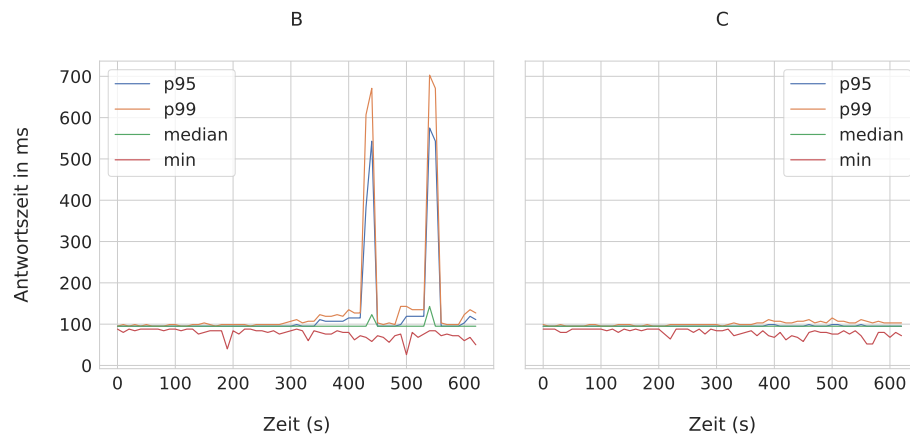


Abbildung 5.9: Antwortzeiten von B (links) und C (rechts) im Vergleich; Stress Test mit 1500 VU und Instanzenminimum 2

skalieren. Hier ist zu erkennen, dass das 95. Quantil und das 99. Quantil jeweils knapp 100 ms schneller sind als mit einer einzigen bereitstehenden Instanz, wie in Abbildung 5.9 zu sehen. Im Vergleich sind rechts bei den Antwortzeiten von C keine Spitzen zu erkennen. Auf eine graphische Darstellung der Antwortzeiten für den Anwendungsfall A wurde verzichtet, da diese denen von C stark ähneln.

2000 CONCURRENT VU: Als nächstes wird der 2000VU Test durchgeführt. Hier betrug die Testlaufzeit 13 Minuten und 40 Sekunden und es wurde bei allen Anwendungsfällen auf eine zweite Instanz hochskaliert. Alle Anwendungsfälle hatten ähnliche Antwortzeiten. Die minimale Antwortzeit befindet sich in allen Fällen zwischen 18 und 60 ms, die maximale zwischen 1895 und 4112 ms und der Median ist bei Fall A auf 91 ms und bei B und C auf 95 ms. Das 95. Quantil ist zwischen 123 und 223 ms und das 99. Quantil zwischen 399 und 607 ms.

Das 95. Quantil deckt einen kleinen und schnellen Zeitbereich. Im 99. Quantil verhält es sich ähnlich, die Varianz der Werte ist niedrig und bewegt sich in einem kleinen ms Bereich. In Abbildung 5.10 sind die Antwortzeiten von Anwendungsfall C zu sehen. Hier sind drei Spitzen erkennbar, da zuerst auf zwei, dann auf eine dritte und zuletzt zurück auf zwei Instanzen skaliert wurde.

Die Fehler-Code Rate dieses Tests ist die Beste der bisherigen Tests. Anwendungsfall A hat durchschnittlich pro Test 0,0071% Fehler-Codes erhalten, wovon 0,0057% 429-Codes waren und 0,0014% 503-Codes. Für Fall B war die Fehler-Code Rate etwas höher, jedoch immer noch geringer als vorherige Tests, mit durchschnittlich 0,1362% Fehler-Codes. Zuletzt hat Fall C noch eine durchschnittliche Fehler-Code Quote von 0,0237%.

Das Instanzenminimum 2 ist auf eine zweite Instanz hochskaliert, ist jedoch nicht auf die Grenzen gestoßen die man in den Stress Test der



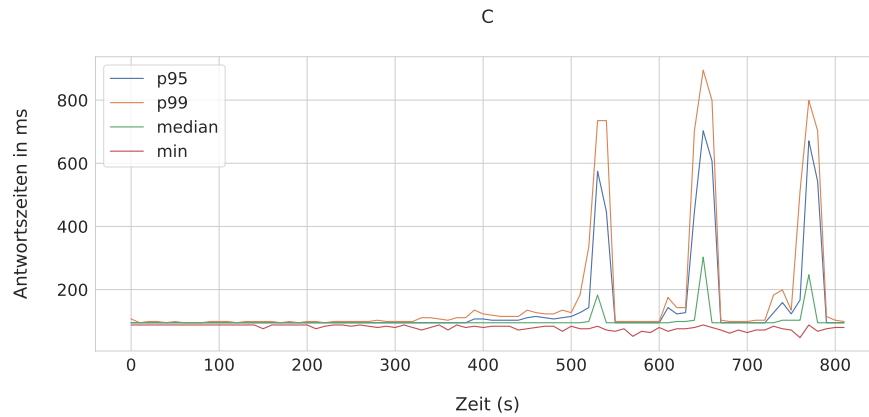


Abbildung 5.10: Antwortszeiten von C; Stress Test mit 2000 gleichzeitige VU und Instanzenminimum 2

Konfiguration mit einen Instanzenminimum von eins sehen konnte. Das zeigt sich auch in der Anzahl an Timeout Fehlern. Die Menge an Fehlern ist mit der Anzahl der Anfragen nicht mehr vergleichbar und es handelt sich um die 0,0008% pro Test.

4000 CONCURRENT VU: Hier wurde die maximale Anzahl an VU wieder auf 4000 gesetzt. Die Testlaufzeit betrug erneut maximal 23 Minuten und 40 Sekunden und es wurden alle drei Instanzen aktiviert. Hier ist wieder zu erkennen das die minimale Zeit verglichen mit den Tests davor besonders niedrig ist. Diesmal in einem Bereich zwischen 2 und 14 ms. Die mediane Antwortzeit ist wieder einheitlich bei allen Anwendungsfällen bei 95 ms. Die oberen Quantile bewegen sich bei jedem der Anwendungsfälle einheitlich in einem ähnlichen Bereich. Für die Fälle A und C liegt das P<sub>95</sub> zwischen 231 und 319 ms, und für das P<sub>99</sub> zwischen 447 und 735 ms. Anwendungsfall B hat etwas langsamere Antwortzeiten mit 367 bis 431 ms für P<sub>95</sub> und 639 bis 735 ms für P<sub>99</sub>.

In Abbildung 5.11 sind die Antwortszeiten des Anwendungsfalles A zu sehen. Hier ist erkennbar, dass bei ungefähr 2000 VU die erste Skalierung stattfand. Im weiteren Verlauf des Tests gab es mehrere Spitzen an denen die Antwortzeiten kurzzeitig stark zunahmen. An diesen Spitzen hat der Skalierungsalgorithmus erneut auf Instanzen hoch oder herunter skaliert, wobei hauptsächlich drei Instanzen aktiv waren.

Die Fehler Menge ist wie erwartet relativ hoch, durch die Skalierung auf drei Instanzen. Bei Anwendungsfall A handelt es sich um insgesamt 1,314% Fehler-Codes und bei Anwendungsfall C sind es insgesamt 1,7515% Fehler-Codes. Beide Fälle hatten hauptsächlich 429-Codes. Anwendungsfall B hat wiederum eine besonders hohe Fehlerquote mit 5,6801% insgesamt, hierbei lag die Mehrheit wieder bei den 429-Codes mit 5,6234%. Es kam erneut zu Timeouts und Connection Errors, jedoch handelt es sich hierbei wieder um niedrige Werte von maximal 0,018%.

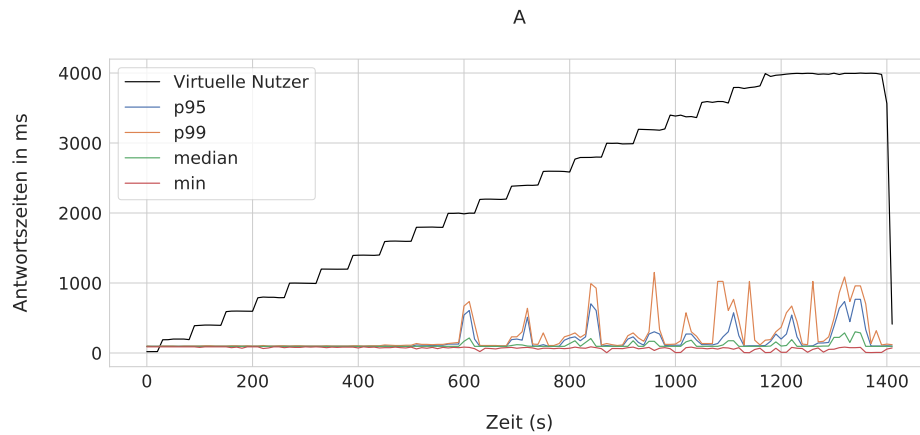


Abbildung 5.11: Antwortszeiten von A; Stress Test mit 4000 VU und Instanzenminimum 2

#### 5.4.2 Last Test

Hier werden nun die Last Tests durchgeführt. Der Verlauf der ansteigenden VU verläuft mit einer Aufwärmphase und einer schnell ansteigenden Last die mit einer Phase endet in der die maximale Last einige Minuten gehalten werden muss.

