HAW HAMBURG

Bachelorarbeit

Ante Škorić

Realisierung und Gegenüberstellung einer serverless und einer containerbasierten Microservice-Architektur für das Client-Reporting

Betreuung durch: Prof. Dr. Jens von Pilgrim / Prof. Dr. Stefan Sarstedt
Eingereicht am: 01.01.2022

Fakultät Technik und Informatik
Department Informatik

Faculty of Computer Science and Engineering
Department Computer Science

Ante Škorić

Thema der Arbeit

Realisierung und Gegenüberstellung einer serverless und einer containerbasierten Microservice-Architektur für das Client-Reporting

Keywords: Microservices, Serverless, Container, Performance, Scaling, Resilience, Container Orchestration, Kubernetes, AWS Lambda, Cloud, Cloud Native Computing, Kafka, Client-Reports

Kurzzusammenfassung

Microservices und Serverless Deploymentstrategien unterscheiden sich in vielerlei Hinsicht. Bei Serverless kümmert sich der Cloud-Anbieter um die Skalierung und bietet zusätzliche Resilience. Im Gegensatz dazu müssen bei der containerbasierten Microservice-Anwendung die Container mithilfe eines Container-Orchestrierungssystems skaliert werden. In dieser Arbeit wird folgende Fragestellung beantwortet: "Wie unterscheiden sich die Performance (Skalierbarkeit, Resilience und Cloud Latency) und Kosten zwischen einer Serverless-Deploymentstrategie mit AWS Lambda und einer containerbasierten Microservices-Deploymentstrategie mit Docker und Kubernetes?" Durch die Beantwortung dieser Forschungsfrage ergeben sich Use Cases, in denen die eine Deploymentstrategie der anderen vorgezogen werden kann. Zur Beantwortung der Fragestellung wurden Performance-Tests durchgeführt und eine Kostenberechnung vorgenommen. Dies geschah anhand von zwei verschiedenen Use Cases, einer event-driven Reporting-Anwendung und einer Employee-Time-Sheet-Management-Portal-Anwendung. Für die Implementierung und das Deployment der Serverless-Anwendungen wurde AWS Lambda verwendet, für die Microservice-Anwendungen wurden Docker und Kubernetes eingesetzt. Die durchgeführten Experimente zeigen, dass beide Deploymentstrategien Vor- und Nachteile aufweisen. Die Nachteile der Microservices-Deploymentstrategie bestehen darin, dass sie unter den Load-Balancing- und Trafficverteilungs-Problemen leidet und die Skalierbarkeits- und Ressourcen-Konfiguration "aufwendiger" ist als die von Serverless. Die Vorteile sind, dass sie im Vergleich zu Serverless eine bessere Requestsbearbeitungsleistung bietet. Der Nachteil der Serverless-Deploymentstrategie ist der, dass sie unter dem Problem des Cold-Starts leidet. Die Vorteile sind, dass sie agiler in Bezug auf Skalierbarkeit als die Microservices-Deploymentstrategie ist und dass sie bei den Anwendungen

mit geringem Traffic weniger Kosten verursacht. Aus diesen Erkenntnissen ergeben sich auch Use Cases, in denen die eine Deploymentstrategie der anderen vorgezogen werden kann. Die Microservices-Deploymentstrategie eignet sich für Real-Time-Data-Processing-Anwendungen und Anwendungen, bei denen es auf Bruchteile von Sekunden ankommt. In Gegenteil dazu eignet sich die Serverless-Deploymonetstrategie für die Anwendungen mit Traffic-Schwankungen, Job-Scheduler und automatisch skalierbare APIs. An dieser Stelle ist es wichtig, darauf hinzuweisen, dass die Empfehlungen anhand der Kriterien aus dieser Arbeit gemacht worden sind und keine weiteren Kriterien berücksichtigt wurden. Bei der Auswahl einer Deploymentstrategie in der Praxis sollten auch andere Kriterien, die in dieser Arbeit nicht behandelt wurden, berücksichtigt werden.

Ante Škorić

Title of Thesis

Realisation and comparison of a serverless and a container-based microservice architecture for client reporting

Keywords: Microservices, Serverless, Container, Performance, Scaling, Resilience, Container Orchestration, Kubernetes, AWS Lambda, Cloud, Cloud Native Computing, Kafka, Client-Reports

Abstract

Microservices and serverless deployment strategies differ in many ways. With serverless, the cloud provider takes care of scaling and provides additional resilience. In contrast, with the container-based microservice application, the containers must be scaled using a container orchestration system. In this work, we answer the following research question: "How do performance (scalability, resilience, and cloud latency) and cost differ between a serverless deployment strategy using AWS Lambda and a container-based microservices deployment strategy using Docker and Kubernetes?" By answering this research question, use cases emerge where one deployment strategy may be preferred over the other. To answer the research question, performance tests were conducted and a cost calculation

was performed. This was done using two different use cases, an event-driven reporting application and an employee time sheet management portal application. AWS Lambda was used to implement and deploy the serverless applications, and Docker and Kubernetes were used for the microservice applications. The experiments conducted show that both deployment strategies have advantages and disadvantages. The disadvantages of the microservices deployment strategy are that it suffers from the load balancing and traffic distribution issues. Scalability and resource configuration is also ”more complex” than that of serverless. The advantages are that it provides better request handling performance compared to serverless. The disadvantage of the serverless deployment strategy is that it suffers from the cold start problem. The advantages are that it is more agile in terms of scalability than the microservices deployment strategy and it incurs less cost for the low traffic applications. From these findings, use cases (recommendations) emerge in which one deployment strategy may be preferred over the other. The microservices deployment strategy is suitable for real-time data processing applications and applications where fractions of seconds matter. In contrast, the serverless deployment strategy is suitable for the applications with traffic fluctuations, job schedulers and automatically scalable APIs. At this point, it is important to point out that the recommendations have been made based on the criteria from this work and no other criteria have been considered. When selecting a deployment strategy in practice, other criteria not covered in this work should also be considered.

Inhaltsverzeichnis

Glossar

- Resilience nach [MMK17] - ”abgestimmt auf die Softwareentwicklung, ist die Fähigkeit einer Komponente wie z. B. eines Betriebssystems (OS), eines Servers, eines Netzwerks, eines Rechenzentrums oder eines Speichersystems, sich schnell wiederherzustellen und mit dem Betrieb fortzufahren, selbst wenn bestimmte Ausfälle oder Fehler mit begrenzter Unterbrechung auftreten, die andernfalls den Ablauf der im Computer laufenden Operationen beeinträchtigen könnten”
- Skalierbarkeit nach [Tec17] - ”ist ein Attribut, das die Fähigkeit eines Prozesses, eines Netzwerks, einer Software oder einer Organisation beschreibt, zu wachsen und eine erhöhte Nachfrage zu bewältigen”
- Latency nach [GaroJ] - ”Maß für die Reaktionsfähigkeit eines Netzwerks, oft ausgedrückt als Round-Trip-Time (in Millisekunden), d. h. die Zeit zwischen dem Auslösen eines Netzwerk-Requests und dem Erhalt eines Responses”
- Availability nach [Her15] - ”bezieht sich auf die Eigenschaft von Software, dass sie vorhanden und bereit ist, ihre Aufgabe zu erfüllen, wenn sie gebraucht wird. Dies ist eine weit gefasste Sichtweise und umfasst das, was normalerweise als Zuverlässigkeit bezeichnet wird (obwohl es auch zusätzliche Aspekte wie Ausfallzeiten aufgrund regelmäßiger Wartung einschließen kann).”
- Amazon Web Services (AWS) nach [AWSoJj] - ”ist die weltweit umfassendste und am weitesten verbreitete Cloud-Plattform, die mehr als 200 voll funktionsfähige Service in Rechenzentren auf der ganzen Welt anbietet. Millionen von Kund*innen - darunter die am schnellsten wachsenden Start-ups, die größten Unternehmen und führende Regierungsbehörden - nutzen AWS, um Kosten zu senken, agiler zu werden und schneller zu innovieren.”
- Amazon Elastic Container Registry (AWS ECR) nach [AWSoJe] - ”ist eine vollständig verwaltete Container-Registry, die hochleistungsfähiges Hosting bietet, sodass die Anwendungsimages und Artefakte überall zuverlässig deployt werden können”
- Amazon Elastic Kubernetes Service (AWS EKS) nach [AWSoJi] - ”ist ein verwalteter Container-Service zur Ausführung und Skalierung von Kubernetes-Anwendungen in der Cloud oder on-premises”

- Amazon Virtual Private Cloud (AWS VPC) nach [AWSoJh] - ”ist ein Service, mit dem Sie AWS-Ressourcen in einem logisch isolierten virtuellen Netzwerk starten können, das Sie definieren”
- AWS Load Balancer bzw. Elastic Load Balancing nach [AWSoJc] - ”verteilt den eingehenden Anwendung-Traffic automatisch auf mehrere Ziele und virtuelle Anwendungen in einer oder mehreren Availability Zones (AZs)”
- AWS API Gateway nach [AWSoJa] - ”ist ein vollständig verwalteter Service, der es Entwickler*innen leicht macht, APIs in beliebigem Umfang zu erstellen, zu veröffentlichen, zu pflegen, zu überwachen und zu sichern. APIs dienen als ”Eingangstür” für Anwendungen, die auf Daten, Geschäftslogik oder Funktionen von Backend-Services zugreifen. Mit API Gateway können RESTful-APIs und Web Socket-APIs erstellt werden, die Anwendungen eine Echtzeit-Zwei-Wege-Kommunikation ermöglichen.”
- AWS CloudWatch nach [AWSoJb] - ”ist ein Monitoring- und Observabilityservice, der für DevOps-Ingenieur*innen, Entwickler*innen, Site Reliability Ingenieur*innen (SREs), IT-Manager*innen und Product Owner*innen entwickelt wurde”
- AWS Aurora nach [AWSoJg] - ”ist eine MySQL- und PostgreSQL-kompatible relationale Datenbank für die Cloud, die die Performance und Availability herkömmlicher Unternehmensdatenbanken mit der Einfachheit und Kosteneffizienz von Open-Source-Datenbanken kombiniert”
- Amazon Elastic Compute Cloud (AWS EC2) nach [AWS21b] - ”bietet skalierbare Rechenkapazität in der Amazon Web Services (AWS) Cloud”
- Amazon Elastic Container Service (AWS ECS) nach [AWSoJf] - ”ist ein vollständig verwalteter Container-Orchestrierungsservice, der das Deployment, Verwaltung und Skalierung von containerisierten Anwendungen erleichtert”
- AWS Pricing Calculator nach [AWS21c] - ”mit dem AWS Pricing Calculator können Sie AWS Services entdecken und eine Schätzung der Kosten für Ihre Use Cases auf AWS erstellen”
- AWS Free Tier nach [AWSoJd] - ”bietet Kund*innen die Möglichkeit, AWS Services bis zu bestimmten Grenzen für jeden Service kostenlos zu erkunden und auszuprobieren”

- REST-API nach [RH20] - ”ist eine Anwendungsprogrammierschnittstelle (API oder Web-API), die den Beschränkungen des REST-Architekturstils entspricht und die Interaktion mit RESTful-Webservices ermöglicht. REST steht für Representational State Transfer und wurde von dem Informatiker Roy Fielding entwickelt.”

- Monolith nach [Mic21] - ”bei einer monolithischen Anwendung handelt es sich in der Regel um ein Anwendungssystem, bei dem alle relevanten Module in einer einzigen, einsatzfähigen Ausführungseinheit zusammengefasst sind. Dabei kann es sich beispielsweise um eine Java-Webanwendung (WAR) handeln, die auf Tomcat läuft, oder um eine ASP.NET-Anwendung, die auf IIS läuft. Eine typische monolithische Anwendung verwendet ein mehrschichtiges Design mit separaten Schichten für die UI, die Anwendungslogik und den Datenzugriff.”

- Traffic nach [Tec15] - ”bezeichnet die Datenmenge, die sich zu einem bestimmten Zeitpunkt über ein Netzwerk bewegt”

- Cold-Start nach [Bes21] - ”wenn der Lambda-Service ein Request zur Ausführung einer Funktion über die Lambda-API erhält, bereitet der Service zunächst eine Ausführungsumgebung vor. In diesem Schritt lädt der Service den Code für die Funktion herunter, der in einem internen Amazon S3-Bucket (oder in Amazon Elastic Container Registry, wenn die Funktion Container-Packaging verwendet) gespeichert ist. Anschließend wird eine Umgebung mit dem angegebenen Speicher, der Runtime und der Konfiguration erstellt. Nach der Fertigstellung führt Lambda jeglichen Initialisierungscode außerhalb des Event-Handlers aus, bevor der Handler-Code ausgeführt wird.”

- Happy-Path nach [Kho21](S.188) - ”der Flow, der ein erfolgreiches Geschäftsszenario beschreibt”

- Virtualisierungskonzepte nach [AzuoJb] - ”schafft eine simulierte oder virtuelle Computerumgebung im Gegensatz zu einer physischen Umgebung. Die Virtualisierung umfasst oft computergenerierte Versionen von Hardware, Betriebssystemen, Speichergeräten und mehr.”

- Virtuelle Maschinen nach [AzuoJa] - ”im Allgemeinen kurz VM genannt, unterscheidet sich nicht von einem anderen physischen Computer wie einem Laptop, Smartphone oder Server. Sie hat eine CPU, Arbeitsspeicher, Festplatten zum Speichern Ihrer Dateien und kann bei Bedarf eine Verbindung zum Internet herstellen. Während die Teile, aus denen Ihr Computer besteht (Hardware genannt), physisch

und greifbar sind, werden VMs oft als virtuelle Computer oder softwaredefinierte Computer innerhalb physischer Server betrachtet, die nur als Code existieren."

- Structured Query Language (SQL) nach [Tec21] - "ist eine Programmiersprache, die in der Regel in relationalen Datenbanken oder Datenstreammanagementsystemen verwendet wird"
- YAML nach [OBKN09] - "ist eine menschenfreundliche, sprachen übergreifende, auf Unicode basierende Datenserialisierungssprache, die auf den gängigen nativen Datentypen agiler Programmiersprachen basiert. YAML eignet sich für eine Vielzahl von Programmieranforderungen, von Konfigurationsdateien über Internet-Messaging und Objektpersistenz bis hin zur Datenprüfung."
- Datawrapper nach [VisoJ] - "ist ein benutzerfreundliches Open-Source-Webtool, mit dem Sie einfache interaktive Diagramme erstellen können"
- kubectl nach [Kub21e] - "mit dem Command-Line-Tool kubectl kann der Kubernetes-Cluster kontrolliert werden"
- InfluxDB nach [Inf21] - "ist eine Open-Source-Time-Series-Plattform. Sie umfasst APIs für die Speicherung und Abfrage von Daten, die Verarbeitung im Hintergrund für ETL- oder Monitoring- und Alarmierungszwecke, Dashboards sowie die Visualisierung und Erkundung der Daten und mehr."
- Grafana nach [GraoJ] - "ist ein vollständiger Observability-Stack, mit dem Metriken, Protokolle und Traces überwachen und analysieren werden können. Es ermöglicht, die Daten abzufragen, zu visualisieren, zu alarmieren und zu verstehen, unabhängig davon, wo sie gespeichert sind."
- Docker-Compose nach [Doc21] - "ist ein Tool zur Definition und Ausführung von Multi-Container-Docker-Anwendungen"
- Flask-Framework nach [Pal21] - "ist ein leichtgewichtiges WSGI-Webanwendungs-Framework. Es wurde entwickelt, um den Einstieg schnell und einfach zu machen, mit der Fähigkeit zur Skalierung auf komplexe Anwendungen."