Instanzenminimum 2	A		B		C	
Metrik \ Maximale VUs	2000	4000	2000	4000	2000	4000
Anfragen	472028	1128915	897241	2091211	663859	1578700
Durchschn. Anfragen/Sek.	551,39	1090,23	1050,06	2020,07	775,59	1524,59
Min. Antwortzeit (ms)	54	8	28	6	44	8
Max. Antwortzeit (ms)	1964	3847	7589	3922	3682	4248
Mediane Antwortzeit (ms)	95	95	95	103	95	95
95. Quantil (ms)	151	271	223	463	247	335
99. Quantil (ms)	543	575	399	703	639	639
Error: ETIMEDOUT	1	26	16	188	6	49
Error: ECONNRESET	1	/	29	18	16	28
HTTP-Code: 429	8	653	167	171389	120	21055
HTTP-Code: 503	7	791	312	58	78	2709

Tabelle 5.5: Instanzenminimum 2 - Ergebnisse der Last Tests

**2000 CONCURRENT VU:** Zunächst wird zuerst der Last Test mit einem Maximum von 2000 VU durchgeführt. Die Testlaufzeit beträgt in allen Anwendungsfällen 14 Minuten und 10 Sekunden, wobei der Verlauf der ankommenden VU für jeden Anwendungsfall unterschiedlich ist. (siehe Abbildung 5.6)

Die Antwortzeiten für Anwendungsfall A sind bei diesem Test schnell. Für P<sub>95</sub> waren es zwischen 107 und 151 ms und für P<sub>99</sub> waren es zwischen 119 und 543 ms. Anwendungsfall B war einheitlich bei P<sub>95</sub> mit

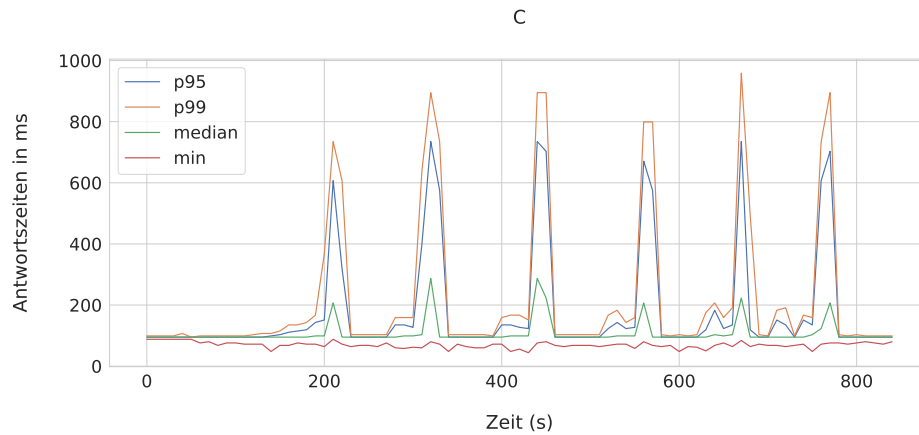


Abbildung 5.12: Antwortzeiten von C; Last Test mit 2000 VU und Instanzenminimum 2

207 bis 223 ms und P99 mit 319 bis 511 ms. Die Varianz der Werte ist nicht hoch und bewegt sich im selben Bereich, wie bei dem Last Test mit einer minimalen Instanz. Für Anwendungsfall C waren die Quantil-Werte für P95 zwischen 239 und 271 ms und für P99 zwischen 607 und 639 ms. Fall C hat somit die langsamsten Werte, befindet sich aber immer noch im Bereich der 2000 VU Last Tests mit einer minimalen Instanz.

In Abbildung 5.12 sind die Antwortzeiten von Anwendungsfall C zu sehen. Hier sind viele Antwortsspitzen erkennbar, die erste bei Sekunde 200, dem Zeitpunkt an dem auf die maximale Anzahl VU erreicht wurde. Die darauffolgenden Skalierungen kamen in einem regelmäßigen Abstand, wobei hier der Skalierungsalgorithmus erneut von zwei herunter auf eine und wieder auf zwei Instanzen skalierte.

Die Fehler-Code-Quote ist hier wieder niedrig, mit insgesamt 0,0147% Fehler-Codes für Fall A, 0,0519% für Fall B und 0,0315% für Fall C. Es gab auch hier wieder Timeout Fehler, wobei kein Anwendungsfall eine höhere Quote hatte als 0,002%.

4000 CONCURRENT VU: Zuletzt wurde die maximale Anzahl auf 4000 gesetzt, um das Grenzverhalten bei schnell ansteigender Last zu beobachten. Hierbei wurde bei jedem Anwendungsfalls auf drei Instanzen hochskaliert. Die Testlaufzeit betrug in allen Anwendungsfällen 17 Minuten und 10 Sekunden. Der Verlauf der ankommenden VU erfolgte wie für die Konfiguration mit einer minimalen Instanz (siehe Abbildung 5.6).

Hier ist wieder zu beobachten, dass die minimale Antwortzeit besonders niedrig ist, zwischen 2 und 20 ms. Der Median ist wie immer bei 95 ms, außer bei Fall B. Dort bewegte sich der Median bei den 5 Testdurchläufen zwischen 103 und 167 ms. Die Quantile der Antwortzeiten sind bei Fall A bei P95 zwischen 247 und 303 ms und für P99

zwischen 495 und 639 ms. Anwendungsfall B bewegt sich bei  $P_{95}$  zwischen 447 und 543 ms und für  $P_{99}$  zwischen 703 und 767 ms. Zuletzt sind bei Fall C die Quantile wie folgt aufgeteilt.  $P_{95}$  bewegt sich in einem Bereich von 319 bis 367 ms und  $P_{99}$  von 607 bis 703 ms.

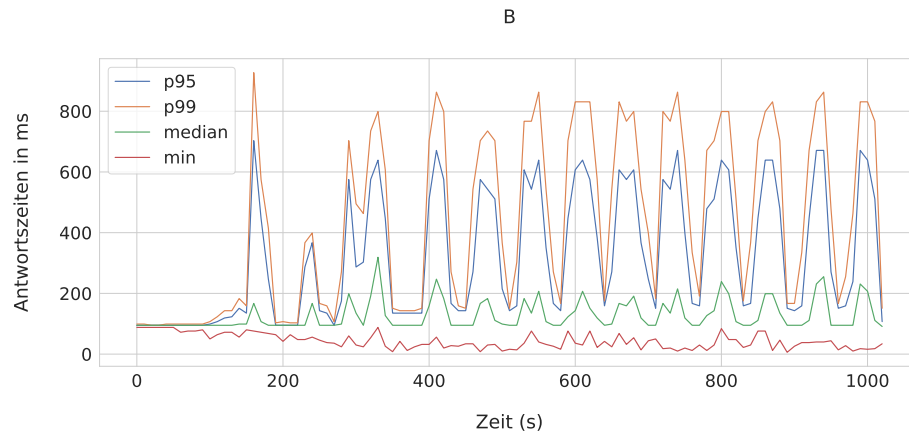


Abbildung 5.13: Antwortszeiten von B; Last Test mit 4000 gleichzeitige  $VU$  und Instanzenminimum 2

In Abbildung 5.13 sind die Antwortszeiten von Anwendungsfall B zu sehen. Hier erkennt man die vielen Antwortszeitspitzen, die die Quantil-Werte der Antwortszeiten in die Höhe getrieben haben. Hierbei handelt es sich um eine frühe Skalierung auf drei Instanzen, wobei hier der Skalierungsalgorithmus mehrere Male hoch und wieder herunter skaliert hat.

Die Fehler-Quoten waren für Fall B und C überdurchschnittlich hoch. Bei B waren insgesamt 9,7389% aller Antworten Fehler-Codes, für Fall C lag dieser Wert bei 2,1821%. Nur Anwendungsfall A hat eine geringe Fehler-Quote mit 0,7886% Fehler insgesamt. Hier waren die Mehrheit aller Fehler-Codes 429-Codes. Anhand dieser Fehler-Mengen kann behauptet werden, dass die Anwendungsfälle B und C an die Grenzen des Instanzenmaximums gestoßen sind.

## ANALYSE DER ERGEBNISSE

---

Dieses Kapitel beschäftigt sich mit der Analyse der in Kapitel 5 durchgeführten Tests. Wobei besonders auf die unterschiedlichen Ergebnisse der beiden Instanzen Minima eingegangen wird, sowie die Unterschiede in Leistung bei schrittweisem Lastanstieg und schnellem Lastanstieg. Anschließend werden die Kosten aller Konfigurationen miteinander verglichen. Das Kapitel schließt mit den Implikationen die diese Analyse für die Praxis darstellen ab.

### 6.1 ANALYSE DER LEISTUNGSASPEKTE

Zuerst werden die Leistungsaspekte definiert, die als Basis für diese Analyse dienen werden. Es gibt verschieden Aspekte die die Leistung eines Services beschreiben können, darunter die Verfügbarkeit des Dienstes, die Antwortzeiten und der Durchsatz der Anfragen, sowie die Nutzung der Ressourcen. [11] (Seite 2-3)

Durch den Aufbau der Tests werden in der kommenden Analyse vor allem die Aspekte der Verfügbarkeit und der Antwortzeit betrachtet.

#### 6.1.1 Verfügbarkeit

Die Menge fehlgeschlagener Anfragen dient als Index für die Verfügbarkeit, da bei fehlschlagenden Anfragen, dem Nutzer der Dienst nicht zur Verfügung steht. Diese fehlende Verfügbarkeit wurde hauptsächlich während einer Skalierung und Ausreizung des Instanzenmaximums beobachtet. Vergleicht man die Ergebnisse der Pipe-Clean Tests mit Tests an denen eine Skalierung stattfand, zeigt sich, dass mehr Fehler ausgelöst wurden. Diese Fehler waren einerseits Fehler-Codes, sowie 429, 502 und 503-Codes, andererseits Timeouts und Verbindungsfehler mit dem Server.

Wie in 4.2.2 beschrieben, hat ein Timeout (ETIMEDOUT) Fehler den Hintergrund, dass die Antwort des Servers länger als 10 Sekunden benötigt hat. Ein Verbindungsfehler (ECONNRESET) ist das Resultat einer plötzlichen Verbindungsstörung mit dem App Runner Server. Beide dieser Fehlertypen bildeten in den meisten Testdurchläufen beider Konfigurationen von Instanzenminima eine geringe Menge, von unter 0,006%. Zwei Fälle fielen nicht in diesen Wert hinein. Bei Anwendungsfall A in dem Stress Test mit 2000 VU mit einer minimalen Instanz, kam es zu höheren Prozentsätzen. Hier ist jedoch zu bedenken, dass dieser Testfall überdurchschnittliche Antwortzeiten und Fehler-Codes aufwies, weswegen es sich wahrscheinlich um einen Ausreißer handelt. Der zweite Fall bei dem es zu dem absoluten Maximum dieser Fehlertypen kam, war der Stress Test mit 4000 VU mit einer minima-

len Instanz. Bei diesem Testdurchlauf wurden auf das Instanzenmaximum hochskaliert und dieses ausgereizt. Hier wurden bei Fall B, 0,13% der Anfragen mit Timeouts beantwortet. Da es sich bei diesem Fall B um einen Extremfall handelt und trotzdem ein geringer Prozentsatz entstand, kann grundsätzlich davon ausgegangen werden, dass diese Arten Fehler kein ausschlaggebendes Verfügbarkeitsproblem darstellen.

Anders verhält es sich jedoch mit den Fehler-Codes. Die prozentuale Menge an Fehler-Codes übersteigt deutlich die Menge an Timeout Fehlern. Hier war zu beobachten, dass das Erreichen der „maximum Concurrency“ einer Instanz eine hohe Menge an 429- und 503-Codes auslöst, wobei die 429-Codes immer die Mehrheit bildeten. Sobald die „maximum Concurrency“ erreicht worden ist kann die Instanz keine weiteren Anfragen beantworten und der Skalierungsalgorithmus beginnt eine weitere Instanz zu aktivieren. Wie in 2.3.2 beschrieben kann es dabei zu Kaltstarts kommen. Hierbei wird zuerst eine Warteschlange erstellt in denen Anfragen darauf warten von einer aktivierten Instanz bearbeitet zu werden. Werden diese Anfragen nicht rechtzeitig beantwortet so kommt es zu einem Kaltstart, also zu Fehler-Codes.

Zunächst betrachten wir die Menge Fehler-Codes für die Konfiguration mit einer minimalen Instanz. Bei den meisten Anwendungsfällen der Stress Tests bewegten sich die Fehler für die Skalierung auf eine weitere Instanz zwischen 1,53% und 1,79%. Für die Skalierung auf zwei weitere Instanzen lagen zwischen 0,83% und 1,19%. Zwei Fälle fielen hier wieder aus der Reihe, Fall A mit 2000 VU und Fall B mit 4000 VU. Da es sich bei Fall A um einen Ausreißer handelt ist die Fehler Quote von 3,64% nicht ausschlaggebend. Bei Fall B handelt es sich um den Extremfall bei dem die dritte Instanz ebenfalls ausgereizt wurde und es zu einer weiteren Skalierung hätte kommen müssen, um die Last zu stemmen. Aus diesem Grund kam es zu einer sehr hohen Fehler Quote von 6,33%. Somit ist grundsätzlich mit einer Fehler Quote zwischen 0,83% und 1,79% zu rechnen. Vergleicht man diese Werte mit den Fehlern der Last Test mit einer minimalen Instanz ist interessant, dass bei dem Test mit 2000 VU der Bereich zwischen 0,42% und 1,35% liegt. Wobei je schneller die Last anstieg, desto geringer die Fehler-Quote war. Für den Test mit 4000 VU liegt dieser Bereich deutlich höher, zwischen 1,98% für Fall A und 3,14% für Fall C. Hier sprengt Fall B wieder den Rahmen mit insgesamt 8,16%. Hier ist zu beachten, dass wahrscheinlich alle Anwendungsfälle die maximale Last von drei Instanzen überschritten haben.

Letztendlich ist anzunehmen, dass bei einem stetigen Lastanstieg eine längere Belastung bessere Ergebnisse erzielt. Die Verfügbarkeit pendelt sich über eine längere Zeit hinweg ein. Dazu gilt, dass der Skalierungsalgorithmus auf schnelle Last vorteilhafter reagiert als auf eine stetige. Je schneller die Last ansteigt desto geringer ist die Menge an Fehlern, somit steigt die Verfügbarkeit. Dies gilt solange die maximale Last des Instanzenmaximums noch nicht erreicht wurde.

Als nächstes werden die Anzahl Fehler-Codes mit einem Instanzenminimum von zwei betrachtet. Hier sind, wie zu erwarten, die Fehler-Codes für die Stress Tests von 1500 und 2000 VU geringer, da auf zwei aktive Instanzen

skaliert wurde ohne Kaltstart. Jedoch kam es immer noch zu Fehler-Codes, vor allem bei beiden Anwendungsfällen B. Fall B bei 1500 VU hatte eine Fehler-Quote von 0,04%, deutlich höher als die Fehler-Quoten von A und C während des Tests mit 2000 VU. Anwendungsfall A hatte nur 0,007% Fehler und C nur 0,024%. Dafür hatte Anwendungsfall B bei 2000 VU sogar 0,14%. Dies sind deutlich bessere Werte als bei einem Instanzenminimum von 1, jedoch ist zu beobachten, dass die Aktivierung einer bereitstehenden Instanz nicht ohne Fehler geschieht. Eine bereitstehende Instanz muss somit erst warm werden, bevor sie die gleiche Last einer bereits aktiven Instanz übernehmen kann. Dafür ist bei dem 4000 VU Stress Test anhand der Fehler Quote zu beobachten, dass auf eine weitere Instanz hochskaliert wurde, die nicht bereitstand. Die Fehler Anzahl für diesen Test stieg auf 1,31% für Fall A und 1,75% für Fall C. Anwendungsfall B ist durch die Überlastung der drei Instanzen bei 5,69%. Diese Fehler Quote befindet sich im gleichen Bereich wie die Stress Tests bei der ebenfalls auf eine weitere Instanz die nicht bereitstand hochskaliert werden musste. Vergleicht man diese Werte mit den Werten der Last Tests erkennt man Ähnlichkeiten zu den Erkenntnissen zur Verfügbarkeit des Instanzenminimum von 1. Der Last Test mit 2000 VU hat eine Fehler Quote von 0,015% bis 0,052%. Die Aktivierung der zweiten bereitstehenden Instanz verlief besser als bei den Stress Tests. Bei dem Last Test mit 4000 VU bewegen sich die Fehler Prozente zwischen 0,79% für Anwendungsfall A, 2,18% für Fall C bis hin zu 9,74% für Anwendungsfall B. Wobei hier erneut alle Anwendungsfälle die maximale Last des Instanzenmaximums überschritten haben.

Diese Ergebnisse stimmen mit der Erkenntnis die für das Instanzenminimum 1 aufgestellt wurden überein. Die Skalierung auf eine weitere Instanz bei dem Stress Test mit 4000 VU hat eine schlechtere Verfügbarkeit, da nur einmal hochskaliert wurde in einem kurzen Zeitraum. Sonst ist noch erkennbar geworden, dass das Aktivieren einer Instanz eine gewisse Anzahl Fehler mit sich bringt und das ein schneller Lastanstieg erneut bessere Ergebnisse erzielt hat als ein stetiger.

Grundsätzlich ist also die Verfügbarkeit bei einer Skalierung von der Art, aber auch von der Länge der Belastung abhängig. Eine lange Belastung mit einem schnellen Lastanstieg, bietet den Nutzern die beste Verfügbarkeit. Dazu erhöht eine hohe Anzahl bereitstehender Instanzen die Verfügbarkeit des Dienstes. Kommt es jedoch zu kurzer und stetiger Belastung kann es zu einer Fehlerrate von bis zu 1,8% kommen. Zuletzt ist die Verfügbarkeit bei der Überlastung des Instanzenmaximums, besonders schlecht, mit einer Fehlerrate bis zu 9,74%.