1 Einleitung

1.1 Motivation, Zielsetzung und Fragestellung

Martin Flower schreibt in seinem Artikel [Fow19], dass die Microservice-Architektur ein Ansatz für die Entwicklung einer einzigen Anwendung in Form einer Reihe von Services, wobei jeder Service seinen eigenen Prozess hat und die Kommunikation über eine HTTP-API erfolgt.
In der Praxis haben sich Microservices bereits durchgesetzt. Viele große Unternehmen wie Netflix [Kha19], Uber [Glu20], eBay [Gon19] und Zalando [Kol17] nutzen Microservices zusammen mit Containern und Orchestrierungssystemen.
Serverless ist eine weitere Technologie, die langsam an Bedeutung gewinnt, aber noch nicht so weit verbreitet ist wie Microservices. Serverless bietet eine Plattform für die Entwicklung von Anwendungen ohne die Verwaltung einer Infrastruktur [NT20].
Der Unterschied zwischen den beiden Technologien besteht darin, dass der Serverless-Anbieter die Skalierung übernimmt und einer Serverless-Anwendung zusätzliche Resilience bietet. Bei einer containerisierten Microservice-Anwendung müssen die Container mithilfe eines Container-Orchestrierungssystem wie Kubernetes orchestriert werden.
Es stellt sich die Frage, welche der beiden Technologien in welchem Use Case eingesetzt werden sollte, bzw. wann eine Technologie der anderen vorzuziehen ist.
Dies führt auch zum Ziel der Arbeit: zu ermitteln, wie sich die Performance und Kosten zwischen einer Serverless-Deploymentstrategie mit AWS Lambda und einer containerbasierten Microservices-Deploymentstrategie mit Kubernetes und Docker unterscheiden. In diesem Fall wird die Performance durch Skalierbarkeit, Resilience und Cloud Latency definiert.
Dies hilft auch dabei, zu ermitteln, welche Vor- und Nachteile Serverless und Microservices in Bezug auf die Kosten, Skalierbarkeit, Resilience und Cloud Latency haben. Durch die Beantwortung dieser Fragestellung ergeben sich die Use Cases, in denen eine Technologie gegenüber der anderen bevorzugt eingesetzt werden kann. Diese Use Cases beziehen sich jedoch nur auf die in dieser Arbeit verwendeten Kriterien (Kosten, Skalierbarkeit, Resilience und Cloud Latency).

Für die Evaluation der Performance und der Kostenberechnung werden zwei Use Cases verwendet. Die Anwendung aus dem Paper "Microservices vs. Serverless: A Performance Comparison on a Cloud-native Web Application" [FJG20] und eine event-driven Reporting-Anwendung.

Die Anwendung aus dem Paper ist ein Employee-Time-Sheet-Management-Portal. Diese Anwendung wird für die Verwaltung der Stundenzettel der Mitarbeiter*innen verwendet. Der zweite Use Case ist die event-driven Reporting-Anwendung. Diese wird in Zusammenarbeit mit dem Unternehmen "HanseCom Public Transport Ticketing Solutions GmbH" realisiert.
HandyTicket Deutschland ist ein Teil von HanseCom und bietet den HanseCom Verkehrsunternehmen die Möglichkeit, ihren Kund*innen digitale Tickets per App oder Web zu verkaufen. Neben dem Ticketverkauf ist eine der Hauptaufgaben der HandyTicket-Deutschland-Plattform die monatliche Bereitstellung von verkehrsunternehmensspezifischen Vertriebsabrechnungen.

Derzeit werden diese Reports in verschiedenen Komponenten des HandyTicket-Deutschland-Systems erzeugt. Einige der Reports werden im monolithischen Backend-System generiert, welches im Laufe der Jahre architektonische Erosion erlebt hat und so immer schlechter wartbar geworden ist. Der Rest der Reports wird über verschiedene crongesteuerte PHP-, Python- und Shell-Skripte erzeugt. Ein Teil der Dateien wird per E-Mail an die Verkehrsunternehmen gesendet und der Rest wird auf dem Server abgelegt, auf dem die Verkehrsunternehmen Zugriff haben.
Die Komplexität der Architektur führt zu Fehlern und Differenzen in den Statistiken. Die Fehlerbehebung nimmt viel Zeit in Anspruch, da das Logging unzureichend ist und es kein Monitoring gibt.

Eine neue Komponente des HandyTicket-Deutschland-Systems ist der registrierungsfreie Ticketkauf, mit einer vom legacy Backend-System getrennten event-driven Architektur. Dieses System ermöglicht es den Benutzer*innen, Tickets ohne vorherige Registrierung und Anmeldung zu kaufen. Für dieses System wurde noch kein Reporting-System implementiert. Die oben erwähnte event-driven Reporting-Anwendung bezieht sich auf das Reporting-System für den registrierungsfreien Ticketkauf. In dieser Arbeit wird der erste Prototyp des Systems gebaut und zur Kosten- und Performancemessung verwendet.

1.2 Gliederung

Die Arbeit ist in vier Teile gegliedert.
Der erste Teil besteht aus den ersten vier Kapiteln. Im ersten Kapitel werden die Motivation, Zielsetzung und die Fragestellung der Arbeit vorgestellt. Das zweite Kapitel

der Arbeit befasst sich mit den wesentlichen Konzepten und der Terminologie, die als Grundlage für die folgenden Kapitel dienen. Darüber hinaus werden in Kapitel 3 verwandte Arbeiten vorgestellt und in Kapitel 4 wird das Konzept des Performance-Testings beschrieben, das in den Kapiteln 5 und 6 verwendet wird.
Der zweite Teil der Arbeit beschäftigt sich mit dem ersten Use Case aus dem Paper "Microservices vs. Serverless: A Performance Comparison on a Cloud-native Web Application" [FJG20]. Die Anforderungen, das Setup, der Entwurf und die Ergebnisse werden vorgestellt und am Ende werden die Ergebnisse diskutiert.
Im dritten Teil der Arbeit wird die event-driven Reporting-Anwendung implementiert, die mittels KSQL, Statistiken bzw. Reports aus Kafka-Events erzeugt. Die Struktur des Kapitels ist ähnlich wie die des vorherigen Kapitels, es werden die Anforderungen, das Setup, der Entwurf, die Ergebnisse und die Diskussion der Ergebnisse vorgestellt.
Am Ende der Arbeit, bzw. im vierten Teil, wird ein Fazit für die gesamte Arbeit gezogen und ein Ausblick gegeben.

2 Grundlagen

In diesem Kapitel werden die Grundlagen der in dieser Arbeit verwendeten Konzepte vorgestellt. Das Kapitel dient der Verständlichkeit der folgenden Kapitel.

2.1 Microservices

Die Definition der Microservices nach [LF14] lautet:
"Der Microservice-Architekturstil ist ein Ansatz zur Entwicklung einer einzelnen Anwendung als eine Reihe kleiner Services, die jeweils in einem eigenen Prozess laufen und mit leichtgewichtigen Mechanismen, oft einer HTTP-Ressource-API, kommunizieren. Diese Services sind um Geschäftsfunktionen herum aufgebaut und können unabhängig voneinander von vollautomatischem Deployment eingesetzt werden. Es gibt nur ein Minimum an zentraler Verwaltung dieser Services, die in verschiedenen Programmiersprachen geschrieben sein können und unterschiedliche Datenspeichertechnologien verwenden."

Im Laufe der Zeit erhalten die Anwendungen immer mehr Features, was zu einer Vergrößerung der Codebase führt. Eine große Codebase führt zu Problemen. Monolithen haben eine große Codebase, und in manchen Fällen ist sie auch unübersichtlich. In einer solchen monolithischen Anwendung ist die Implementierung neuer Features und die Fehlerbehebung kompliziert [New15](S. 2).
Die Services innerhalb der Microservices-Landschaft werden innerhalb der "Business Boundaries" aufgebaut, d. h. ein Service ist nur für eine Funktionalität zuständig. Dadurch wird sichergestellt, dass die Codebase des Services nicht zu groß wird [New15](S. 2). Die Abbildung 1 zeigt eine Microservices-Landschaft mit drei Services. Es ist zu erkennen, dass jeder Service eine Funktionalität hat und dass sich diese in einer "Business Boundarie" befindet.
Microservices haben folgende Vorteile gegenüber einer traditionellen Softwarearchitektur:

- Heterogenität - da die Services unabhängig voneinander sind, können sie auch in unterschiedlichen Technologien implementiert werden [New15](S. 4-5).
- Resilience - die Microservices-Landschaft gewährleistet die Ausfallsicherheit des Systems. Wenn einer der Services ausfällt, wird dieses Problem isoliert und die

anderen Services sind nicht betroffen, was zu keinem Ausfall des Systems führt [New15](S. 5).

- Skalierbarkeit - in einer monolithischen Anwendung werden alle Teile der Anwendung gleichzeitig skaliert. In der Microservices-Landschaft kann jeder Service unabhängig voneinander skaliert werden [New15](S. 5-6).
- Deployment - genau wie bei der Skalierung muss die gesamte Anwendung nicht gleichzeitig deployt werden. Jeder Service kann unabhängig von den anderen Services deployt werden [New15](S.6-7).

Die Microservices bringen nicht nur Vorteile, sondern auch Nachteile mit sich:

- Monitoring - da die Microservices-Landschaft aus mehreren Services besteht, müssen alle Services gemonitored werden, was die Komplexität des Systems erhöht [New15](S.155-156).
- Testing - da die Services voneinander getrennt sind, kann das System nicht wie ein traditioneller Monolith getestet werden. Dies führt zu einer erhöhten Komplexität, ein Beispiel dafür sind die End-to-End-Tests [New15](S.131, 138-139).

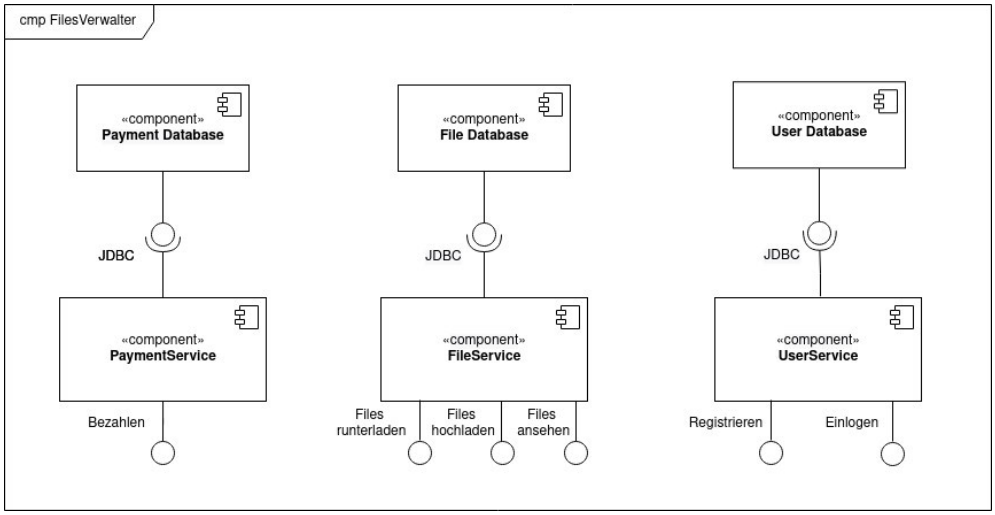

Abbildung 1: Beispiel einer Microservice-Architektur, UML nach [OMG17]

2.2 Serverless

Definition des Serverless Computing oder auch Serverless genannt, gemäß [NT20]:
"Serverless Computing bietet eine Plattform zur effizienten Entwicklung und Deployment von Anwendungen auf dem Markt, ohne dass eine zugrunde liegende Infrastruktur verwaltet werden muss. Verschiedene Serverless Computing-Plattformen wie AWS Lambda, Microsoft Azure Functions und Google Functions werden von großen Cloud-Anbietern angeboten. Solche Plattformen erleichtern und ermöglichen es den Entwickler*innen, sich mehr auf die Geschäftslogik zu konzentrieren, ohne sich um die Skalierung und Deployment der Infrastruktur kümmern zu müssen, da das Programm technisch auf externen Servern mit Unterstützung des Cloud-Service-Anbieters läuft."
In dieser Arbeit wird häufig der Begriff "Serverless" verwendet, der sich jedoch nur auf den Function as a Service (FaaS)-Teil des Serverless Computings bezieht.

Serverless ermöglicht eine einfache Entwicklung, da die Cloud-Anbieter die Serververwaltung, die Skalierung, das Deployment, die Fehlertoleranz, das Monitoring, das Logging und mehr übernehmen [JSSS+19].
Serverless Computing besteht aus zwei Teilen:

- Backend as a Service (BaaS) - bietet "off-the-shelf"-Services, d. h. Services, die vom Cloud-Anbieter angeboten werden und als solche genutzt werden können. Dies bedeutet, dass die Entwickler*innen nicht selber Backend-Services schreiben müssen. Dadurch ergibt sich, dass sich die Entwickler*innen auf das Frontend konzentrieren können und es nur verwalten müssen. Ein Beispiel für ein BaaS-System ist Google Firebase [NT20].
- Function as a Service (FaaS) - wird im nächsten Kapitel ausführlich beschrieben.

2.2.1 Function as a Service

Function as a Service ist ein Teil des Serverless Computings. FaaS bietet eine Umgebung, in der event-driven Funktionen ausgeführt werden [NT20]. Wie der Name schon sagt, werden die Funktionen durch ein Event ausgelöst, wobei die Events vom Cloud-Anbieter abhängen, das können z. B. HTTPS Request oder Timer-Events sein. Nach der Ausführung läuft jede Funktion in einer isolierten Umgebung. Mehrere Funktionen

bilden eine Anwendung.
FaaS bringt Vorteile mit sich, der Cloud-Anbieter sorgt für die horizontale Skalierung der Funktionen [NT20]. Da die Funktionen nicht die ganze Zeit laufen, wie z. B. bei einer Microservice-Anwendung, zahlen die Benutzer*innen nur für die Zeit, in der die Funktionen aktiv sind. Jede Funktion ist nur für eine bestimmte Zeit aktiv, z. B. 15 Minuten (AWS Lambda). Dies kann auch zu Problemen führen, wenn ein Prozess länger laufen muss, daher sind FaaS Anwendungen nicht für lang laufende Prozesse geeignet [NT20].
Für die Experimente in dieser Arbeit wurde AWS Lambda eingesetzt. AWS Lambda ist FaaS, der von Amazon Web Services angeboten wird. Der Source Code wird in einem isolierten Container deployt, für den Speicher, Festplattenplatz und CPU reserviert sind. Eine Funktion kann mehrfach parallel aufgerufen werden, und unter einer Funktion versteht man die Kombination aus Code, Konfiguration und Dependencies [Sba17](S. 9).
In dieser Arbeit wird Serverless-Framework verwendet. Es wird benutzt, um die Serverless-Anwendungen auf die Cloud-Infrastruktur zu deployen. Es verwendet eine YAML-Syntax für die Konfiguration des Deployments und unterstützt eine Reihe von Plugins zur einfacheren Entwicklung und Deployment der Anwendungen [Ser21].