### 6.1.2 Antwortzeiten

Nun werden die Antwortzeiten der verschiedenen Instanzenminima und Lastanstiege analysiert. Damit soll der klassische Aspekt der Leistung untersucht werden, wie lange der Dienst benötigt um Anfragen abzuarbeiten und wie gut dies unter steigender Last geschieht. Hierbei wird deutlich,



dass die Skalierung nach oben verschlechterte Antwortzeiten auslöst. Diese Verzögerung in den Antwortzeiten wird vor allem in den oberen Quantilen deutlich. Da es sich bei dem Skalierungsalgorithmus um eine Black Box handelt kann diese nur geringfügig beeinflusst werden um eventuelle Probleme mit den Antwortzeiten beheben zu können. Zunächst dienen die Ergebnisse der Pipe-Clean Tests als Basiswerte. Die Antwortzeiten der Pipe-Clean Tests bewegten sich, ohne Betrachtung der maximalen Antwortzeit, insgesamt zwischen 88 und 111 ms, somit in einem Wertebereich von 33 ms. Die mediane Antwortzeit lag hierbei zwischen 92,5 und 95 ms und das P<sub>95</sub> zwischen 94 und 95 ms. Das P<sub>99</sub> zwischen 99 und 111 ms.

Diese Werte werden, sobald es zu einer Skalierung auf eine nicht bereitstehende Instanz kommt, langsamer. Diese Art Skalierung wurde in mehreren Testfällen festgestellt, nämlich bei den Stress Tests mit 1500 und 2000 VU, sowie bei dem Last Test mit 2000 VU mit der Konfiguration von einer minimal bereitgestellten Instanz. Die mediane Antwortzeit verändert sich bei diesen Fällen im Grunde nicht und liegt weiterhin bei 95 ms. Dafür nehmen die Werte für P<sub>95</sub> und P<sub>99</sub> deutlich zu und die Unterschiede in den Antwortzeiten steigen. Hierbei muss jedoch erneut unterschieden werden zwischen einem schnellen und einem stetigen Lastanstieg, also zwischen den Ergebnisse der Last und der Stress Tests.

Ein stetiger Lastanstieg, also der Lastanstieg im Rahmen des Stress Tests, führte zu einer Erhöhung der Antwortzeiten im P<sub>95</sub> auf einen Bereich von 215 bis 319ms für die Anwendungsfälle B der 1500 und 2000 VU Tests. P<sub>95</sub> für Anwendungsfall C während des 2000 VU Tests liegt zwischen 199 und 367ms, in einem ähnlichen Bereich, jedoch mit höherer Varianz. Innerhalb von P<sub>95</sub> kann also mit einem Unterschied in Antwortzeiten von bis zu 168 ms gerechnet werden. Bei P<sub>99</sub> verhält es sich ähnlich, Anwendungsfall C mit 2000 VU weist die höchste Abweichung in den Antwortzeiten auf. Mit Werten zwischen 511 und 735 ms, handelt es sich hierbei um einen Wertebereich von 224 ms. Bei den Anwendungsfällen B hingegen handelt es sich um Werte zwischen 575 und 671 ms, also einer Abweichung von 96 ms. Vergleicht man diese Werte mit den Antwortzeiten der Pipe-Clean Tests sieht man, dass der Median identisch geblieben ist. Die oberen Quantile der Antwortzeiten haben jedoch im schlechtesten Fall, für P<sub>95</sub> bis zu 272 ms und für P<sub>99</sub> bis zu 624 ms, also mehr als eine halbe Sekunde, zugenommen. Unter Betrachtung der Abweichungen handelt es sich im Besten Fall um eine Verschlechterung der Antwortzeit von 104 ms für P<sub>95</sub> und 400 ms für P<sub>99</sub>.

Der schnelle Lastanstieg wurde erreicht mit der Testkonfiguration der Last Tests. Hier wurden folgende Änderungen in den Quantilen beobachtet. Bei allen Anwendungsfällen des Last Tests mit 2000 VU lagen die oberen Quantile in einem ähnlichen Bereich. Anwendungsfall A hat ein P<sub>95</sub> zwischen 143 und 319 ms. Fall B bewegt sich ein wenig darunter mit 123 bis 287 ms und einer ähnlichen Varianz. Anwendungsfall C hat für P<sub>95</sub> Werte zwischen 183 und 287 ms und der geringsten Varianz. Die maximale Abweichung für P<sub>95</sub> liegt somit bei 196 ms. Für P<sub>99</sub> weichen die Werte zwischen den Anwendungsfällen etwas ab. Die Antwortzeiten für Fall A bewegen sich in einem



Bereich zwischen 543 und 767 ms, ähnlich zu Fall C dessen Werte zwischen 495 und 639 ms liegen. Die Werte von Fall C sind jedoch kleiner und besitzen eine geringere Varianz. Bei Fall B liegt P<sub>99</sub> zwischen 183 und 607 ms. Der minimale Werte ist der kleinste der Anwendungsfälle, jedoch gibt es eine hohe Varianz. Insgesamt liegt die maximale Varianz über alle Anwendungsfälle hinweg bei 584 ms. Im direkten Vergleich mit den Pipe-Clean Tests sieht man, dass P<sub>95</sub> im schlechtesten Fall 224 ms und P<sub>99</sub> 656 ms langsamer geworden sind. Wird die Varianz der Werte mitberücksichtigt, handelt es sich im Besten Fall um eine Verschlechterung der Antwortzeit von 28 ms für P<sub>95</sub> und 72 ms für P<sub>99</sub>.

Es wird deutlich, dass ein schneller Lastanstieg eine höhere Varianz der oberen Quantile bedeutet, somit bessere Antwortzeiten erzielen kann als ein stetiger Anstieg der VU. Für die Skalierung auf eine neue und nicht bereitstehende Instanz kann es aber bei beiden Lastanstiegen zu einer Verschlechterung des 99. Quantil von bis zu 600 ms kommen.

Der nächste Fall der betrachtet wird ist die Skalierung auf eine bereits bereitstehende Instanz. Hier werden die Testergebnisse der Tests mit einem Instanzenminimum von zwei betrachtet. Dazu gehören die Ergebnisse von 1500 und 2000 VU der Stress Tests, sowie 2000 VU der Last Tests. Hier ist die mediane Antwortzeit aller Testfälle wieder bei 95 ms, die Werte für P<sub>95</sub> und P<sub>99</sub> steigen jedoch an. Auch hier gibt es einen Unterschied in Antwortzeiten zwischen der Art des Lastanstiegs, weswegen die Test Typen zunächst einzeln untersucht werden.

Bei den Stress Test mit zwei minimalen Instanzen lagen die Werte für P<sub>95</sub> nah bei einander und wiesen eine geringe Varianz auf. Die Werte der Test bewegten sich alle zwischen 119 und 223 ms. Somit kommt es zu einer Varianz von 104 ms. Für P<sub>99</sub> waren die Ergebnisse für alle Tests ebenfalls ähnlich. Anwendungsfall B mit 1500 VU bewegte sich zwischen 415 und 607 ms. Die Anwendungsfälle A und C mit 2000 VU bewegten sich ebenfalls in diesem Bereich, mit C zwischen 447 und 607 ms und A zwischen 431 und 543 ms. Der Bereich für Anwendungsfall B beginnt jedoch bei 399 ms und endet bei 607 ms. Die Varianz insgesamt beträgt somit 208 ms. Im Vergleich zu den Pipe-Clean Tests ist hier erkennbar, dass die Antwortzeit für P<sub>95</sub> im schlechtesten Fall um 128 ms zugenommen hat. Unter Betrachtung der Varianz, hat P<sub>95</sub> somit im Besten Fall eine Verschlechterung von 24 ms. Für P<sub>99</sub> hat die Antwortzeit im schlechtesten Fall 596 ms zugenommen, im besten Fall jedoch nur 388 ms. Die Antwortzeit im P<sub>95</sub> hat sich nicht bedeutend verändert, dafür wurden die Werte im P<sub>99</sub> deutlich langsamer.

Nun werden die Ergebnisse der Last Tests betrachtet. Anwendungsfall A mit 2000 VU hat ein P<sub>95</sub> von 107 bis 151 ms. Anwendungsfall B hingegen 207 bis 223 ms und Fall C 239 bis 271 ms. Die Varianz innerhalb dieser Wertebereiche ist niedrig, jedoch unterscheiden sich die Werte der Anwendungsfälle untereinander mehr als gewöhnlicherweise. Insgesamt liegt die Varianz aller Fälle bei 164 ms. Das 99. Quantil der Antwortzeiten des Anwendungsfalls A liegt zwischen 119 und 543 ms, bei Anwendungsfall B sind es zwischen 319 und 511 ms. Für Fall C liegen die Werte zwischen 607 und 639 ms. Hier ist

wieder zu erkennen, dass die Werte ungewöhnlich auseinander liegen und geringe Überlappung zeigen in den Anwendungsfällen. Insgesamt ist die Varianz relativ hoch bei 520 ms. Hier fällt Anwendungsfall C auf, da dieser die längsten Antwortzeiten hat und keinerlei Überschneidung zu den anderen Werten. Verglichen mit den Pipe-Clean Tests hat somit die Antwortzeit des 95. Quantils, durch die Skalierung auf eine bereitstehende Instanz, im schlechtesten Fall 176 ms zugenommen. Im besten Fall könnte es sich nur um 12 ms handeln. Für das 99. Quantil handelt es sich um eine Steigerung der Antwortzeit von 528 ms im schlechtesten Fall und im besten Fall um eine Steigerung von nur 8 ms.

Hier zeigt sich erneut, dass ein schneller Lastanstieg eine höhere Varianz der Antwortzeiten in den oberen Quantilen auslöst. Die oberen Quantile der Antwortzeiten sind bei beiden Fällen ähnlich, jedoch besteht bei einer schnell ansteigenden Last die Möglichkeit ein besseres Antwortzeiten zu erreichen. Insgesamt sollte man jedoch bei einer Skalierung auf eine bereitstehende Instanz auf eine Verschlechterung der Antwortzeiten im 99. Quantil rechnen.

Vergleicht man die eben genannten Werte, mit denen der Konfiguration mit einer minimal bereitstehenden Instanz, wird deutlich, dass die Skalierung auf eine bereits bereitgestellte Instanz den Antwortzeiten weniger beeinflusst. Bei den Stress Tests ist die Verbesserung hauptsächlich in dem 95. Quantil zu erkennen, dort handelt es sich um eine Verbesserung von bis zu 140 ms. Bei den Last Tests verbessern sich die Antwortzeiten in beiden Quantilen. Im 95. Quantil ist es eine Verbesserung von knapp 50 ms, für das 99. Quantil ist es eine Verbesserung von bis zu 128 ms.

Bei dem letzten Fall handelt es sich um die Überlastung des Systems. In diesem Testfall muss der Skalierungsalgorithmus auf das Instanzenmaximum skalieren. Hierbei wurde das System einer Last von 4000 VU pro Sekunde ausgesetzt, was alle aktiven Instanzen ausgereizt hat. Hier werden alle 4000 VU Tests gemeinsam analysiert.

Zunächst werden die Ergebnisse der 4000 VU Stress Tests für eine minimale Instanz von 1 und 2 betrachtet. Direkt auffällig bei einer minimalen Instanz von 1 war die mediane Antwortzeit des Anwendungsfalles B. Diese stieg von 95 auf 99 ms an. Die Ergebnisse der oberen Quantile sahen aus wie folgt. Für Anwendungsfall A lagen die Werte für P<sub>95</sub> zwischen 231 und 287 ms, ähnlich wie bei Anwendungsfall C zwischen 271 und 287 ms. Anwendungsfall B dagegen hatte ein 95. Quantil der Antwortzeiten zwischen 367 und 2047 ms. Für die Fälle A und C entsteht eine Abweichung der Antwortzeiten von 56 ms, für Fall B liegt diese bei 1680 ms. Ähnlich ist diese Verteilung auch im 99. Quantil. Hier bewegen sich die Anwendungsfälle A und C in einem Wertebereich von 415 bis 831 ms, wobei Fall A den unteren Bereich deckt und Fall C den oberen. Anwendungsfall B dagegen besitzt einen Wertebereich von 639 bis 5119 ms. Auch hier sind die Abweichung für die Fälle A und C deutlich geringer mit 416 ms, Fall B dagegen besitzt eine Abweichung von 4480 ms.

Die Ergebnisse für eine minimale Instanz von 2 waren dagegen etwas schneller. Für das 95. Quantil lagen die Werte für die Fälle A und C zwischen 231 und 319 ms. Anwendungsfall B ist hier wieder auffallend langsam im Vergleich zu A und C, mit einer [P<sub>95</sub>](#) zwischen 367 und 431 ms. Für diesen Fall wird die Abweichung wieder für alle Fälle zusammen betrachtet. Die Abweichung liegt bei 200 ms. Für das 99. Quantil lagen alle Werte der Anwendungsfälle zwischen 447 und 735 ms, somit mit einer Abweichung von 288 ms.

Vergleicht man diese Werte mit den Basiswerten der Pipe-Clean Tests, sind für die Fälle A und C bei einer minimalen Instanz folgende Änderungen erkennbar. Im 95. Quantil steigen die Antwortzeiten im schlechtesten Fall 192 ms, und im Besten 136 ms. Im 99. Quantil ist es eine Steigerung auf 304 bis 720 ms. Für Fall B handelt es sich im 95. Quantil um eine Steigerung von 272 bis 1952 ms. Für [P<sub>99</sub>](#) ist es eine Steigerung von 528 bis 5008 ms. Für die Konfiguration mit zwei minimalen Instanzen war für alle Anwendungsfälle das 95. Quantil zwischen 136 und 336 ms und das 99. Quantil zwischen 336 und 624 ms langsamer.

Allgemein befinden sich die Antwortzeiten bei beiden Instanzenminima in einem ähnlichen Bereich. Hier wird jedoch deutlich, dass die Antwortzeiten einer einzigen minimalen Instanz, bei einer Überlastung, stark variieren, vor allem bei Fall B mit der höchsten Anfragendichte. Bei einer einzigen bereitstehenden Instanz nehmen im 99. Quantil die Abweichungen in den Antwortzeiten stark zu, verglichen mit zwei bereitstehenden Instanzen.

Als nächstes werden die Ergebnisse der Last Tests mit 4000 [VU](#) betrachtet, wobei hier erneut das Instanzenminimum von 1 zuerst erläutert wird. Hier ist wieder zu erkennen, dass Anwendungsfall B einen erhöhten Median besitzt verglichen zu den anderen Anwendungsfällen, mit 115 ms. Die Antwortzeiten für [P<sub>95</sub>](#) waren in den Fällen A und C zwischen 271 und 399 ms, für B wiederum zwischen 399 und 495 ms. Insgesamt besitzen die Werte eine Abweichung von 224 ms. Für [P<sub>99</sub>](#) liegen die Werte bei den Fällen A und C zwischen 607 und 767 ms, für B zwischen 703 und 767 ms. Dadurch bildet sich eine Abweichung von 160 ms.

Nun die Ergebnisse des Last Tests mit 4000 [VU](#) und einem Instanzenminimum von zwei. Hier war der Median für den Anwendungsfall B erneut höher als bei anderen Testdurchläufen, mit 103 ms. Für Anwendungsfall A waren die Werte für [P<sub>95</sub>](#) zwischen 247 und 303 ms und für Fall C zwischen 319 und 367 ms. Für Fall B waren die Werte zwischen 447 und 543 ms. Über alle Anwendungsfälle hinweg ergibt sich also eine Abweichung von 296 ms. Für [P<sub>99</sub>](#) waren die Werte wie folgt aufgeteilt. Anwendungsfall A hatte ein 99. Quantil von 495 bis 639 ms. Die Werte für Anwendungsfall C lagen zwischen 607 und 703 ms und für Anwendungsfall B zwischen 703 und 767 ms. Hier ergibt sich also eine Abweichung zwischen allen Fällen von 272 ms.

Im Vergleich mit den Basiswerten der Pipe-Clean Tests werden folgende Änderungen erkennbar. Bei einer minimalen Instanz von 1, werden in allen Anwendungsfällen die Antwortzeiten für das 95. Quantil 176 bis 400 ms langsamer. Für das 99. Quantil liegen diese Werte zwischen 496 und 656 ms.

Für ein Instanzenminimum von zwei sind die Werte aller Anwendungsfälle ähnlich. Für das 95. Quantil wurden die Antwortzeiten 152 bis 448 ms und für das 99. Quantil 384 bis 656 ms langsamer. Hier ist zu erkennen, dass es keine signifikanten Unterschiede zwischen den beiden Instanzenminima gibt. Nur im 99. Quantil mit zwei bereitstehenden Instanzen ist eine höhere Abweichung zu erkennen, welche eventuell geringere Antwortzeiten liefern könnte.

Die hohe Abweichung der Antwortzeiten, die bei den Stress Tests, mit einer minimal bereitstehenden Instanz, bei Fall B entstanden ist, ist bei den Last Tests nicht aufgekommen. Das bestätigt die Idee, dass der Skalierungsalgorithmus auf einen schnellen Lastanstieg, besser reagiert als auf einen stetigen. Betrachtet man die Antwortzeiten ohne diesen Fall B wird deutlich, dass die Antwortzeiten der Stress Tests mit einer minimal bereitstehenden Instanz im Vergleich zu den Last Tests mit gleicher Konfiguration im 95. Quantil schneller waren. Das 99. Quantil verhält sich jedoch bei allen Test Typen ähnlich. Bei einem Vergleich zwischen Last und Stress Test mit zwei minimal bereitstehenden Instanzen werden keine signifikanten Änderungen deutlich. Dennoch ist auffällig, dass die Antwortzeiten der Last Tests bei allen Testdurchläufen weniger Abweichungen in den Werten aufzeigen, als bei den Stress Tests.