2.2.2 Skalierungsalgorithmus

AWS Lambda verwendet bestimmte Konzepte und Algorithmen zur Skalierung der Funktionen. Diese werden in diesem Kapitel beschrieben.
Wenn eine Funktion zum ersten Mal aufgerufen wird, wird die Instanz der Funktion erstellt und der Event-Handler ausgeführt, um das Event zu verarbeiten. Die Funktion, die das Event verarbeitet hat, bleibt eine gewisse Zeit lang aktiv, da sie auch weitere Events verarbeiten soll. Wenn eine Funktion bereits ein Event verarbeitet und ein weiteres Event hinzukommt, wird eine weitere Instanz der Funktion erstellt, die das zweite Event verarbeitet. Das bedeutet, dass mehrere Instanzen der Funktion parallel laufen können. Wenn die Anzahl der Events abnimmt, werden die nicht verwendeten Instanzen gestoppt. Wie viele Funktionen parallel ausgeführt werden können, hängt von der Konfiguration der Funktion ab, der Standardwert ist 1.000 Funktionen. Es gibt keine Begrenzung für die Anzahl der Funktionen, die parallel ausgeführt werden können, aber die Konfiguration muss angepasst werden, um mehr als 1.000 Funktionen parallel auszuführen [AWS21a].
Wenn eine neue Instanz einer Funktion erstellt wird, wird der erste Aufruf dieser Instanz verzögert, da der Code geladen und initialisiert werden muss. Um dies zu umgehen, kann die Konfiguration der bereitgestellten Parallelität (provisioned concurrency) verwendet

werden [AWS21a].

2.3 Container und Container Orchestration

Der Artikel von Docker [DocoJb] enthält die Definition von Container:
"Ein Container ist eine Standard-Softwareeinheit, die den Code und alle seine Dependencies zusammenfasst, damit die Anwendung schnell und zuverlässig in einer anderen Computerumgebung ausgeführt werden kann. Ein Docker-Container-Image ist ein leichtgewichtiges, in sich geschlossenes, ausführbares Softwarepaket, das alles enthält, was zur Ausführung einer Anwendung erforderlich ist: Code, Runtime, Systemtools, Systembibliotheken und Einstellungen."
In dem Paper von Netflix [LSB17] werden die Erfahrungen mit Containern und deren Vorteile vorgestellt. Im Folgenden sind die Vorteile von Containern aus dem Paper aufgeführt:

- End-to-end Anwendungspackaging
- Flexibles Packaging
- Einfache Cloud-Abstraktion
- Container sorgen für schnellere und effizientere Verwaltung der Cloud-Ressourcen

Container sehen auf den ersten Blick ähnlich aus wie virtuelle Maschinen, aber es handelt sich um zwei unterschiedliche Virtualisierungskonzepte. Die Container virtualisieren das Betriebssystem und nicht die Hardware. Mehrere Container können auf demselben Server bzw. Rechner ausgeführt werden und denselben Betriebssystem-Kernel verwenden. Im Gegensatz zu VMs benötigen Container weniger Speicherplatz, was bedeutet, dass sie in den meisten Fällen nur wenige MB groß sind [DocoJb].
Ein Beispiel für die Containerisierung ist Docker.
Um einen Docker-Container zu erstellen, muss zunächst ein Docker-Image erstellt werden. Das Docker-Image ist ein read-only Template, welches zur Erstellung des Docker-Containers verwendet wird. Das bedeutet, dass ein Docker-Container eine Instanz des Docker-Images ist. Dieser Container kann erstellt, gestartet, gestoppt und gelöscht werden [DocoJa]. Container müssen auch verwaltet werden, wofür das Konzept der Container-Orchestrierung verwendet wird. Einige der für die Container-Orchestrierung verwendeten Tools sind Borg, Omega und Kubernetes [BGO+16].
In dieser Arbeit wird für die Container-Orchestrierung das Tool Kubernetes verwendet.

In der Kubernetes-Dokumentation wird Kubernetes wie folgt definiert: "Kubernetes ist eine portable, erweiterbare Open-Source-Plattform für die Verwaltung von Container-Workloads und -Services, die sowohl eine deklarative Konfiguration als auch eine Automatisierung ermöglicht. Sie verfügt über ein großes, schnell wachsendes Ökosystem. Kubernetes-Dienste, -Support und -Tools sind weithin verbreitet [Kub21h]."
Um Kubernetes zu verwenden, muss es zunächst deployt werden; nach dem Deployment wird ein Cluster erstellt. Auf dem Cluster laufen Worker-Maschinen, sogenannte Nodes. Die Nodes bestehen aus Pods bzw. auf den Nodes werden die Pods erstellt, die ein Teil des Anwendung-Workloads sind. Sie enthalten die laufenden Container. Um die Pods und Nodes im Cluster zu verwalten, verwendet Kubernetes den Control-Plane. Control-Plane ist ein Container-Orchestrierung-Layer, welches die API zum Deployment und zur Verwaltung von Containern bereitstellt [Kub21d].
Die Kubernetes API ist selbst ein Teil des Control-Planes. Sie bietet eine HTTP-API, die von den Entwickler*innen und anderen Komponenten in Kubernetes verwendet wird, um die Objekte in Kubernetes zu verwalten und abzufragen [Kub21f].
Kubernetes-Objekte [Kub21g] sind als Entitäten im Kubernetes-System definiert. Sie werden verwendet, um den Zustand des Clusters zu definieren. Sie beschreiben die folgenden Zustände:

- Anwendungen, die auf dem Node laufen
- Ressourcen, die diese Anwendungen nutzen
- Richtlinien, die das Verhalten dieser Anwendungen regeln

Diese werden, im .YAML-Format definiert.

2.3.1 Skalierungsalgorithmus

In diesem Kapitel werden die Skalierungsalgorithmen und -konzepte von Kubernetes vorgestellt.
Kubernetes verfügt über drei Skalierungsalgorithmen:

- Horizontal Pod Autoscaling nach [Kub21c] - aktualisiert die Workload-Ressource, z. B. ein Deployment, mit dem Ziel, das Workload automatisch an den Bedarf - z. B. die Anzahl des Traffics - anzupassen.

- Vertical Pod Autoscaler nach [Kub21b] - passt die Ressourcen eines Containers automatisch an die Auslastung an. Dies ermöglicht die Planung von Pods auf Nodes und gewährleistet, dass eine angemessene Menge an Ressourcen für jeden Pod verfügbar ist.
- Cluster Autoscaler nach [Kub21a] - ist ein Tool, mit dem der Kubernetes-Cluster auf- und abwärts skaliert werden kann.

In dieser Arbeit wird Horizontal Pod Autoscaling verwendet.
Wenn die Last einer Anwendung steigt, skaliert der Algorithmus einen weiteren Pod. Dies ist das Gegenteil von Vertical Pod Autoscaler, da der Algorithmus auf eine erhöhte Last mit einer Zuweisung von mehr Ressourcen an den Pod reagiert. Diese beiden Algorithmen sind nicht für die gleichzeitige Verwendung vorgesehen [Kub21c].
Der Algorithmus besteht aus zwei Teilen: der Kubernetes-Ressource und dem Controller. Der Controller ist ein Control-Loop, der das System reguliert. Das bedeutet, dass er mit Unterbrechungen läuft. Dieser Loop wird alle 15 Sekunden ausgeführt, aber dies kann konfiguriert werden.
Der Controller-Manager fragt die Ressourcenauslastung in jedem Loop anhand der in der HorizontalPodAutoscaler-Definition angegebenen Metriken und Pods ab. Diese Metriken werden über die API für selbst definierte Metriken oder die Ressourcenmetriken-API abgefragt. Die Pod-spezifischen Metriken werden von der Ressourcenmetriken-API abgerufen, alle anderen von der API für selbst definierte Metriken. Die von der Ressourcenmetrik-API abgerufenen Metriken werden mit den im HorizontalPodAutoscaler konfigurierten Ressourcen verglichen. Dieser Vergleich wird nur durchgeführt, wenn im HorizontalPod-Autoscaler ein Auslastungszielwert definiert wurde. Wenn der Auslastungszielwert festgelegt wurde, wird der Auslastungswert von dem Controller als Prozentsatz der entsprechenden Ressourcenanforderung an die Container in jedem Pod berechnet. Im HorizontalPodAutoscaler kann auch ein Rohwert angegeben werden; in diesem Fall werden die rohen metrischen Werte direkt verwendet. Der Durchschnitt der Auslastung oder der Rohwert wird von dem Controller verwendet, um ein Verfahren zur Skalierung der Anzahl der gewünschten Replikate der Pods zu erstellen [Kub21c].
Die Skalierungskonfiguration wird durch die folgenden beiden Konfigurationsfunktionen angepasst: scaleDown und scaleUp. ScaleDown wird verwendet, um die Instanzen nach unten zu skalieren, und scaleUp, um sie nach oben zu skalieren.
In beiden Konfigurationen kann StabilizationWindowSeconds angepasst werden. Dieser wird verwendet, um sich die historisch empfohlenen Größen zu merken, und der Algorithmus arbeitet nur mit der niedrigsten Größe innerhalb dieses Zeitfensters.

Horizontal Pod Scaler arbeitet mit Ressourcen, diese werden dem Container zugewiesen. Es unterscheiden sich zwei Werte: requests und limits. Requests ist die Menge an Ressourcen, die das System für den Container garantiert, und Kubernetes verwendet diesen Wert, um zu entscheiden, auf welchem Node der neue Pod platziert werden soll. Das Limit ist die maximale Menge an Ressourcen, die Kubernetes dem Container zur Verfügung stellt [Kub21c].

2.4 Apache Kafka und KSQL

In diesem Kapitel werden Apache Kafka und KSQL vorgestellt. Diese beiden Anwendungen werden in Kapitel 6 "Customer-Reporting-System" verwendet.
Kafka basiert auf dem Konzept des Event-Streamings. Event-Streaming wird in der Dokumentation von Apache Kafka [KafoJ] wie folgt definiert: "Event-Streaming ist das digitale Äquivalent zum zentralen Nervensystem des menschlichen Körpers. Es ist die technologische Grundlage für die "Always-on"-World, in der Unternehmen zunehmend softwaredefiniert und automatisiert sind und in der Benutzer*innen von Software mehr Software sind. Technisch gesehen ist Event-Streaming das Erfassen von Daten in Echtzeit aus Event-Sources wie Datenbanken, Sensoren, mobilen Geräten, Cloud-Diensten und Softwareanwendungen in Form von Event-Streams. Das dauerhafte Speichern dieser Event-Streams für den späteren Abruf, das Manipulieren, Verarbeiten und Reagieren auf die Event-Streams sowohl in Echtzeit als auch im Nachhinein und das Weiterleiten der Event-Streams an verschiedene Zieltechnologien nach Bedarf. Das Event-Streaming gewährleistet somit einen kontinuierlichen Datenfluss und eine kontinuierliche Interpretation der Daten, damit die richtigen Informationen zur richtigen Zeit am richtigen Ort sind.".
Zur Umsetzung des Event-Streaming-Konzepts kann Kafka verwendet werden.
Kafka ist ein publish/subscribe Messaging-System. Es wird oft als verteiltes Commit-Log oder verteilte Streaming-Plattform betrachtet. Die Daten in Kafka werden in einer Reihe dauerhaft gespeichert, diese können von anderen Anwendungen deterministisch gelesen werden. Diese können auf verteilte Weise im System gespeichert werden, wodurch Kafka widerstandsfähig und skalierbar ist [NSP17](S.4).
Die Daten, die in Kafka gespeichert werden, werden als Messages bzw. Events bezeichnet. Diese werden in Kafka als Batches gespeichert. Batch ist eine Menge von Events, die alle im selben Topic bzw. Partion gespeichert werden.
Events sind Arrays von Bytes, die für Kafka keine spezifische Form oder Bedeutung ha-

ben. Sie können ein Schema haben und dies wird empfohlen, da andere Anwendungen die Events deserialisieren müssen. Schemata können durch verschiedene Datenformate wie JSON oder XML realisiert werden [NSP17](S.4-5).
Die Events werden in Topics gespeichert. Jedes Topic kann aus mehreren Partitionen bestehen. Um die Topics zu verstehen, können sie mit Tabellen in einer relationalen Datenbank verglichen werden. Da die Topics aus mehreren Partitonen bestehen, sorgt dies für Skalierbarkeit in Kafka. Die an Kafka angeschlossenen Anwendungen lesen die Events vom Anfang einer Partition [NSP17](S.5-6).
Producer und Consumer sind zwei Arten von Anwendungen, die mit Kafka verbunden werden können. Producer erstellen und speichern die Events in Topics. In anderen Systemen werden sie auch Publisher und Writer genannt.
Consumer lesen die Events aus Topics. In anderen ähnlichen Systemen werden sie Subscriber und Reader genannt [NSP17](S.6-7).
Der Server wird als Kafka Broker bezeichnet. Er ist für den Empfang von Events von Producer und deren Speicherung auf der Festplatte zuständig. Außerdem ist er für den Empfang der Consumer-Requests und die Übermittlung der Events an die Consumer zuständig. Ein System kann aus mehreren Kafka-Brokern bestehen, dies wird als Kafka-Cluster bezeichnet [Mol09](S.7-9).

Um die Events aus den Topics zu lesen und zu verarbeiten, kann KSQL verwendet werden. In der Confluent Dokumentation [Inc21f] wird KSQL wie folgt definiert: ”KSQL ist die Streaming-SQL-Engine für Apache Kafka. Sie bietet eine einfach zu bedienende und dennoch eine leistungsstarke, interaktive SQL-Schnittstelle für die Stream-Verarbeitung auf Kafka, ohne dass Sie Code in einer Programmiersprache wie Java oder Python schreiben müssen. KSQL ist skalierbar, elastisch, fehlertolerant und in Echtzeit. Es unterstützt eine breite Palette von Streaming-Operationen, einschließlich Datenfilterung, Transformationen, Aggregationen, Joins, Windowing und Sessionization”.
KSQL verwendet SQL, aber die Konzepte von KSQL unterscheiden sich von den Konzepten einer traditionellen, relationalen Datenbank.
Das KSQL-Konzept basiert auf dem Konstrukt von Tabellen und Streams. Beide werden zum Speichern von Daten verwendet und nutzen ein Key-Value-Modell. Tabellen sind mutable und repräsentieren den letzten Zustand der Daten. Tabellen können bei den Use Cases verwendet werden, in denen Änderungen über die Zeit dargestellt werden müssen oder in denen Aggregationen von Events wichtig sind.
Das Gegenteil von Tabellen sind Streams. Sie sind immutable und unterstützen das Append-Only-Modell. Streams werden verwendet, um eine Reihe historischer Fakten zu

zeigen und um Daten zu sehen, die sich im Laufe der Zeit ändern [Inc21a].
Um die Daten aus den Tabellen und Streams abzurufen, wird das Konzept von Persistent-, Push- und Pull-Queries verwendet.
Persistent-Queries werden verwendet, um die Daten aus den Streams zu verarbeiten. Die Queries können über Events kontinuierliche Berechnungen ausführen. Die Events können transformiert, gefiltert, aggregiert und zusammenführt werden, um die neuen Collections oder Materialized-Views abzuleiten.
Push-Queries abonnieren Streams, die geänderten Daten in Streams werden an den Push-Query übermittelt. Push-Queries können bei asynchronen Anwendungsabläufen verwendet werden. Im Gegensatz zu Push-Queries werden die Pull-Queries verwendet, um den aktuellen Zustand einer Materialized-View abzurufen. Die Push-Queries sind besser für die Request/Response-Flows geeignet [Inc21a].

3 Verwandte Arbeit

In diesem Kapitel werden verwandte Arbeiten vorgestellt, die drei Themen behandeln: Architektonische Entscheidung zwischen Serverless und Microservices, Performance von Serverless-Anwendungen und Performance von containerbasierten Microservices, die über Kubernetes orchestriert werden.

An der Universität Vels wurde 2018 das Paper "Architecture Decision on using Microservices or Serverless Functions with Containers" [JY18] veröffentlicht. Das Paper analysierte drei Technologien: Container, Serverless und Microservices. Es gab Empfehlungen, welche Technologie verwendet werden sollte. Dabei wurde festgestellt, dass jede Technologie ihre Vor- und Nachteile hat. Die Technologien sollten je nach Use Case und Geschäftsanforderungen ausgewählt werden. Alle drei Technologien sind ein wesentlicher Bestandteil einer cloud-based Solution. Durch die Wahl der richtigen Technologie kann es zur Minimierung der Kosten und Implementierung einer skalierbaren und sicheren Anwendung kommen. Die Ergebnisse sind für diese Arbeit wichtig, da diese Technologien auch in dieser Arbeit verwendet wurden.