Grundsätzlich können also folgende Aussagen getroffen werden. Die Skalierung auf eine nicht bereitstehende Instanz erhöht die Antwortzeiten deutlich, bis zu 600 ms, weswegen es sich lohnt eine zweite Instanz bereitzustellen. Dies würde Kaltstarts vermeiden und die Antwortzeiten bei einer Skalierung niedrig halten, somit die Leistung der Anwendung verbessern. Hierbei spielt jedoch die Art des Lastanstiegs ebenfalls eine Rolle. Bei einem schnellen Lastanstieg verbessern sich die Antwortzeiten verglichen mit einem stetigen Lastanstieg vor allem in dem 95. und 99. Quantil. Dies gilt ob auf eine bereitstehende oder nicht bereitstehende Instanz skaliert wird. Dieser Faktor ist jedoch von dem Skalierungsalgorithmus abhängig und eine Folge der Belastung der Anwendung. Bei einer Überlastung des Dienstes ist dies für ein Instanzenminimum von eins ebenfalls zu erkennen. Die Leistung mit einer bereitstehenden Instanz war vergleichbar mit der von zwei bereitstehenden Instanzen, solange der Lastanstieg schnell war. War der Lastanstieg stetig, so stiegen die Antwortzeiten für Fälle mit einer hohen Anfragedichte stark an, bis auf 5000 ms. Generell hat eine Überlastung zu einer Verschlechterung der Leistung geführt, wobei es sich bei den meisten Fällen um ungefähr 600 bis 700 ms gehandelt hat.

## 6.2 KOSTEN

Es gibt mehrere Faktoren die die Kosten einer App Runner Konfiguration beeinflussen, darunter die Wahl der [AWS](#) Region des Servers in der man die Anwendung in Betrieb nehmen wird. In der folgenden Kostenuntersuchung betrachten wir die Kosten für die Region eu-central-1, beziehungs-

weise Frankfurt. Weitere Faktoren die die Kosten einer Anwendung in App Runner beeinflussen ist die Wahl der Konfiguration, also die Anzahl an **vCPU** und die Größe des Speichers. Man kann zwischen den Kombinationen in Tabelle 6.1 wählen, wobei pro genutzte Zeit der **vCPU** und des Speichers Kosten abgerechnet werden. Die in diesen Tests genutzte Konfiguration ist die

CPU	Speicher
1 <b>vCPU</b>	2GB, 3GB, 4GB
2 <b>vCPU</b>	4GB

Tabelle 6.1: Konfiguration (**vCPU** und Speicher)

Kombination einer **vCPU** und 2GB Speicher, die kleinste Konfiguration. Des Weiteren gibt es Kosten zur Erstellung der Anwendung. Dabei werden pro Erstellungsminute, also die Zeit die benötigt wird um die Anwendung aus dem bereitgestellten Quellcode zu erstellen, 0,005 \$ abgerechnet. Das sind in unserem Fall jedoch nur einmalige Kosten, da die Anwendung einmalig erstellt wurde. Die bereitgestellten aber inaktiven Instanzen werden ebenfalls abgerechnet, diese haben jedoch eine andere Rate als aktiv arbeitende Instanzen. Eine bereitgestellte Instanz wird über den genutzten Speicher abgerechnet. Aktuell betragen die Kosten 0,007\$ pro GB-Stunde. Man zahlt pro bereitgestellte Instanz 0,007\$ pro GB pro Stunde.

Aktive Instanzen hingegen werden über die **vCPU** und den Speicher abgerechnet. Dabei betragen die Kosten 0,064\$ pro **vCPU** Stunde und dazu die 0,007\$ pro GB-Stunde. [17] Somit gilt allgemein für die Kosten einer bereitstehenden Instanz (**KBr**):

$$\mathbf{KBr} = \text{Instanzen} * \text{Stunden} * \text{Tage} * (\text{GB} - \text{Stunde} * 0,007\$) \quad (6.1)$$

und für die Kosten einer aktiven Instanz (**KAk**):

$$\mathbf{KAk} = \text{Instanzen} * \text{Stunden} * \text{Tage} * (\text{GB} - \text{Stunde} * 0,007\$ + \text{Anzahl}\mathbf{vCPU} * 0,064\$) \quad (6.2)$$

Aus dieser Formel lässt sich erkennen, dass die n-fache Anzahl an Instanzen die n-fachen Kosten für die Anwendung ergeben. Somit sind die Kosten besonders von der Last die die Anwendung erfährt abhängig. Nun wird die Zeit die benötigt wurde um die Anwendung zu erstellen errechnet. Hierbei handelt es sich bei der Testanwendung, um ungefähr eine Minute für die Erstellung, also 0,005\$. Die Erstellungszeit ist ebenfalls abhängig von der Komplexität der Anwendung. Durch die Aufteilung in aktive Nutzung der Anwendung und bereitgestellte Anwendung ohne Nutzung, entsteht ein Kostenbereich in dem sich die Anwendung bewegen kann. Minimale Kosten für die Konfiguration mit einem Instanzenminimum von 1 pro Monat lägen bei:

$$\mathbf{KBr} = 1 * 24 * 30 * (2 * 0,007\$) = 10,08\$$$

Für zwei Instanzen wären das demnach  $2 * 10,08\$ = 20,16\$$ .

Wobei hier kein einziges Mal eine Instanz aktiviert werden kann, es handelt sich nur um die bereitstehende Instanz die keine Anfragen empfängt. Tatsächlich legt das Instanzenmaximum die obere Grenze der Kosten fest. Das würde bedeuten, dass bei einem Dauer Betrieb mit maximaler Auslastung für beide Konfiguration die Kosten wie folgt wären:

$$KAk = 3 * 24 * 30 * (2 * 0,007\$ + 1 * 0,064\$) = 168,48\$$$

Als nächstes werden die Kosten der für diese Tests genutzte Testanwendung für einen Zeitraum von 3 Tagen, also 72 Stunden betrachten. In diesen drei Tagen wurde ein großer Teil des Tests durchgeführt. Dies summierte sich auf 28,5 Stunden Aktivität. Von diesen 28,5 Stunden waren 13,33% der Zeit nur eine Instanz aktiv, also 3,8 Stunden. 46,6% der Zeit waren 2 Instanzen aktiv, also 13,3 Stunden und 40% der Zeit 3 Instanzen, also 11,4 Stunden. Die Kosten summieren sich somit auf folgenden Wert:

$$\begin{aligned} KAk &= 1 * 3,8 * (2 * 0,007\$ + 1 * 0,064\$) \\ &+ 2 * 13,3 * (2 * 0,007\$ + 1 * 0,064\$) \\ &+ 3 * 11,4 * (2 * 0,007\$ + 1 * 0,064\$) = 5,04\$ \end{aligned}$$

Wichtig ist hierbei jedoch, dass zwischen Testreihen die Anwendung pausiert wurde. In dem pausierten Zustand ist die Anwendung nicht erreichbar und verbraucht keine Ressourcen. Die Kosten für die pausierte Zeit wäre dann mit einer minimalen Instanz wie folgt:

$$KBr = 1 * 43,5 * (2 * 0,007\$) = 0,609\$$$

Addiert man diese Werte noch mit der Erstellungszeit, summieren sich die Kosten auf folgenden Wert:

$$\begin{aligned} \text{Kosten} &= KAk + KBr + \text{Erstellungskosten} \\ &= 5,04\$ + 0,609\$ + 0,005\$ = 5,654\$ \end{aligned}$$

Allgemein bewegt man sich mit App Runner in einem relativ weitem Preisbereich. Viele Faktoren sind abhängig von dem Anwendungsbereich des Dienstes. Je mehr Speicher notwendig ist um die Anwendung zu betreiben, desto mehr passive Kosten entstehen, da auch bei der ungenutzten Zeit der Anwendung die GB-Stunden pro bereitstehende Instanz berechnet wird. Sollte jedoch nicht viel Speicher notwendig sein, aber dafür die Anwendung oft genutzt werden und hoher Belastung ausgesetzt sein so entstehen Kosten pro aktive Instanz.

Grundsätzlich können Einstellungen vorgenommen werden, um diese Kosten in einem gewissen Rahmen zu halten. Zuerst kann der Nutzer eine „maximum Concurrency“ einstellen. Je höher die Concurrency, desto mehr kann eine einzige Instanz an Anfragen bearbeiten. Dies senkt die Anzahl an aktivierten Instanzen, bei einer hohen Belastung. Diese Einstellung ist jedoch



auch von dem Quell-Code der Anwendung abhängig und sollte gut gewählt sein, um eine Überlastung der Anwendung zu verhindern. Durch eine hohe Einstellung der „maximum Concurrency“ kann man ein kleines Instanzen Minimum wählen. Ein geringe Anzahl an bereitstehender Instanzen senkt die passiven Kosten, für das Betreiben der Anwendung. Zuletzt kann ein Instanzenmaximum eingestellt werden, um die aktiven Kosten zu limitieren. Hier ist jedoch zu berücksichtigen, wie viel Last die Anwendung aushalten muss. Wie in 6.1 erkennbar können bei zu hoher Last die Verfügbarkeit und Antwortzeiten der Anwendung stark abnehmen.