Für die Performancemessung in den Kapiteln 5 und 6 der Arbeit ist die Availability von Microservices die auf Kubernetes laufen ein wichtiger Faktor, und das Paper "Deploying Microservice Based Applications with Kubernetes: Experiments and Lessons Learned" [AVSTK18] beschäftigt sich mit dieser Frage. In diesem Paper wurde die Availability einer Microservice-Anwendung untersucht, die über Kubernetes orchestriert wird. Es wurde festgestellt, dass Kubernetes-Fehlerbehebungsaktionen oft durch interne Operationen ausgewertet werden. Bei dieser Art von Operationen reagiert Kubernetes im Vergleich zu Ausfällen, die durch externe Vorgänge ausgelöst werden, mit hoher Qualität. Die Arbeit zeigt auch, dass die Standardkonfiguration von Kubernetes im Falle eines extern ausgelösten Node-Ausfalls zu einem signifikanten Service-Ausfall führt. Dies bedeutet, dass Microservices und Kubernetes mit der Kubernetes-Standardkonfiguration nicht für hochverfügbare Software geeignet sind.

Ein weiteres Paper, das sich mit Microservices und der Performance von Kubernetes beschäftigt, ist "Auto-scaling of Containers: the Impact of Relative and Absolute Metrics" [CP17]. Darin wurde der Horizontal Pod Autoscaling Algorithmus von Kubernetes analysiert, der mit relativen Metriken arbeitet. Außerdem wird ein neuer Autoscaling Algorithmus vorgeschlagen, der mit absoluten Metriken arbeitet. Es wurde festgestellt,

dass die "Response Time" mit dem neuen Algorithmus um einen Faktor zwischen 0,5 und 0,66 im Vergleich zum aktuellen Horizontal Pod Autoscaling Algorithmus von Kubernetes sinkt.

Andere verwandte Arbeiten befassen sich mit dem Thema Serverless. Dieses Thema ist auch ein wesentlicher Teil dieser Arbeit. Eines der Paper, die sich mit dem Thema befassen, ist "Serverless Computing: An Investigation of Factors Influencing Microservice Performance" [LRC+18]. In dieser Arbeit werden die Faktoren untersucht, die einen Einfluss auf die Performance der Serverless-Anwendungen mit einer Microservice-Architektur haben. Bei den Messungen wurden vier Zustände definiert: Provider Cold, VM Cold, Container Cold und Warm. Es wurde festgestellt, dass der Zustand der Funktion einen Einfluss von bis zu 15 x auf die Leistung haben kann. Die Ergebnisse des Papers können eine enorme Auswirkung auf die Ergebnisse dieser Arbeit haben, da die Messungen in dieser Arbeit als Provider Cold ausgeführt werden.

Das Paper "Serverless Computing: Design, Implementation, and Performance" [MB17] befasst sich mit dem Design, Implementierung und Performance von einer Performance orientierten Serverless Computing-Plattform. Das Paper erörtert Implementierungsherausforderungen wie Skalierbarkeit, Containererkennung, Lebenszyklus und Wiederverwendung. Im weiteren Verlauf des Papers werden Performancemessungen durchgeführt, diese zeigen, dass der Prototyp aus dem Paper einen besseren Durchsatz als andere Serverless-Plattformen aufweist.

Im Jahr 2018 wurde das Paper "Evaluation of Production Serverless Computing-Environments" [LSF18] veröffentlicht.
Die Autoren führten eine Performancemessung der Serverless-Plattformen (Amazon Lambda, Microsoft Azure Functions, Google Cloud Functions und IBM Cloud Functions (Apache OpenWhisk)) durch. Die Ergebnisse zeigen, dass die Elastizität von Amazon Lambda die anderen Anbieter in Bezug auf CPU-Leistung, Netzwerkbandbreite und Datei-I/O-Durchsatz übertrifft. Darüber hinaus wurde festgestellt, dass die Serverless-Skalierbarkeit effektiv ist und dass die Serverless-Anwendung kostengünstiger ist als die traditionellen Anwendungen, die auf einer VM laufen.

Im folgenden Kapitel wird das Paper "Microservices vs. Serverless: A Performance Comparison on a Cloud-native Web Application" [FJG20] vorgestellt. Das Paper wurde als Grundlage für diese Arbeit verwendet, in den Kapiteln 4, 5 und 6 werden die Methoden

und Anwendungen aus dem Paper verwendet.

3.1 Microservices vs Serverless: A Performance Comparison on a Cloud-native Web Application

Wie bereits erwähnt, als Basis für die Arbeit wurde das Paper "Microservices vs. Serverless: A Performance Comparison on a Cloud-native Web Application" [FJG20] verwendet. In diesem Kapitel werden die Thematik, Methodiken, Ergebnisse und Feststellungen des Papers vorgestellt.

Im Paper wurde eine Cloud-native Webanwendung unter den Aspekten Skalierbarkeit, Zuverlässigkeit, Kosten und Latenz analysiert, die sowohl als Serverless- als auch als Microservice-Anwendung deployt wurde. Am Ende des Papers, werden die architektonischen Empfehlungen gemacht, die auf der Art der Last, des Szenarios und der Größe der Requests basieren.
Bei der Webanwendung handelt es sich um ein Portal für die Verwaltung von Stundenzetteln für Mitarbeiter*innen, in dem die Zeit, die ein*eine Mitarbeiter*in an einem Projekt gearbeitet hat, erfasst wird. Die Anwendung besteht aus drei in Node.js geschriebenen Modulen: Favourite Projects, Timesheet und Miscellaneous.
Favourite Projects verwaltet die Lieblingsprojekte der Benutzer*innen. Es besteht aus 5 Endpoints:

- GET getAll - gibt alle Projekte zurück
- POST getByEmployeeId - gibt die Lieblingsprojekte eines Benutzers bzw. einer Benutzerin zurück, dafür muss im Request die User-ID mitgegeben werden
- CREATE projectFav-Service - erstellt ein Projekt
- UPDATE projectFav-Service - aktualisiert ein Projekt
- DELETE projectFav-Service - löscht ein Projekt

Timesheet besteht aus vier Endpoints und dient der Erfassung und Anzeige der von den Benutzer*innen für ein Projekt aufgewendeten Zeit. Benutzer*innen können CRUD-Operationen für jedes Projekt durchführen.
Miscellaneous ist zuständig für die Userverwaltung, Feiertagsinformationen der einzelnen Regionen und für die Verwaltung der Liste von Aktivitäten, die die Art der Rolle der

Benutzer*innen bei der Durchführung von Projekten darstellt. Alle diese Informationen sind in diesem Modul erfasst und werden durch drei Endpoints aufgerufen.
Die Anwendung wurde als Function as a Service und Microservices deployt, das Frontend der Anwendung ist gleich für beide Deployments geblieben und das Backend wurde an eine AWS Aurora MYSQL Datenbank angeschlossen. Bei der Microservice-Deploymentstrategie wurden Container verwendet, die über Amazon Elastic Container Service (ECS) orchestriert wurden, jedem Container wurden 0,25 CPU-Core und 512 MB Memory zugewiesen. Die Autoscaling Policy von ECS wurde aktiviert, was bedeutet, dass ECS zwei weitere Container hinzufügt, wenn die CPU-Auslastung für eine Minute über 50 % liegt, und zwei Container entfernt, wenn die CPU-Auslastung für mehr als eine Minute unter 30 % liegt.
Das Function as a Service Deployment verwendet AWS Lambda und AWS API Gateway als API-Zugangspunkt, der eingehenden Requests an die richtige AWS Lambda Funktion weiterleitet. Jeder Funktion wurden 512 MB Memory zugewiesen und die AWS Lambda-Concurrency-Quota wurde bei 1.000 Funktionen belassen.
Für die Performancemessung wurde k6 zusammen mit InfluxDB und Grafana verwendet. Es wurden drei Arten von Tests durchgeführt:

- incremental - ständig steigend
- random - zufällige Anzahl von Requests
- triangle - bis zur Hälfte der Zeit ansteigend und dann abnehmend für die restliche Zeit

12 API-Endpoints wurden mit drei verschiedenen Testtypen und zwei Deploymentstrategien getestet, was zu insgesamt 72 Tests führte.
Die Ergebnisse lassen sich in vier Punkten zusammenfassen:

- Function as a Service leidet unter dem Problem des Coldstarts. Coldstart ist die Einrichtungszeit, die benötigt wird, um eine Funktion zum Laufen zu bringen, wenn sie innerhalb eines bestimmten Zeitraums zum ersten Mal ausgeführt wird.
- Microservices leiden unter dem Problem des Lastausgleichs und der Neuverteilung des Traffics.
- Microservices haben eine bessere Performance als Serverless bei kleinen und sich wiederholenden Requests.

- Function as a Service ist agiler in Bezug auf die Skalierbarkeit, dies geschieht aus dem Grund, dass Microservices eine Scale-Up- und Scale-Down-Policy verwenden, die Policy-Kriterien müssen mindestens eine Minute lang erreicht werden, daher gibt es eine Verzögerung bei der Reaktion auf die aktuelle Arbeitslast.

Feststellung des Papers ist, dass die Microservices-Deploymentstrategie bei dauerhaften Operationen einen Kostenvorteil gegenüber Function as a Service bietet. Aufgrund der Skalierbarkeit von Function as a Service, hat es den Vorteil, dass es besser mit Requests umgehen kann, die eine große Response zurückgeben.

4 Performance-Testing

In den Kapiteln 5 und 6 werden Performance-Tests durchgeführt und die Ergebnisse vorgestellt. Um den Testaufbau und die Ergebnisse zu verstehen, wird in diesem Kapitel eine Einführung in das Performance-Testing gegeben und die in den beiden Kapiteln verwendeten Konzepte erläutert.
Performance-Testing wird nach [ISToJa] als folgend definiert: "Performance-Testing sind Tests zur Bestimmung der Leistungseffizienz einer Komponente oder Systems."
Eine andere Art von Tests, die für diese Arbeit von Bedeutung ist, sind die Loadtests. Load-Testing nach [ISToJb]: "Load-Testing ist eine Art von Performance-Tests, diese werden durchgeführt, um das Verhalten einer Komponente oder eines Systems unter verschiedenen Belastungen zu bewerten, normalerweise zwischen erwarteten Bedingungen mit geringer, typischer und Spitzenbelastung."
Für das Load-Testing wurde das Open-Source Load-Testing-Tool k6 verwendet. Das Tool k6 basiert auf dem Konzept der virtual User (VUs). VUs sind im Grunde parallele while(true)-Loops, die Skripte ausführen. Diese sind in Javascript geschrieben und definieren das Verhalten des Tests [Gra21]. Die k6 Ergebnisse werden auf der Konsole ausgegeben. Für die Arbeit wurde die Ausgabe erweitert. Die Ausgaben wurden in einer Instanz der InfluxDB Datenbank gespeichert, außerdem wurde das Tool Grafana mit der Datenbank verbunden und mithilfe von Grafana wurden die live Ergebnisse im Detail visualisiert. So war es möglich zu sehen, ob ein Test fehlerhaft ausgeführt wurde oder ob es Fehler in der Infrastruktur gab. Die Realisierung des Setups erfolgte über Docker und Docker-Compose. Weiterhin ist für das Performance-Testing die Testumgebung wichtig. Wie von Ian Molyneaux beschrieben [Mol09] (S.17), eine Performance-Testingumgebung soll sich der Deploymentumgebung ähneln. Daher wurde eine Umgebung eingerichtet, die über dieselben Ressourcen wie die Deploymentumgebung verfügt. Alle Tests wurden auf einem lokalen Rechner durchgeführt, wobei darauf geachtet wurde, dass jeder Test und jede Messung vom gleichen Netzwerk aus durchgeführt wurde.
Es wurden zwei Arten von Tests ausgeführt:

- incremental - ständig steigend
- triangle - bis zur Hälfte der Zeit ansteigend und dann abnehmend für die restliche Zeit

Eine Testart, die im Kapitel 3.1 vorgestellt wurde, aber nicht durchgeführt worden ist, ist die Testart "random". Dies geschah aus zwei Gründen nicht: Es lohnte sich finanziell

nicht und im Paper aus dem Kapitel 3.1 wurde auf die Testart nicht genauer eingegangen. Es wurden insgesamt 48 Tests durchgeführt. Für den in Kapitel 3.1 vorgestellten Use Case aus dem Paper "Microservices vs Serverless: A Performance Comparison on a Cloud-native Web Application" [FJG20] wurden 40 Tests und für die event-driven Reporting-Anwendung 8 Tests durchgeführt.
Die Anzahl der Tests wird wie folgt berechnet:
$2\ Durchlaeufe\ *\ ((5\ Endpoints\ *\ 2\ Deploymentstrategien(Serverless\ und\ Microservices) * 2\ Testarten) + (1\ Endpoint * 2\ Deploymentstrategien(Serverless\ und\ Microservices)\ * 2\ Testarten))$
Während des Testens wurde jeder Use Case separat getestet, alle Endpoints eines Use Cases wurden mit einer Testart und dann mit der anderen getestet, dies wurde noch einmal wiederholt und am Ende wurde der zweite Use Case mit der gleichen Vorgehensweise getestet. Nach jedem Test wurden die Daten in der Datenbank gelöscht und das Datenbankschema und die gemockten Datensätze wieder hinzugefügt. Zwischen den Tests wurde darauf geachtet, dass das System kalt war, um die Ergebnisse nicht zu verfälschen; zu diesem Zweck wurden Pausen eingelegt.
Außerdem ist es wichtig zu erwähnen, dass das System im kalten Zustand getestet wurde, weil der Coldstart von Lambda für die Ergebnisse der Arbeit wichtig war.
Die folgenden Metriken wurden für die Arbeit verwendet:

- HTTP req duration - ist die Zeit, die der Server zur Verarbeitung und Beantwortung des Requests benötigte, ohne die anfänglichen DNS-Such-/Verbindungszeiten.
  Für die Auswertung der HTTP req duration wurden folgende Metriken verwendet:
  - Minimum (Min) - zeigt den niedrigsten Wert in der Menge an
  - Maximum (Max) - zeigt den höchsten Wert in der Menge an
  - Median - ist der mittlere Datenpunkt, wenn der Datensatz vom kleinsten zum größten Wert geordnet ist
  - Percentile-95 - der Wert, unter den 95% der Daten fallen
  - Average (Avg) bzw. Mean - ist die Summe aller Werte geteilt durch die Anzahl der Werte
- average number of requests served per second (Avg. RPS) - durchschnittliche Anzahl von Requests pro Sekunde
- VUsm - Anzahl der virtual User

- total number of requests - Gesamtzahl der von k6 gesendeten Requests
- number of unsuccessful requests - Anzahl der erfolglosen Requests
- number of successful requests - Anzahl der erfolgreichen Requests
- Laufzeit in Sekunden
- Podscaling - die Anzahl der Anwendungsinstanzen, die auf Kubernetes laufen, und deren Verbrauch von CPU und Memory
- AWS CloudWatch - wird zur Überwachung und Beobachtung von AWS Diensten benutzt. In der Arbeit wurde es zur Überwachung von AWS Lambda-Funktionen verwendet, wobei die folgenden Metriken gesammelt wurden:
  - ConcurrentExecutions Average - durchschnittliche Anzahl der parallel ausgeführten Funktionen
  - ConcurrentExecutions Max - höchste Anzahl der parallel ausgeführten Funktionen
  - ConcurrentExecutions p95 - Percentile-95 der parallel ausgeführten Funktionen
  - Duration Average - durchschnittliche Ausführungszeit der Funktionen
  - Duration Sum - Summe der Ausführungszeit der Funktionen
  - Duration p95 - Percentile-95 der Ausführungszeit der Funktionen
  - Invocations Sum - Summe der Funktionsaufrufe

Es ist zu erkennen, dass die Anzahl der Anwendungsinstanzen und die Ressourcennutzung (CPU und Memory) von den auf Kubernetes laufenden Anwendungsinstanzen gemessen wurden. Diese Daten wurden in Schritten von 10 VUs gemessen, was Aufschluss über den Zusammenhang zwischen der Anzahl der Anwendungsinstanzen und der VUs bzw. der Requests geben kann. Vergleichbar mit den Daten wurden auch die AWS Lambda CloudWatch Metriken erfasst. Diese Daten und die Kubernetes-Daten stellen den Zusammenhang zwischen der Anzahl der ausgeführten Funktionen und der VUs bzw. der Requests dar.
Die genauen Einzelheiten zu den Tests und der Infrastruktur für die Tests für jeden Use Case werden in Kapitel 5 bzw. 6 ausführlich beschrieben.