Eine Einstellung die zuvor nicht erwähnt wurde, ist eine automatisierte Bereitstellung, welche monatlich abgerechnet wird. Bei Änderungen im Quell-Code in GitHub wird die Anwendung automatisch neu erstellt und in Betrieb genommen. Dadurch können Änderungen und neue Features direkt in der Anwendung genutzt werden. [17] Diese Einstellungsmöglichkeit lohnt sich für Anwendungsfälle bei denen kontinuierliche Änderung an der Anwendung vorgenommen werden.

All diese Einstellungen sind somit von der Art und dem Anwendungsbereich der Anwendung abhängig. Generell kann aber gesagt werden, dass sich die Kosten einer App Runner Anwendung schnell summieren können, wenn die Anwendung viel Last bekommt. Somit ist der dominierende Faktor die Anzahl aktiver Instanzen, gefolgt von der Konfiguration der vCPU und des Speichers.

### 6.3 IMPLIKATIONEN FÜR DIE PRAXIS

App Runner ist ein hochskalierbarer Dienst der es ermöglicht bis zu 25 Instanzen gleichzeitig laufen zu lassen. Je nach Anwendung und Wahl der „maximum Concurrency“ und der Kombination aus vCPU und Speicher, kann dieser Dienst große Mengen Last bearbeiten. Dazu wird das Skalierungsverhalten und das Load Balancing zwischen den Instanzen von App Runner durchgeführt was die Inbetriebnahme einer Anwendung erleichtert. Diese Aspekte nehmen dem Anwender jedoch die Möglichkeit eigene Änderungen, Konfiguration oder eventuelle Optimierungen an, zum Beispiel dem Skalierungsverhalten, vorzunehmen. Während der Analyse wurde festgestellt, dass es vor allem bei der Skalierung auf eine neue Instanz, zu einer Verschlechterung der Leistung kam. Die Antwortzeiten und die Menge an Fehler-Codes steigen durch eine Skalierung auf eine weitere Instanz stark an. Dieses instabile Skalierungsverhalten, bedeutet für die Anwendung eine Senkung der Leistung, weswegen dieser Dienst für zeitkritische und in den Antwortzeiten konsistente Anwendungen ungeeignet ist.

Wie in der Sektion 6.1 jedoch festgestellt gibt es Anwendungsfälle, welche die Verfügbarkeit und Leistung der Anwendung während einer Skalierung weniger beeinflussen, nämlich ein lang anhaltender Betrieb mit kurzen und schnellen Lastanstiegen. Bei solchen Anwendungsfällen arbeitet der Skalierungsalgorithmus am schnellsten und skaliert mit geringem Einfluss auf die allgemeine Leistung. Diese Art Belastung würde im besten Fall zu einer

kaum messbaren Verschlechterung der Leistung führen. Da solche Aspekte der Belastung jedoch nicht im Vorhinein bekannt sind, sollte es sich also, um eine Anwendung handeln welche von einer kurzzeitige Senkung der Leistung nicht stark beeinflusst ist.

Nichts desto trotz wäre der beste Anwendungsfall, um die Leistung von App Runner vollkommen auszunutzen, eine Anwendung die nicht skalieren muss, oder nur auf Instanzen skaliert die bereits bereitgestellt sind. Hier ist die Leistung von App Runner am stabilsten. Dies könnte erreicht werden in dem eine hohe „maximum Concurrency“ gewählt wird mit einem hohen Instanzenminimum.

Unter Betrachtung der Kosten wäre diese Art der Anwendung jedoch eher ungünstig, da eine hohe Anzahl an bereitstehenden Instanzen hohe passive Kosten bedeuten. Generell sind die Kosten von App Runner nicht zu unterschätzen. Wie in 6.2 errechnet kommt es bei einem Dauerbetrieb und einer hohen Belastung in kurzer Zeit zu hohen Geldausgaben. Vor allem, wenn für die Anwendung eine hohe Zahl vCPU oder Speicher gewählt wurde. Um diese Kosten zu limitieren sollte also ein Instanzenmaximum gewählt werden, welches immer noch die Last der Anwendung stemmen kann, um extreme Leistungsverluste zu verhindern.



## ZUSAMMENFASSUNG UND AUSBLICK

---

Das folgende Kapitel fasst die Ergebnisse der Analyse dieser Arbeit zusammen und bietet einen Ausblick für App Runner.

Im Laufe dieser Arbeit wurde der neue [AWS](#) Dienst App Runner genauer vorgestellt und dessen Leistung untersucht. Dafür wurde zuerst die Funktionalität von App Runner, mit allen Besonderheiten und Einstellungsmöglichkeiten des Dienstes dargestellt. Im Anschluss wurden eine Testanwendung und die Tests konzipiert, mit dem Ziel eine große Vielfalt an Leistungsaspekten von App Runner untersuchen zu können. Aus den Ergebnissen der Tests konnte folgende Erkenntnisse gezogen werden. App Runner verfügt über einen Skalierungsalgorithmus, welcher automatisch, abhängig von der Belastung der Anwendung, hoch und herunter skaliert. Die Leistung von App Runner ist besonders von dem Verhalten dieses Skalierungsalgorithmus abhängig, da die Skalierung kurzzeitig zu Fehler-Codes und einer Verschlechterung der Antwortzeiten führt. Diese Verschlechterung der Leistung kann von dem Anwender in gewissem Maße beeinflusst werden. Es existieren Einstellungen, um dem Skalierungsalgorithmus Anhaltspunkte für ein spezifisches Verhalten zu geben und für die Anwendung anzupassen. Hier hat der Nutzer die Möglichkeit ein Instanzenminimum zu wählen. Diese minimal bereitstehenden und untätigen Instanzen stehen für eine Skalierung bereit und beeinflussen dadurch die Leistung der Anwendung weniger. Darüber hinaus wurde während der Tests festgestellt, dass eine lang anhaltende Belastung mit kurzen Anstiegen in Belastung die Leistung weniger beeinflusst. Zuletzt wurde ein Kostenmodell für App Runner aufgestellt und dieses dann analysiert. App Runner berechnet hierbei die Kosten pro Instanz, ob aktiv oder untätig. Dadurch summieren sich passive Kosten für den Nutzer, unabhängig von der Benutzung der Anwendung. Zusätzlich summieren sich die Kosten, bei hoher Belastung der Anwendung, abhängig des gewählten Instanzenmaximums. Sollte das Instanzenmaximum hoch gewählt sein, so können viele Instanzen zur gleichen Zeit aktiv Anfragen bearbeiten und hohe Kosten erzeugen. Diese Erkenntnisse wurden anhand Last und Stress Tests gemacht, wobei hier primär der Aspekt der Skalierung betrachtet wurde, indem unterschiedlich Instanzenminima und Lastanstiege getestet wurden. Wie die Anzahl [vCPU](#) und die Menge Speicher die Leistung beeinflusst wurde in dieser Arbeit nicht betrachtet. Weiterhin handelte es sich bei der Testanwendung um ein simples HTTP-Backend ohne Verbindung zu einer externen Datenbank. Wie sich App Runner in Kombination mit einer externen Datenbank verhält und wie sich die Leistung hierbei verändert, wären Themen auf die in Zukunft eingegangen werden könnte, um die Leistung von App Runner noch genauer zu untersuchen.

## LITERATUR

---

- [1] Artillery.io. *Installing Artillery Pro*. (besucht am 17.11.2021). URL: <https://www.artillery.io/docs/guides/getting-started/installing-artillery-pro>.
- [2] Artillery.io. *Testing HTTP*. (besucht am 17.11.2021). URL: <https://www.artillery.io/docs/guides/guides/http-reference#request-timeouts>.
- [3] Artillery.io. *User Guide*. (besucht am 26.08.2021). URL: <https://artillery.io/docs/guides/overview/welcome.html>.
- [4] Artillery.io. *Writing your first test*. (besucht am 06.09.2021). URL: <https://artillery.io/docs/guides/getting-started/writing-your-first-test.html>.
- [5] MDN Web Docs. *200 OK - HTTP | MDN*. (besucht am 17.11.2021). URL: <https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/200>.
- [6] MDN Web Docs. *429 Too Many Requests - HTTP | MDN*. (besucht am 17.11.2021). URL: <https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/429>.
- [7] MDN Web Docs. *502 Bad Gateway - HTTP | MDN*. (besucht am 17.11.2021). URL: <https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/502>.
- [8] MDN Web Docs. *503 Service Unavailable - HTTP | MDN*. (besucht am 17.11.2021). URL: <https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/503>.
- [9] Bayo Erinle. *Performance Testing with JMeter 3*. Dritte Edition. 2017. ISBN: 978-1-7872-8577-4. URL: <https://learning.oreilly.com/library/view/-/9781787285774>.
- [10] Nebrass Lamouchi. "Getting Started with Containerization". In: *Pro Java Microservices with Quarkus and Kubernetes: A Hands-on Guide*. Berkeley, CA: Apress, 2021, S. 1–33. ISBN: 978-1-4842-7170-4. DOI: [10.1007/978-1-4842-7170-4\\_1](https://doi.org/10.1007/978-1-4842-7170-4_1). URL: [https://doi.org/10.1007/978-1-4842-7170-4\\_1](https://doi.org/10.1007/978-1-4842-7170-4_1).
- [11] Ian Molyneaux. *The Art of Application Performance Testing*. Zweite Edition. Sebastopol, CA, 2014. ISBN: 978-1-4919-0054-3. URL: <https://learning.oreilly.com/library/view/-/9781491900536>.
- [12] Glenford J. Myers. *The art of software testing*. Dritte Edition. Hoboken, N.J., 2012. ISBN: 978-1-1180-3196-4. URL: <https://learning.oreilly.com/library/view/-/9781118133156>.