5 Employee-Time-Sheet-Management-Portal

In diesem Kapitel werden die Implementierung und die Ergebnisse des Performance-Testings der in Kapitel 3.1 behandelten "Employee-Time-Sheet-Management-Portal Anwendung" vorgestellt. Im ersten Teil des Kapitels werden die Anforderungen der Anwendung erläutert. Die Anwendungslogik und die Aufteilung der Anwendung in Endpoints und Services bzw. Funktionen werden beschrieben. Des Weiteren werden der Architekturentwurf und die Architekturentscheidungen diskutiert, was durch UML-Diagramme veranschaulicht wird. Im weiteren Teil des Kapitels wird das architektonische Setup und das Setup der Tests beschrieben. In dem Teil wird der Einsatz der Services und die genauen Details der Tests behandelt. Am Ende des Kapitels werden die Ergebnisse der Tests vorgestellt und diskutiert. Außerdem werden in diesem Teil die Kosten der beiden Deploymentstrategien erläutert.

5.1 Anforderungen

Bei dem System handelt es sich um ein Portal für die Verwaltung von Stundenzetteln für Mitarbeiter*innen. Die Mitarbeiter*innen arbeiten an verschiedenen Projekten und haben Projekte, die als Lieblingsprojekte markiert sind. Diese Projekte werden angezeigt und der*die Mitarbeiter*in kann sie bearbeiten, aufrufen und löschen. Außerdem können die Mitarbeiter*innen die Zeit eintragen, die sie an einem Projekt gearbeitet haben. Sie können diese Zeit ändern, anzeigen, erstellen und löschen. Weitere Funktionalitäten der Anwendung sind die Rollen der Mitarbeiter*innen und die Anzeige der Feiertaginformationen, die für die Projekte wichtig sind. In dem in Kapitel 3.1 beschriebenen Papier besteht die Anwendung aus 12 API-Endpoints, die in 3 Module unterteilt sind:

- Favourite Projects - zuständig für die Lieblingsprojekte
- Timesheet - zuständig für die Verwaltung der Arbeitszeiten
- Miscellaneous - zuständig für die Rollen der Mitarbeiter*innen und die Feiertaginformationen

In dieser Arbeit werden Favourite Projects und Timesheet implementiert und untersucht. Das dritte Modul, Miscellaneous wird nicht implementiert, da in dieser Arbeit nur ein Prototyp des Papers erstellt wird und das Budget der Arbeit überschritten wäre. Dies

führt auch zu weniger API-Endpoints, die Endpoints, die in der Arbeit berücksichtigt und implementiert wurden, sind wie folgt:

- POST /projects/<id>
- GET /projects
- POST /timesheet
- PUT /timesheet/<id>
- DELETE /timesheet/<id>

Es ist zu erkennen, dass einige Endpoints aus den Modulen Favourite Projects und Timesheet nicht implementiert worden sind. Diese wurden aus denselben Gründen nicht implementiert wie das Modul Miscellaneous; die Implementierung in dieser Arbeit ist nur ein Prototyp des Systems und die Infrastrukturkosten sind zu hoch.

5.2 Setup und Entwurf

In diesem Kapitel werden das Setup der Anwendung und die Deploymentstrategien vorgestellt. Für das Deployment wurden zwei Deploymentstrategien verwendet: Serverless und Microservices. Bei der Microservices-Deploymentstrategie wurden, anders als im Paper, Docker und Kubernetes anstelle von AWS ECS eingesetzt. Für die Serverless-Anwendung wurden AWS Lambda und AWS API Gateway benutzt. Beide Deploymentstrategien wurden an eine AWS Aurora MySQL-Datenbankinstanz "db.t3.large" verbunden, die über 2 vCPU und 8 GiB RAM verfügt. Die Leistung der Datenbankinstanz und die Aufteilung in einen Reader- und einen Writer-Endpoint stellen sicher, dass die Datenbank keinen Bottleneck für das System darstellt.
Wie in Kapiteln 2.2.2 und 2.3.1 vorgestellt, benutzen beide Deploymentstrategien einen Skalierungsalgorithmus. Bei AWS Lambda wurde der Skalierungsalgorithmus bei der Standardkonfiguration belassen. Im Gegensatz zu AWS Lambda muss der Horizontal Pod Autoscaler von Kubernetes aktiviert und konfiguriert werden. Der Kubernetes-Algorithmus wurde so eingerichtet, dass die Anwendung von einer bis zu zehn Instanzen skalieren kann. Die Funktionen scaleDown und scaleUp skalieren die Anzahl der Instanzen nach unten oder nach oben, wenn die CPU-Auslastung über bzw. unter 50 % liegt. Um zu vermeiden, dass die Instanzen nicht ständig gelöscht und neu erstellt werden (flapping of metrics), wurde "stabilizationWindowSeconds" auf 60 Sekunden gesetzt. Außerdem

wurde ”periodSeconds” auf 30 Sekunden mit einem Wert von zwei eingesetzt. Dies gibt den Zeitraum in der Vergangenheit an, für den die Richtlinie gelten muss. Das bedeutet, dass nur zwei Instanzen innerhalb von 30 Sekunden herauf oder herunter skaliert werden können.
Den Kubernetes-Pods werden Ressourcen zugewiesen. Zu Beginn des Experiments wurde den Kubernetes-Pods 0,25 CPU-Core bzw. 250 millis und 488 Mi bzw. 512 MB RAM als requests zugewiesen, aber die CPU Ressourcen wurden auf 0,07 CPU-Core bzw. 70 millis und 138 Mi bzw. 148 MB RAM reduziert. Über die Reduzierung der Ressourcen wird mehr im Kapitel 5.3 und 5.4 gesprochen. Der Wert für die Limits wurde auf 488 Mi bzw. 512 MB RAM und 0,25 CPU-Core bzw. 250 Millis festgelegt. Bei AWS Lambda kann nur der RAM-Wert angegeben werden, da die CPU intern von AWS verwaltet wird. Daher wurden jeder Funktion im Experiment 512 MB RAM zugewiesen.
Der Source-Code der beiden Anwendungen sollte gleich gehalten werden, da die Unterschiede im Code, Auswirkungen auf den Test haben können. Damit wurde versucht, den Source-Code nur an den Stellen zu ändern, wo dies notwendig war. Ein Beispiel ist die Änderung des Codes beim Zugriff auf die MySQL Datenbank. In der Microservice-Anwendung wurde hierfür die MySQL-Library [Mys21] benutzt, da diese Library nicht für Serverless-Anwendungen geeignet ist, wurde die Erweiterung der Library, die Serverless MySQL Library [Dal21], verwendet.

5.2.1 Bausteinsicht

Die Bausteinsicht stellt die Komponenten des Systems dar. Die Komponenten des Serverless- und Microservices-Systems werden durch Komponentendiagramme dargestellt und beschrieben.

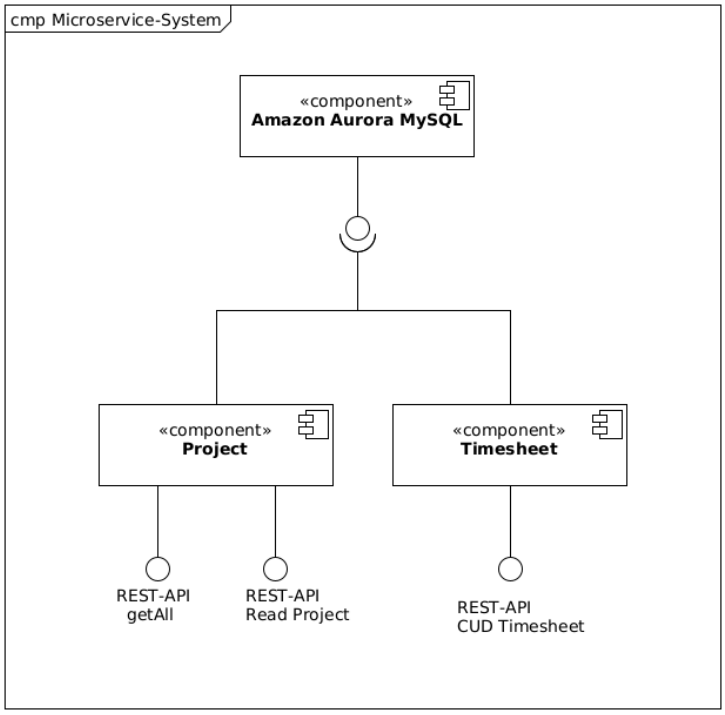

Abbildung 2: Komponentendiagramm Microservices-System, UML nach [OMG17]

Die Abbildung 2 zeigt das Komponentendiagramm des Microservices-Systems. Es ist zu erkennen, dass das System aus drei Komponenten besteht. Project und Timesheet sind zwei Microservices, die an die Amazon Aurora MySQL-Datenbank angebunden sind. Der Microservice Project besteht aus den Endpoints:

- POST /projects/<id>
- GET /projects

Gegenüber besteht das Microservice Timesheet aus den drei Endpoints:

- POST /timesheet

- PUT /timesheet/<id>
- DELETE /timesheet/<id>

Im Diagramm werden diese Endpoints durch eine Lollipop-Notation dargestellt, aber in der Implementierung sind diese Endpoints unabhängig voneinander.

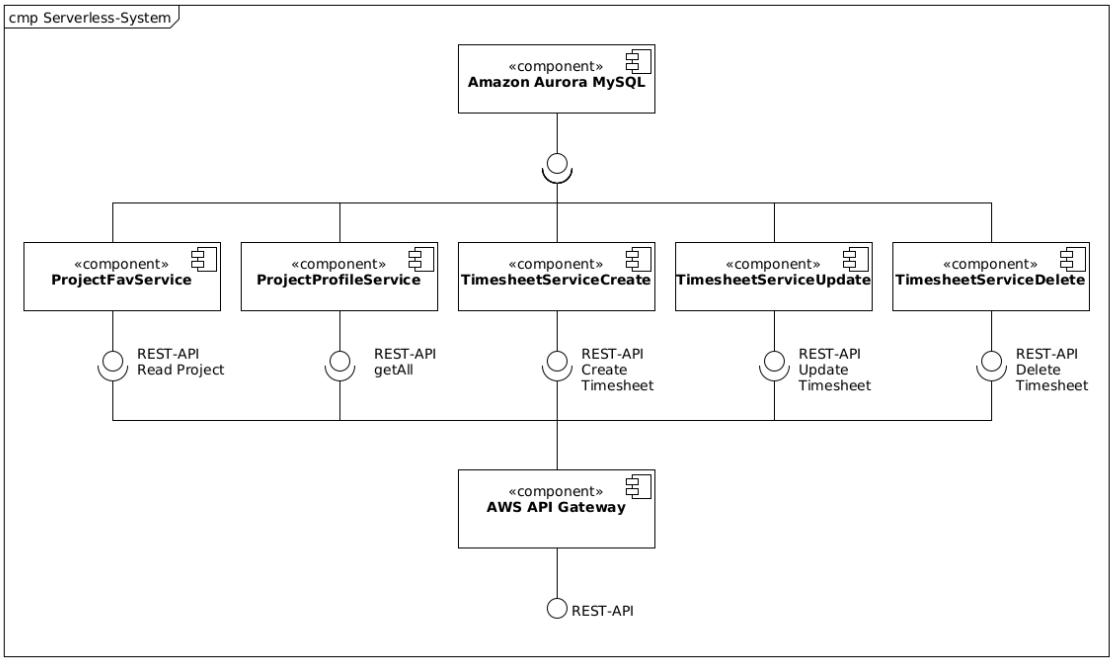

Abbildung 3: Komponentendiagramm Serverless-System, UML nach [OMG17]

Abbildung 3 zeigt das Komponentendiagramm des Serverless-Systems. Es ist zu erkennen, dass es sich von dem Microservices-System unterscheidet.
Das System besteht aus sieben Komponenten. Die erste Komponente ist das AWS API Gateway, das verwendet werden muss, wenn eine AWS Lambda Funktion über das Internet zugänglich sein soll. Das Gateway leitet die Requests an die fünf Lambda-Funktionen weiter: ProjectFavService, ProjectProfileService, TimesheetService Create, TimesheetService Update und TimesheetService Delete. Jede der fünf Funktionen hat einen Endpoint:

- ProjectFavService - POST /projects/<id>
- ProjectProfileService - GET /projects

- TimesheetService Create - POST /timesheet
- TimesheetService Update - PUT /timesheet/<id>
- TimesheetService Delete - DELETE /timesheet/<id>

Die Funktionen sind mit derselben Amazon Aurora MySQL-Datenbank verbunden, mit dem auch die Microservice-Anwendung verbunden ist.

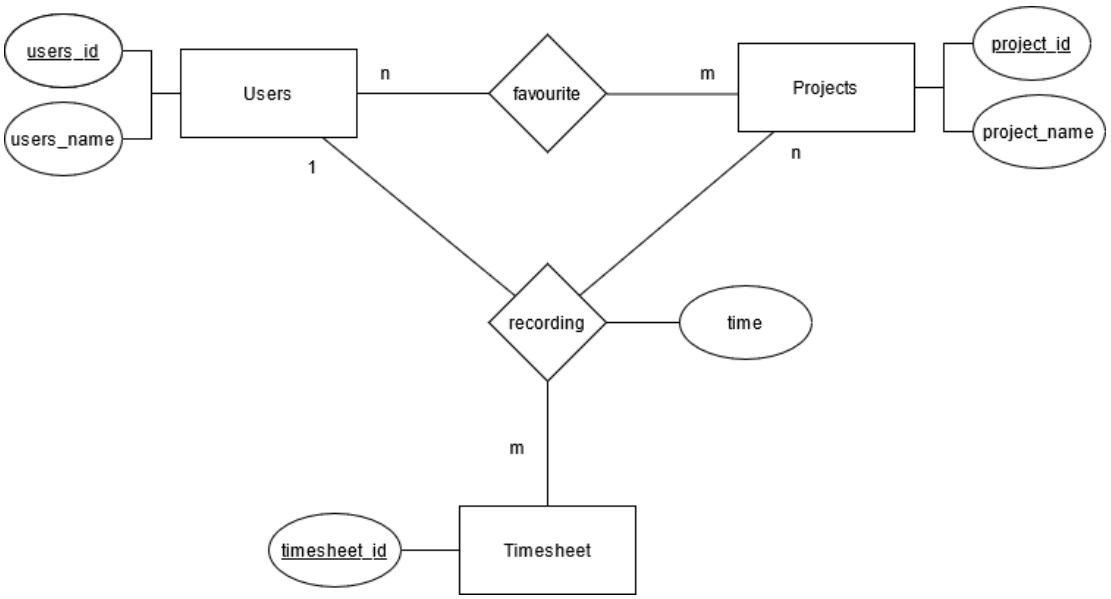

Abbildung 4: ER-Modell der Datenbank, UML nach [OMG17]

Die Abbildung 4 zeigt das ER-Modell der Datenbank. Es ist zu sehen, dass die Datenbank drei Entities hat: Users, Projects und Timesheet. Jeder User und jedes Projekt besitzen einen Namen und eine ID. Der User kann in der Beziehung ”favourite” mit einem Projekt stehen, was bedeutet, dass das Projekt eins der Lieblingsprojekte des Users ist. Eine zweite Beziehung ist die ”recording” Beziehung, die zwischen den Entities Timesheet, Projects und Users besteht. Die Beziehung ”recording” wird für die Erfassung der Stunden verwendet. Ein User hat für jedes Projekt ein Arbeitszeiteintrag. Der Arbeitszeiteintrag wird durch die Entity Timesheet und die Beziehung ”recording” dargestellt.

5.2.2 Laufzeitsicht

In der Laufzeitsicht werden Sequenzdiagramme verwendet, um die Interaktion zwischen den Komponenten darzustellen. Für jeden Endpoint wurde ein Sequenzdiagramm erstellt. Da sich die Diagramme ähneln, wird nur das getAllProjects-Diagramm für jede Deploymentstrategie beschrieben und verglichen; die übrigen Diagramme sind im Anhang zu finden. Die Diagramme beschreiben den Happy-Path der Anwendung und nicht die Validierung der Requests und andere Edge-Cases.

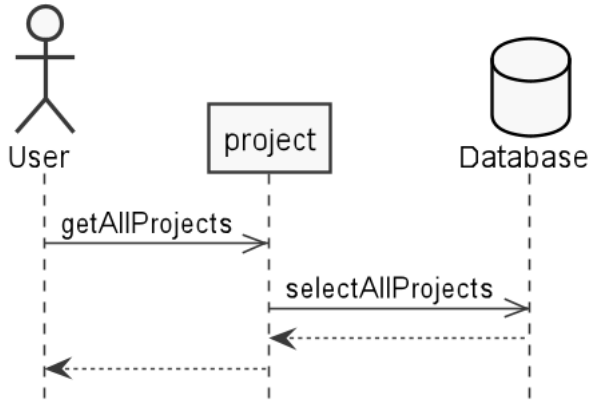

Abbildung 5: Sequenzdiagramm des getAllProjects Endpoints in der Microservice-Anwendung, UML nach [OMG17]

Abbildung 5 zeigt das Sequenzdiagramm des getAllProjects Endpoints in der Microservice-Anwendung. Aus dem Diagramm ist ersichtlich, dass der User die Schnittstelle zur Anwendung ”project” aufruft. Anschließend wird in der Anwendung der Request verarbeitet und ein SQL SELECT-Statement an den Reader der Datenbank gesendet. Die Datenbank liefert alle verfügbaren Projekte zurück, die Anwendung mappt die Objekte zu einem JSON ab und sendet sie an den User zurück.

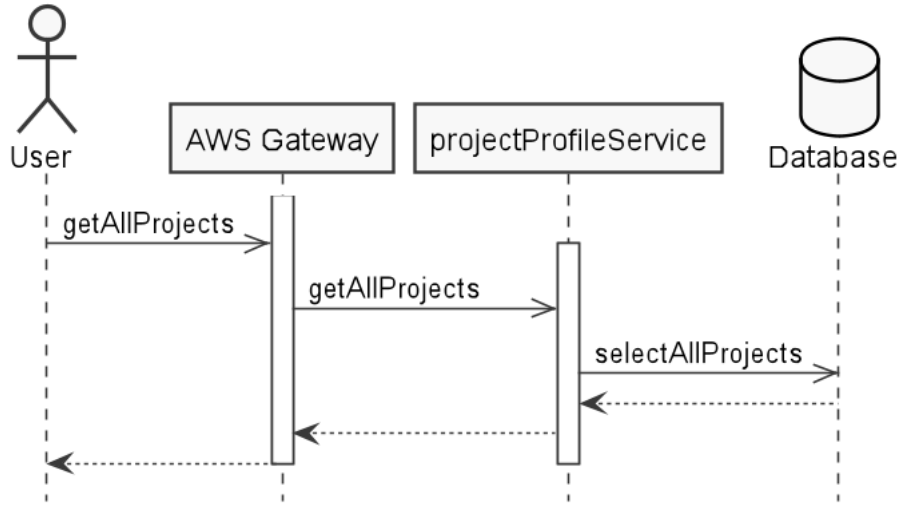

Abbildung 6: Sequenzdiagramm des getAllProjects Endpoints in der Serverless-Anwendung, UML nach [OMG17]

Die Abbildung 6 zeigt denselben Endpoint der Serverless-Anwendung. Es ist zu sehen, dass die Serverless-Deploymentstrategie, eine weitere Komponente besitzt. Es handelt sich um die Komponente ”AWS Gateway” welche im Kapitel ”Bausteinsicht” beschrieben wurde. Der User sendet den Request erst an das AWS Gateway, welches den Request an die Anwendung ”projectProfileService” weiterleitet. Rest des Ablaufs ist gleich zu dem Ablauf in der Microservices Abbildung.

5.2.3 Verteilungssicht

In der Verteilungssicht werden Deploymentdiagramme verwendet, um das Deployment der Anwendungen für die verschiedenen Deploymentstrategien zu beschreiben. Bei der Microservice-Anwendungen wird das Deployment mit Kubernetes und Docker beschrieben, bei der Serverless-Anwendung mit AWS Lambda und Serverless-Framework.

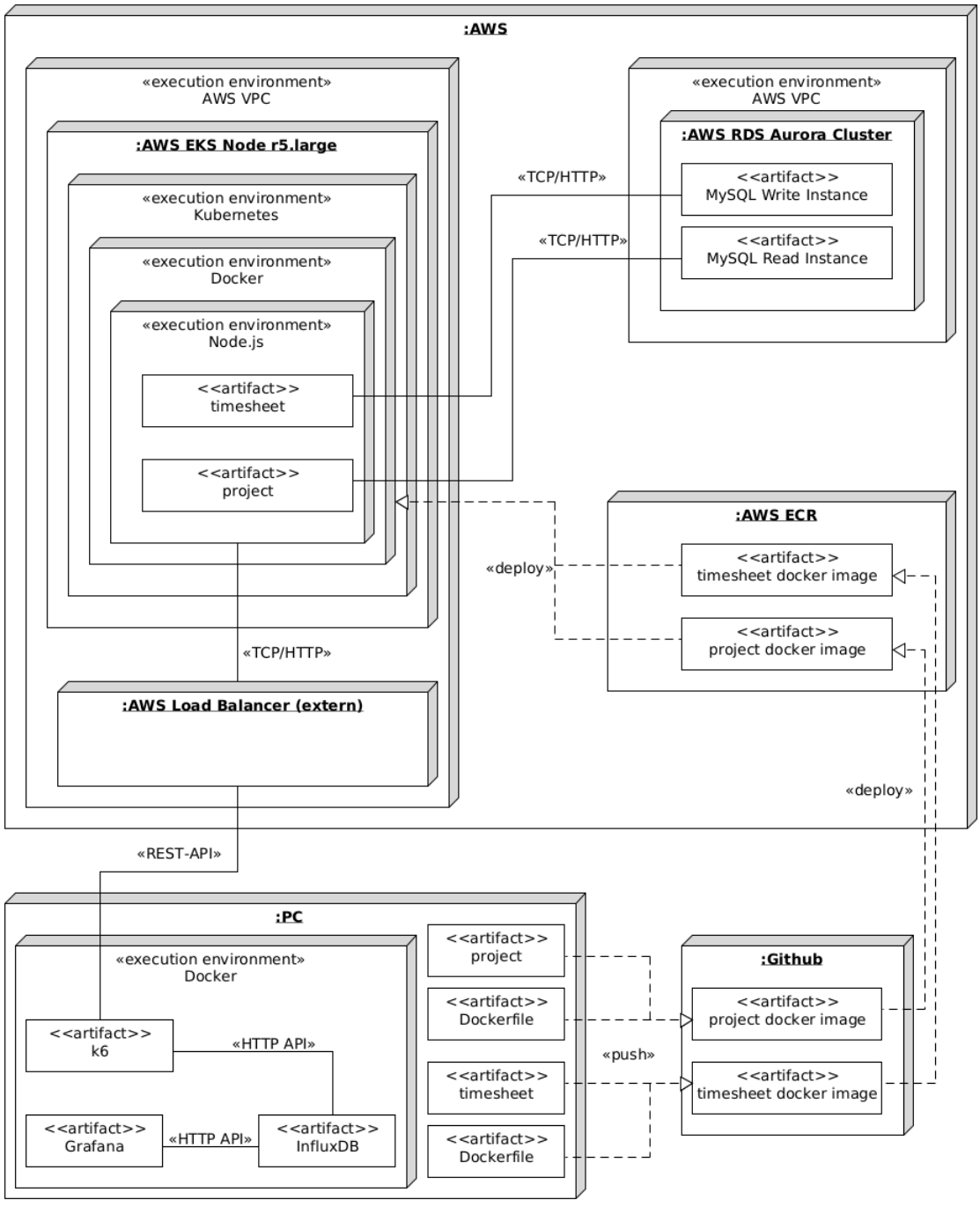

Abbildung 7: Deploymentdiagramm der Microservice-Anwendungen bei der Microservices-Deploymentstrategie mit Kubernetes, UML nach [OMG17]

Die Abbildung 7 zeigt das Deploymentdiagramm aller Miroservice-Anwendungen, die mit der Microservices-Deploymentstrategie deployt worden sind. Das bedeutet, dass die Anwendungen auf Kubernetes in einem Docker-Container laufen.
In der Abbildung 7 ist ein Node mit dem Namen ”PC” zu sehen, das ist der lokale Rechner, auf dem die Anwendung entwickelt wurde. Auf diesem Node sind zwei Anwendungen zu sehen: ’timesheet” und ”project”. Um die Anwendungen zu deployen, wurde Docker verwendet. Dafür wurden Dockerfiles geschrieben, welche die Anwendung in ein Docker-Image packagen. Ein weiterer Node ist ”Gitlab”’. Auf ”Gitlab” werden Source-Code der Anwendung und Dockerfile gepusht. Eine Gitlab CI-Pipeline, pusht die Docker-Images auf AWS ECR. Die Docker-Images müssen deployt werden, wie schon erwähnt, dafür wird Kubernetes verwendet. Für die Verwaltung von Kubernetes wurde AWS EKS eingesetzt. Für die Verwendung von AWS EKS muss mindestens ein Node vorhanden sein. Für die Arbeit wurden Nodes der Größe r6.large eingesetzt. Auf den Nodes werden die Anwendungen deployt. Zu diesem Zweck werden Kubernetes-Objekte wie Service, Deployment und Ingress erstellt. Diese Objekte werden durch YAML-Dateien definiert und manuell ausgeführt. Die Nodes laufen in einem AWS VPC und können aus dem Internet nicht erreicht werden. Das bedeutet, dass die Anwendungen keine Requests von außerhalb der Nodes erhalten können. Um dies zu ändern, wurde ein AWS Load-Balancer eingesetzt.
Dadurch können die Anwendungen aus dem Internet erreicht werden und der Load Balancer verteilt den Traffic auf verschiedene Kubernetes-Pods.
Eine weitere Komponente des Systems ist die Datenbank, auf die die Anwendungen zugreifen. Die Datenbank besteht aus einem Aurora Cluster, welches aus zwei Instances besteht, MySQL Write Instance und MySQL Read Instance. Der Cluster befindet sich in seiner eigenen AWS VPC. Die Konfiguration des AWS VPC der Datenbank wurde angepasst, so dass die Anwendungen aus dem Kubernetes Node die Datenbank aufrufen können.
Auf dem Node ”PC” befindet sich auch das Performance-Testing Setup, welches im Kapitel 4 ”Performance-Testing” beschrieben wurde, und ein Docker-Environment, der mit Docker-Compose erstellt wurde. In diesem Environment befinden sich drei Container: Grafana, InfluxDB und k6. K6 sendet die Requests an den Load-Balancer und speichert die Ergebnisse in die InfluxDB. Die Ergebnisse werden in dem Grafana Container erfasst und angezeigt.

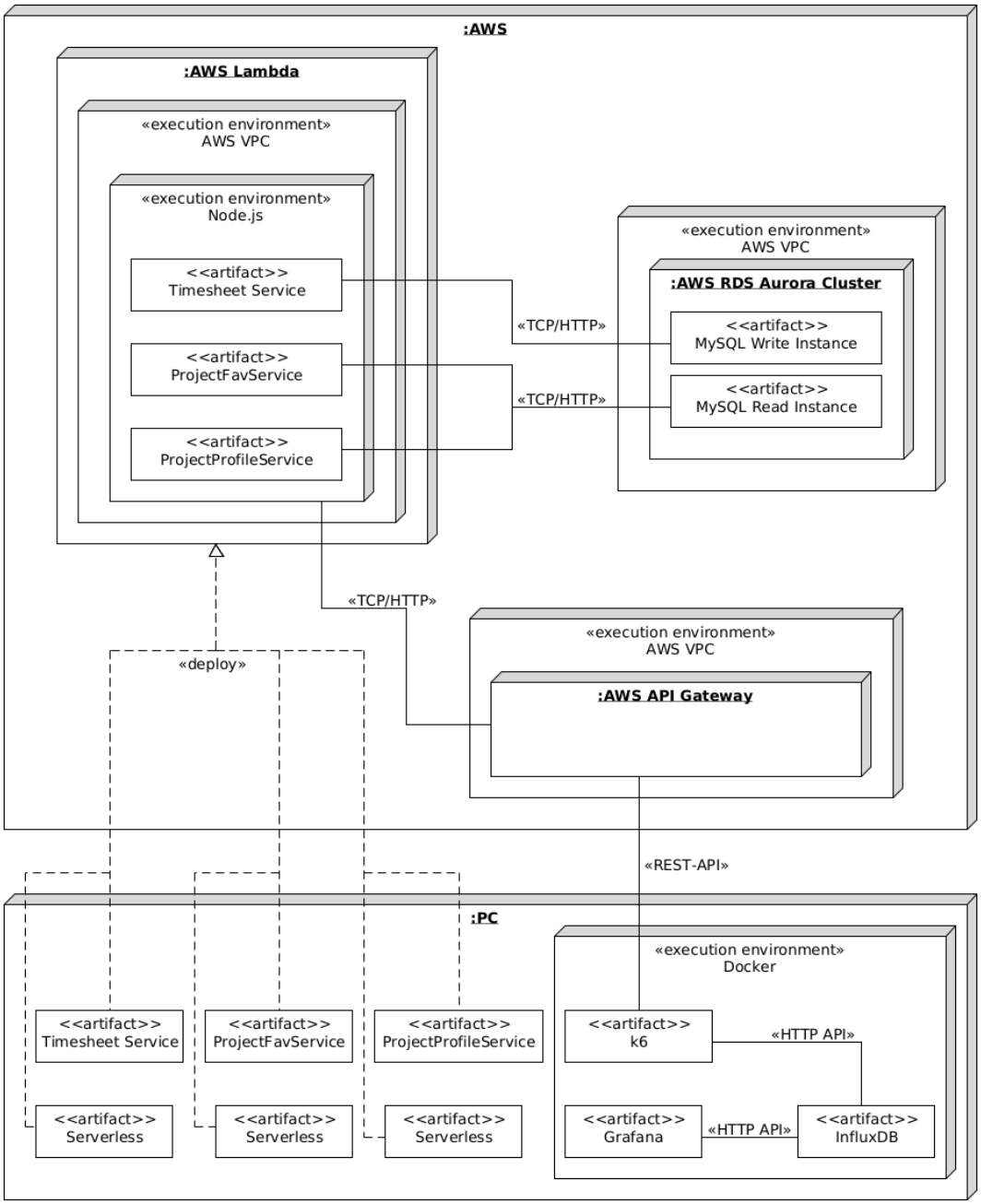

Abbildung 8: Deploymentdiagramm der Serverless-Anwendungen bei der Serverless -Deploymentstrategie mit AWS Lambda und Serverless-Framework, UML nach [OMG17]

Die Abbildung 8 zeigt das Deploymentdiagramm aller Serverless-Anwendungen, die mit der Serverless-Deploymentstrategie deployt worden sind. Dies bedeutet, dass die Anwendungen als AWS Lambda-Funktionen mit dem Serverless-Framework deployt wurden.
Es besteht ein großer Unterschied zu der Microservices-Deploymentstrategie. Es ist zu erkennen, dass die Anwendungen direkt von dem lokalen Rechner auf AWS deployt werden. Dies ist mithilfe des Serverless-Frameworks möglich. Eine serverless.yml-Datei wird verwendet, um das Deployment zu definieren. Die serverless.yml-Datei wird aus einem Terminal ausgeführt und damit wird der Source-Code als AWS Lambda-Funktion auf AWS deployt. Die Anwendungen laufen auf AWS Lambda als unabhängige Funktionen in einem AWS VPC und einer AWS Region. Für jeden definierten Endpoint wird ein Endpoint im AWS API Gateway erstellt, welches die Requests aus dem Internet an die AWS Lambda Funktionen weiterleitet. Genau wie bei der Microservices-Deploymentstrategie gibt es einen Aurora-Cluster, der aus zwei MySQL Instancen besteht. Dabei handelt es sich um denselben Cluster wie bei der Microservices-Deploymentstrategie. Die Konfiguration des AWS VPC des Aurora-Clusters musste nicht angepasst werden, da dies bereits für die Microservices-Deploymentstrategie gemacht wurde.