- [13] Alessandro V. Papadopoulos, Laurens Versluis, André Bauer, Nikolas Herbst, Jóakim von Kistowski, Ahmed Ali-eldin, Christina L. Abad, José N. Amaral, Petr Tuma 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. 1528–1543. ISSN: 1939-3520. DOI: [10.1109/TSE.2019.2927908](https://doi.org/10.1109/TSE.2019.2927908).
- [14] Gabriel Schenker. *Getting Started with Containerization*. Erste Edition. 2019. ISBN: 978-1-8386-4570-0. URL: <https://learning.oreilly.com/library/view/-/9781838645700>.
- [15] Naresh Kumar Sehgal und Pramod Chandra P. Bhatt. "Introduction". In: *Cloud Computing: Concepts and Practices*. Cham: Springer International Publishing, 2018, S. 1–10. ISBN: 978-3-319-77839-6. DOI: [10.1007/978-3-319-77839-6\\_1](https://doi.org/10.1007/978-3-319-77839-6_1). URL: [https://doi.org/10.1007/978-3-319-77839-6\\_1](https://doi.org/10.1007/978-3-319-77839-6_1).
- [16] Amazon Web Services. *AWS App Runner - Developer Guide*. (besucht am 18.08.2021). URL: <https://docs.aws.amazon.com/apprunner/latest/dg/apprunner-guide.pdf>.
- [17] Amazon Web Services. *AWS App Runner Preise*. (besucht am 18.11.2021). URL: <https://aws.amazon.com/de/apprunner/pricing/>.
- [18] Amazon Web Services. *Amazon API Gateway - Developer Guide*. (besucht am 17.11.2021). URL: <https://docs.aws.amazon.com/apigateway/latest/developerguide/api-gateway-request-throttling.html#apigateway-how-throttling-limits-are-applied>.
- [19] Amazon Web Services. *Amazon Elastic Compute Cloud - User Guide*. (besucht am 10.09.2021). URL: <https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/ec2-ug.pdf#instance-optimize-cpu>.
- [20] Amazon Web Services. *Webinar: AWS App Runner Deep Dive*. (besucht am 25.11.2021). URL: [https://pages.awscloud.com/AWS-App-Runner-Deep-Dive\\_2021\\_0606-CON\\_OD.html?&trk=ep\\_card-el\\_a131L0000084iG2QAI&trkCampaign=NA-FY21-AWS-DIGMKT-WEBINAR-SERIES-GC-300-June-2021-0606-CON&sc\\_channel=el&sc\\_campaign=pac\\_2018-2021-exlinks\\_ondemand\\_OTT\\_evergreen&sc\\_outcome=Product\\_Adoption\\_Campaigns&sc\\_geo=NAMER&sc\\_country=mult](https://pages.awscloud.com/AWS-App-Runner-Deep-Dive_2021_0606-CON_OD.html?&trk=ep_card-el_a131L0000084iG2QAI&trkCampaign=NA-FY21-AWS-DIGMKT-WEBINAR-SERIES-GC-300-June-2021-0606-CON&sc_channel=el&sc_campaign=pac_2018-2021-exlinks_ondemand_OTT_evergreen&sc_outcome=Product_Adoption_Campaigns&sc_geo=NAMER&sc_country=mult).
- [21] Andreas Spillner. *Basiswissen Softwaretest*. Sechste Edition. Heidelberg: O'Reilly Media, INC., 2019. ISBN: 978-3-8649-0583-4. URL: <https://learning.oreilly.com/library/view/-/9781492072164>.
- [22] S. Srinivasan. "Basic Cloud Computing Types". In: *Cloud Computing Basics*. New York, NY: Springer New York, 2014, S. 17–41. ISBN: 978-1-4614-7699-3. DOI: [10.1007/978-1-4614-7699-3\\_2](https://doi.org/10.1007/978-1-4614-7699-3_2). URL: [https://doi.org/10.1007/978-1-4614-7699-3\\_2](https://doi.org/10.1007/978-1-4614-7699-3_2).
- [23] *What is a Container? | App Containerization | Docker*. (besucht am 12.08.2021). URL: <https://www.docker.com/resources/what-container>.

Teil II

APPENDIX

## REPOSITORY

---

In diesem Kapitel wird kurz der Aufbau des Repository beschrieben, welches bei diesem Link <https://github.com/laura-franky/bachelor-project> zu finden ist. Das Repository besitzt drei Ordner „artillery“, „pandas“ und „reports“.

Im Ordner „artillery“ sind im Unterordner „config“ die Konfigurationen der einzelnen Test Typen zu finden. Im Unterordner „scenarios“ sind die Szenarien, also die Anwendungsfälle A, B und C, der Tests definiert.

Der Ordner „pandas“ beinhaltet den Quell-Code zur Visualisierung der Ergebnisse. Hier wurde, zur Analyse und Visualisierung der Daten, die python-Bibliothek pandas verwendet. Im Unterordner „Data“ sind die Ergebnisse der Tests zu finden, die zur Erstellung der Graphen verwendet wurden. Alle weiteren Unterordner beinhalten den Code zu Erstellung der Graphen, sowie einen weiteren Unterordner namens „export“. Dort sind die, in der Arbeit verwendeten, Graphen als PDF-Datei abgespeichert.

Zuletzt sind im Ordner „reports“ alle von artillery automatisch erstellen Reports vorzufinden. Die Ergebnisse der Tests sind aufgeteilt auf das Instanzenminimum, dann Test Typ und letztlich Anwendungsfall. Bei den Dateien handelt es sich einerseits um die JSON-Dateien, andererseits um die HTML Dateien, welche eine weitere graphische Darstellung der Ergebnisse beinhaltet. Diese Graphen wurden nicht in dieser Arbeit verwendet, bieten jedoch eine weitere Visualisierung der Ergebnisse pro Testdurchlauf.

Auf der ersten Ebene des Repository existieren weiterhin mehrere Dateien. Die Datei „apprunner.yaml“ beinhaltet die Spezifizierung der Laufzeitumgebung, den Erstellungsbefehl, sowie den Startbefehl und den Port auf dem die Anwendung läuft. Diese Datei wird von App Runner verwendet, um die Anwendung erstellen und in Betrieb nehmen zu können. Der Inhalt sieht aus wie folgt:

---

```
version: 1.0
runtime: nodejs12
build:
  commands:
    build:
      - npm install
run:
  command: npm start
network:
  port: 3000
```

---

## A.1 QUELL-CODE DER TESTANWENDUNG

Die Dateien „index.js“ und „recipes.js“ bilden zusammen die Testanwendung dieser Arbeit. In der Datei „recipes.js“ sind die Beispiel Rezepte der Anwendung zu finden mit denen die Tests durchgeführt wurden. Die Datei „index.js“ beinhaltet den Quell-Code der Testanwendung. Hier sind die Routen der Anwendung zu finden, sowie der Code, um die Anwendung zu starten. Diese sieht aus wie folgt:

---

```
import express, { Router } from "express";
import recipes from "./recipes.js";
import bodyParser from "body-parser";

const port = 3000;
export const recipeRouter = Router({ mergeParams: true });

// Basic get
recipeRouter.get("/", async (_, res) => {
  res.send({
    message: "Willkommen",
  });
});

// Get all recipes
recipeRouter.get("/recipes", async (_, res) => {
  res.send({
    data: recipes,
  });
});

// Get one recipe by ID
recipeRouter.get("/recipes/:id", async (req, res) => {
  const id = Number(req.params.id);
  const recipe = recipes.find((recipe) => recipe.id == id);

  if (typeof recipe === "undefined") {
    res.status(404).send({
      message: "recipe not found",
    });
  } else {
    res.send({
      data: recipe,
    });
  }
});

// Update one recipe by ID
recipeRouter.patch("/recipes/:id", async (req, res) => {
  const id = Number(req.params.id);
  const recipeUpdate = req.body;

  const recipe = recipes.find((recipe) => recipe.id == id);
```

```

    if (typeof recipe === "undefined") {
      res.status(404).send({
        message: "recipe not found",
      });
    } else {
      const updatedRecipe = { id, ...recipeUpdate };
      res.send({
        data: updatedRecipe,
      });
    }
  });

  // Create a recipe
  recipeRouter.post("/recipes/", async (req, res) => {
    const recipeInfo = req.body;
    const newID = recipes.length + 1;
    const createdRecipe = { id: newID, ...recipeInfo };

    res.send({
      data: createdRecipe,
    });
  });

  export const startServer = () => {
    const app = express();

    app.use(bodyParser.json());
    app.use("/", recipeRouter);

    const server = app.listen(port, () =>
      console.log("Server is running on port", port)
    );
    return { server };
  };

  startServer();

```

---