5.3 Methodik

Im Kapitel 4 ”Performance-Testing” wurde das Performance-Testings und die in dieser Arbeit verwendeten Ansätze erläutert.
In diesem Kapitel wird die Vorgehensweise bei der Durchführung der Tests und der Sammlung der Daten für das ”Employee-Time-Sheet-Management-Portal” genauer beschrieben.
Wie im Kapitel 4 erwähnt, wurden 40 Tests für den Use Case ”Employee-Time-Sheet-Management-Portal” durchgeführt. Die Anzahl der Tests wird wie folgt berechnet:
$2\ Durchlaeufe\ *\ ((5\ Endpoints\ *\ 2\ Deploymentstrategien(Serverless\ und\ Microservices)\ *2\ Testarten))$
Zusätzlich zu den 40 Tests wurden zu Beginn 22 weitere Tests durchgeführt. Diese Tests waren nicht erfolgreich, dennoch sind die Ergebnisse der Tests wichtig, weil wir durch die Tests Rückschlüsse auf AWS Lambda und Kubernetes ziehen können.
Zwei der 22 Tests waren erfolglos, weil der Microservice-Anwendung zu viele Ressourcen zugewiesen wurden. Die CPU- und Memory-Werte stammen aus dem Paper ”Microservices vs Serverless: A Performance Comparison on a Cloud-native Web Application” [FJG20], welches im Kapitel 3.1 vorgestellt worden ist. Den anderen 20 Tests wurden zu

wenige Ressourcen zugewiesen.
Die Tests werden in Kapitel 5.4 und 5.5 näher vorgestellt und diskutiert.
Bei den anderen 40 Tests wurde zuerst der ”GET /projects” Endpoint getestet. Zuerst wurde die Microservice-Anwendung und danach die Serverless-Anwendung getestet, beide Anwendungen wurden mit dem incremental Test getestet. Jeder incremental Test dauert 450 Sekunden und die Anzahl der VUs steigt bis zu 158. Nach jeden Test wurde eine Pause eingelegt, um zu gewährleisten, dass das System kalt ist.
Das Gleiche wurde für die Endpoints ”PUT /timesheet/<id>”, ”PUT /timesheet/<id>” und ”POST /timesheet” gemacht. Nach dem Testen der Endpoints ”PUT /timesheet/<id>” und ”POST /timesheet” wurden die Einträge in der Datenbank gelöscht und die initialen Daten hinzugefügt.
Beim letzten Endpoint ”DELETE /timesheet/<id>” wurden erst 160.000 neue Einträge in die Datenbank hinzugefügt. Die IDs der Einträge beginnen bei 1.000 und enden bei 160.000. So konnte im k6-Script sichergestellt werden, dass der VU im Request eine ID sendet, die noch nicht gelöscht wurde, insbesondere wenn die VUs die Requests parallel senden.
Das Gleiche wurde für die triangle Tests gemacht, die triangle Tests dauern ebenfalls 450 Sekunden, aber die Anzahl der VUs wird bis zu 160 inkrementiert, dieser Vorgang dauert 225 Sekunden. Für die restlichen 225 Sekunden wird die Anzahl der VUs dekrementiert. Am Ende des Tests steht die Anzahl der VUs bei eins. Das beschriebene Verfahren umfasst die ersten 20 Tests, die wir als ersten Durchlauf bezeichnen. Ein weiterer Durchlauf wurde mit demselben Verfahren durchgeführt, den wir als zweiten Durchlauf bezeichnen. Es ist auch erwähnenswert, dass nach jedem Test die Daten gesammelt wurden. Die gesammelten Daten werden am Ende von Kapitel 4 ”Performance-Testing” beschrieben.

5.4 Ergebnisse

Nach der Durchführung der Tests wurden alle Daten gesammelt und mithilfe von Python-Scripts gefiltert. Die gefilterten Daten wurden in csv.-Dateien gespeichert, und mit dem Tool Datawrapper wurden die Diagramme erstellt.
Diese Daten bzw. Diagramme werden in diesem Kapitel vorgestellt. Zunächst werden die 22 Tests vorgestellt, die zuerst durchgeführt wurden. Diese Tests bestehen nur aus einem Durchlauf, da die Tests nicht ”valide” sind, aber in dieser Arbeit werden sie trotzdem vorgestellt, um die Konzepte von AWS Lambda und Kubernetes besser zu verstehen und um zu zeigen, welche Schwierigkeiten während der Arbeit aufgetreten sind.

Darüber hinaus werden die restlichen Tests vorgestellt, die im nächsten Kapitel ausführlich beschrieben und diskutiert werden.
Für jeden Test wurden zwei Diagramme erstellt. Das erste Diagramm zeigt die Request-Duration als Percentile-95-Wert über die Zeit X. Ein Beispiel für ein solches Diagramm ist in Abbildung 9 dargestellt. Die Percentile-Werte wurden über k6-Script erfasst und in die InfluxDB gespeichert. Die gespeicherten Daten wurden auf Grafana angezeigt und als csv.-Datei exportiert.
Wie bereits erwähnt, befindet sich auf der x-Achse des Diagramms die Zeit in Sekunden und auf der y-Achse die Request-Duration als Percentile-95-Wert, die in der Zeiteinheit ms dargestellt wird. Im Diagramm sind der Microservice-Durchlauf und der AWS Lambda- bzw. Serverless-Durchlauf dargestellt. In einigen Diagrammen sind auch zwei Durchläufe dargestellt. Ein weiterer Wert sind die VUs, die in Kapitel 4 "Performance-Testing" vorgestellt wurden.
Das zweite Diagramm zeigt die Skalierbarkeit der AWS Lambda Funktionen und Kubernetes-Pods. Ein Beispiel für ein solches Diagramm ist in Abbildung 10 dargestellt. Auf der x-Achse ist die Zeit in Sekunden angegeben und auf der y-Achse die Anzahl der AWS Lambda Funktionen und Kubernetes-Pods, die zum Zeitpunkt x ausgeführt werden. Die Anzahl der Kubernetes-Pods wurde mit einem Shell-Script über die Befehle des Command-Line-Tools kubectl gesammelt. Die Anzahl der AWS Lambda Funktionen wurde nach dem Testen über CloudWatch erfasst.
Genau wie in den anderen Diagrammen kann es einen oder zwei Durchläufe geben, das hängt vom Test ab. Für andere Werte, die nicht in den Diagrammen dargestellt werden können, wurden Tabellen erstellt. Die folgenden Werte sind in den Tabellen zu finden:

- Minimum (Min)
- Maximum (Max)
- Average (Avg)
- Percentile-95 (p(95))
- average number of requests served per second (Avg. RPS)
- number of sucessfulle request (suc req)
- number of unsucessfulle requests (unsuc req)

Für die Erläuterung der Metriken siehe Kapitel 4 ”Performance-Testing”.
In diesem Kapitel werden auch die Kosten für die beiden Deploymentstrategien dargestellt. Die monatlichen und jährlichen Kosten für AWS werden aufgelistet und in Kapitel 5.5 erörtert.

5.4.1 Erhöhte CPU- und Memory-Werte

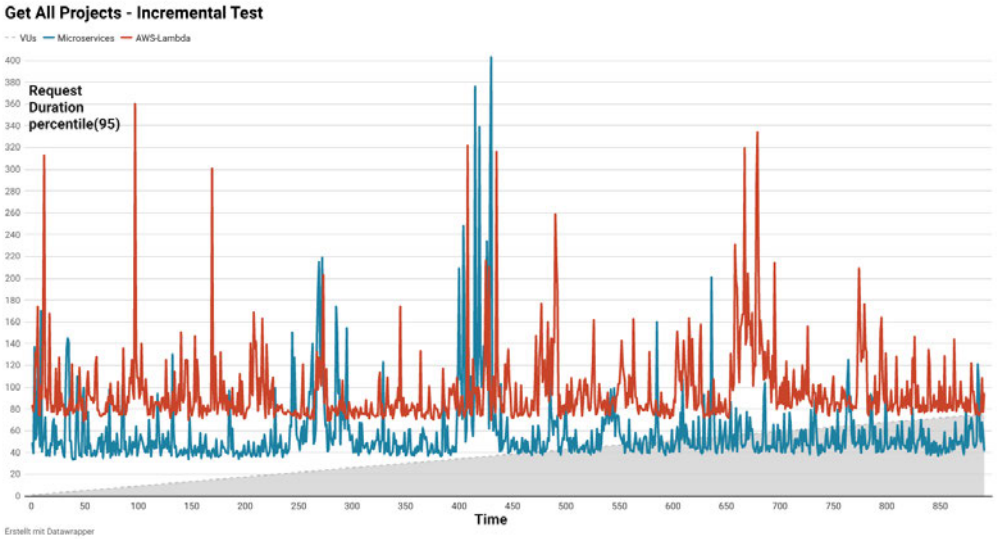

Abbildung 9: Microservices und AWS Lambda - Request-Duration als Percentile-95 über die Zeit x des ”GET /projects” Endpoints

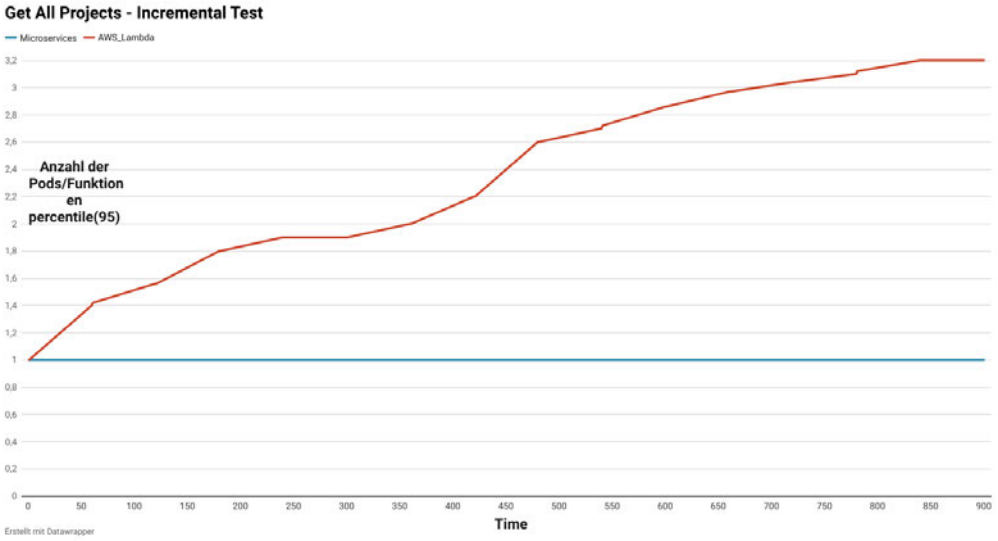

Abbildung 10: Microservices und AWS Lambda - Skalierbarkeit über die Zeit x des ”GET /projects” Endpoints

Incremental Tests	Min	Avg	Max	p(95)	Avg. RPS	suc req	unsuc req
Microservices	30.21	41.88	405.81	67.95	37.85	34100	0
AWS Lambda	29.95	75.6	1230	125.2	36.66	33034	0

Tabelle 1: Microservices und AWS Lambda - Ergebnisse der incremental Tests des ”GET /projects” Endpoints

Abbildung 9, 10 und die Tabelle 1 zeigen die Ergebnisse der ersten beiden Tests, die durchgeführt wurden. Beide Tests sind incremental Tests, die auf dem Endpoint ”GET /projects” ausgeführt wurden. Die CPU- und Memory-Werte der Kubernetes-Pods und AWS Lambda-Funktionen stammen aus dem Paper ”Microservices vs Serverless: A Performance Comparison on a Cloud-native Web Application” [FJG20], das in Kapitel 3.1 vorgestellt wurde. Der Kubernetes-CPU-Wert beträgt 0,25 CPU-Core bzw. 250 Milis und der Memory-Wert beträgt 512 MB. Für die AWS Lambda Funktionen beträgt der Memory-Wert ebenfalls 512 MB. Ein weiterer Wert, der dem Paper entnommen wurde, ist die Testlaufzeit von 900 Sekunden, die in anderen Tests auf 450 Sekunden reduziert wurde.
Es wurde schnell festgestellt, dass den Anwendungen zu viele Ressourcen zugewiesen wurden, sodass beschlossen wurde, die Tests mit reduzierten Werten erneut durchzuführen. Was genau zu viele Ressourcen bedeutet und welche Auswirkung das hat, wird im Kapitel 5.5 ”Diskussion” genauer beschrieben. Aus diesem Grund wurde nur ein Durchlauf und nur ein Endpoint (”GET /projects”) mit diesen CPU- und Memory-Werten getestet. Die Ergebnisse werden dennoch in der Arbeit vorgestellt, da sich anhand der Tests Rückschlüsse auf AWS Lambda und Kubernetes ziehen lassen. Diese Rückschlüsse werden in Kapitel 5.5 ”Diskussion” näher erläutert.

5.4.2 Geringe CPU- und Memory-Werte

Da festgestellt wurde, dass die Werte in vorherigen Tests zu hoch waren, wurden sie angepasst. Die Kubernetes CPU- und Memory-Werte wurden auf 145 MB und 0,07 CPU-Core bzw. 70 Milis reduziert. Außerdem wurde der Memory-Wert der AWS Lambda Funktionen auf 145 MB reduziert. Die Ausführungszeit wurde auch von 900 Sekunden auf 450 Sekunden reduziert, um den "requests per second"-Wert zu erhöhen und um Kosten der Arbeit zu reduzieren. Nach der Durchführung der Tests wurde festgestellt, dass die Anwendungen zu wenig Ressourcen zur Verfügung hatten. Die genaue Bedeutung des Begriffs "zu wenig Ressourcen" und die Auswirkungen werden in Kapitel 5.5 "Diskussion" näher beschrieben.
Da nur eine Teilmenge der Ergebnisse für die Bewertung notwendig ist, wird nur eine Teilmenge der Ergebnisse in diesem Kapitel erscheinen, die restlichen Ergebnisse sind im Anhang zu finden. Die folgenden Ergebnisse sind in diesem Kapitel zu sehen:

1. Get All Projects - Incremental Test
2. Get All Projects - Triangle Test
3. Update Timesheet - Incremental Test
4. Update Timesheet - Triangle Test

Incremental Tests		Min	Avg	Max	p(95)	Avg. RPS	suc req	unsuc req
Get All Projects	Microservices	0.41	27.77	495.9	75.05	76.75	34619	1
	AWS Lambda	27.02	41.89	2640	60.27	75.69	34138	0
Update Timesheet	Microservices	13.62	39.37	692.47	104.25	75.86	34213	4
	AWS Lambda	29.19	43.62	2000	60.22	75.55	34084	0

Tabelle 2: Microservices und AWS Lambda - Ergebnisse der incremental Tests

Triangle Tests		Min	Avg	Max	p(95)	Avg. RPS	suc req	unsuc req
Get All Projects	Microservices	14.01	31.03	485.47	79.13	77.80	35035	2
	AWS Lambda	27.99	61.21	2610	92.1	75.55	34044	0
Update Timesheet	Microservices	13.91	60.6	952.32	206.8	75.66	34043	19
	AWS Lambda	9.13	70.72	2400	160.23	74.85	33702	12

Tabelle 3: Microservices und AWS Lambda - Ergebnisse der triangle Tests

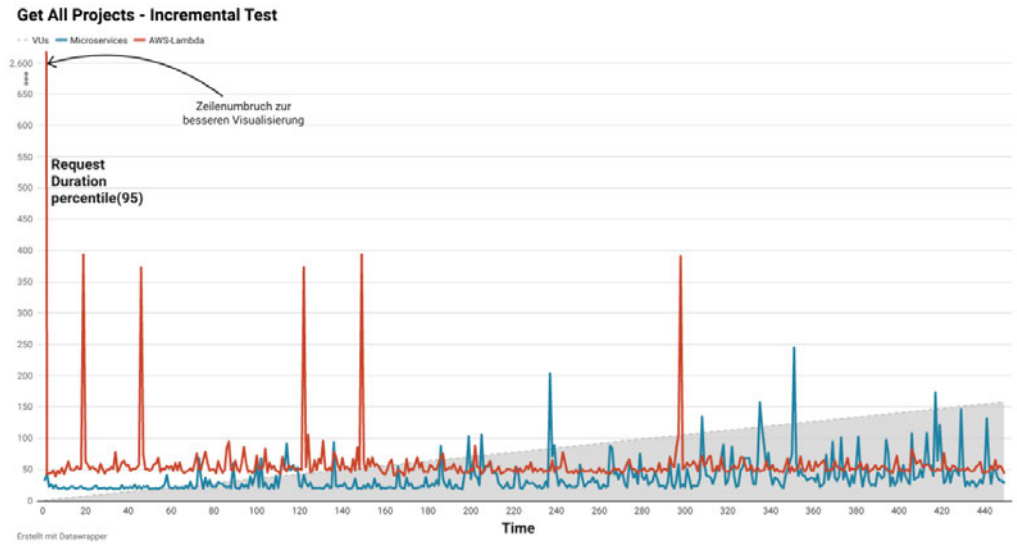

Abbildung 11: Microservices und AWS Lambda - Request-Duration als Percentile-95 über die Zeit x des ”GET /projects” Endpoints

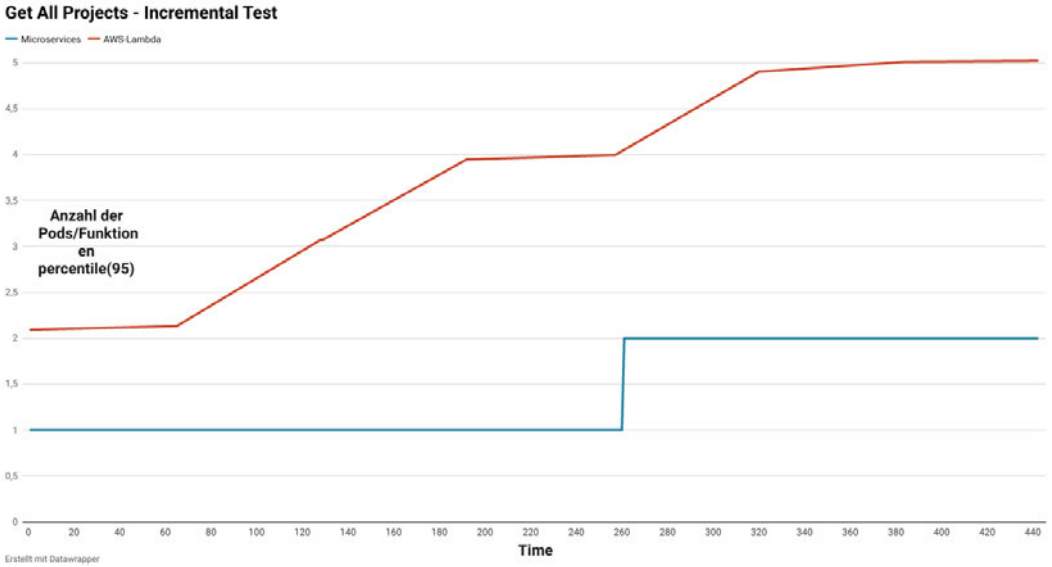

Abbildung 12: Microservices und AWS Lambda - Skalierbarkeit über die Zeit x des ”GET /projects” Endpoints

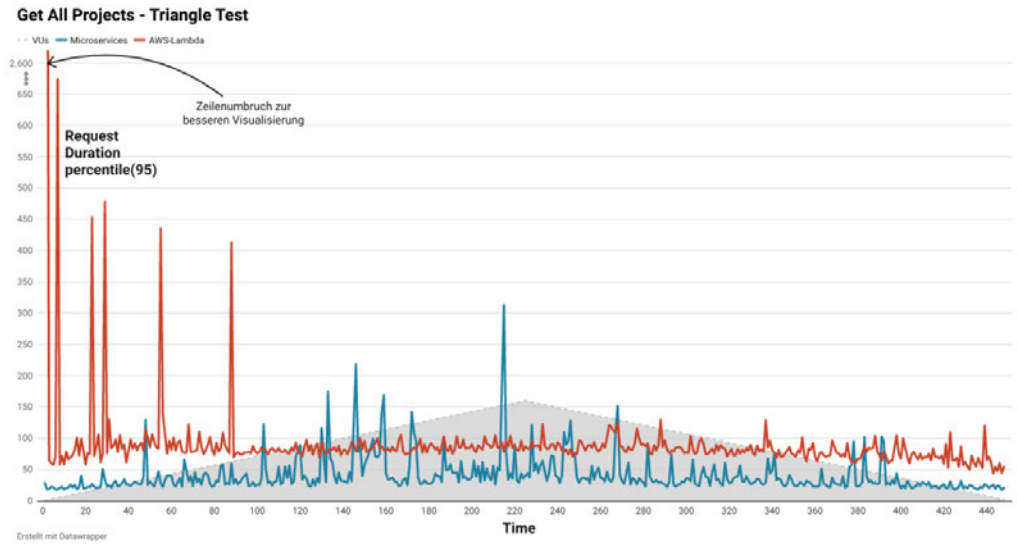

Abbildung 13: Microservices und AWS Lambda - Request-Duration als Percentile-95 über die Zeit x des ”GET /projects” Endpoints

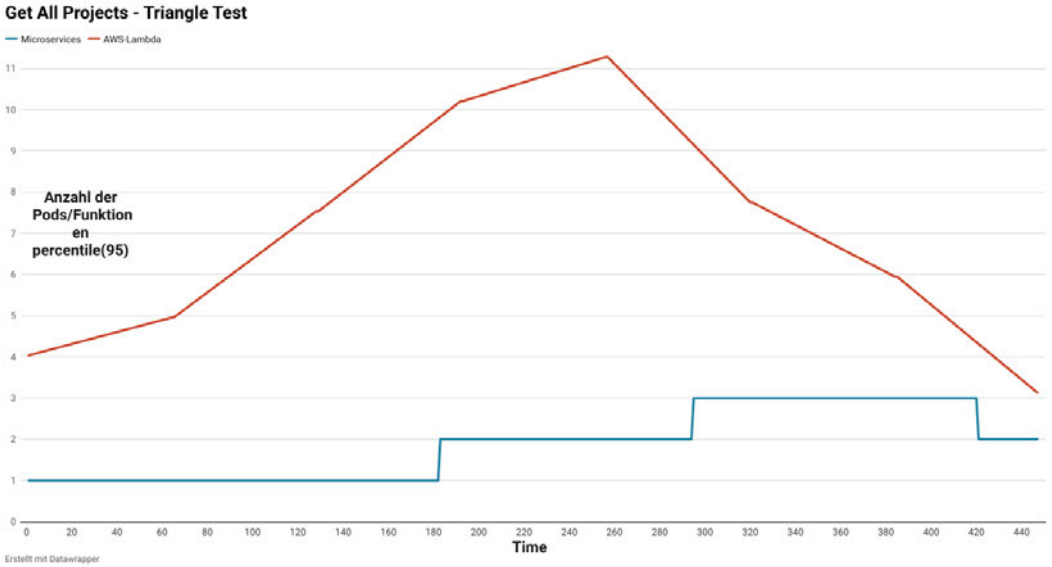

Abbildung 14: Microservices und AWS Lambda - Skalierbarkeit über die Zeit x des ”GET /projects” Endpoints

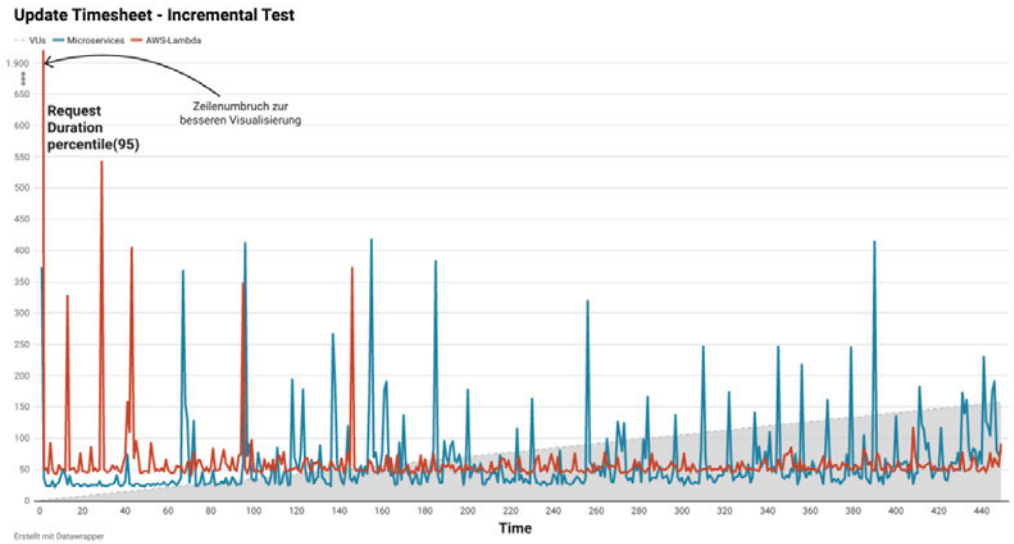

Abbildung 15: Microservices und AWS Lambda - Request-Duration als Percentile-95 über die Zeit x des ”PUT /timesheet/<id>” Endpoints

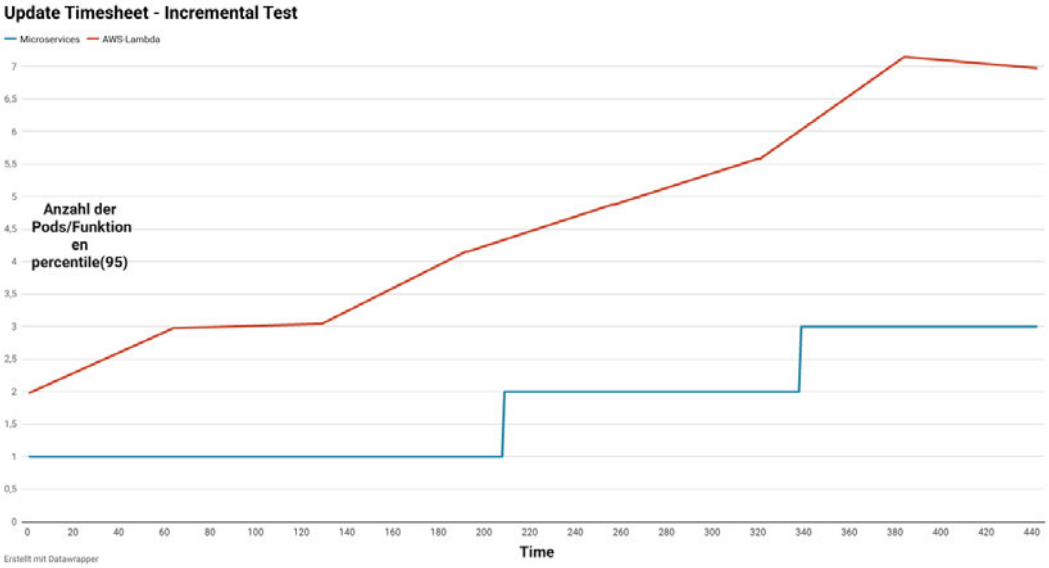

Abbildung 16: Microservices und AWS Lambda - Skalierbarkeit über die Zeit x des ”PUT /timesheet/<id>” Endpoints

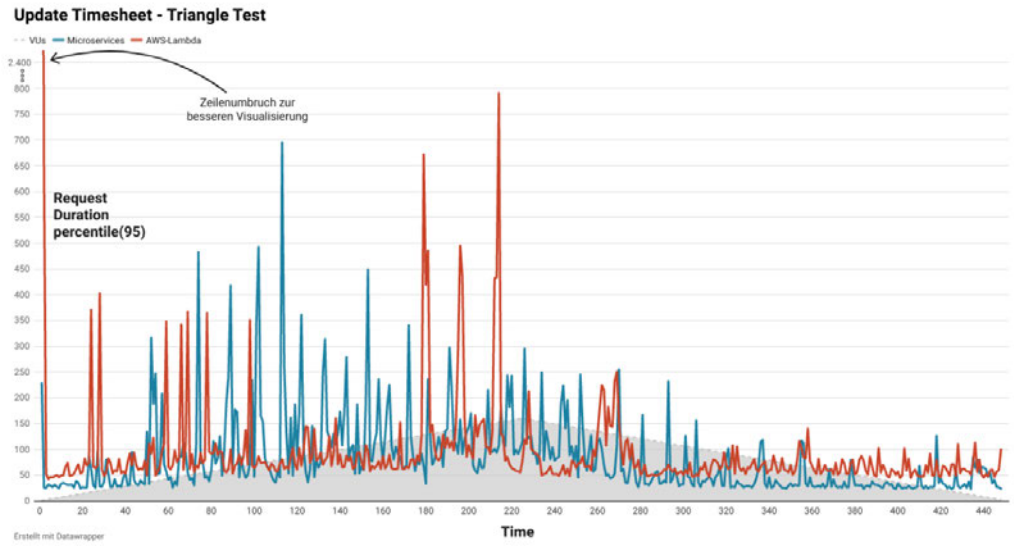

Abbildung 17: Microservices und AWS Lambda - Request-Duration als Percentile-95 über die Zeit x des ”PUT /timesheet/<id>” Endpoints

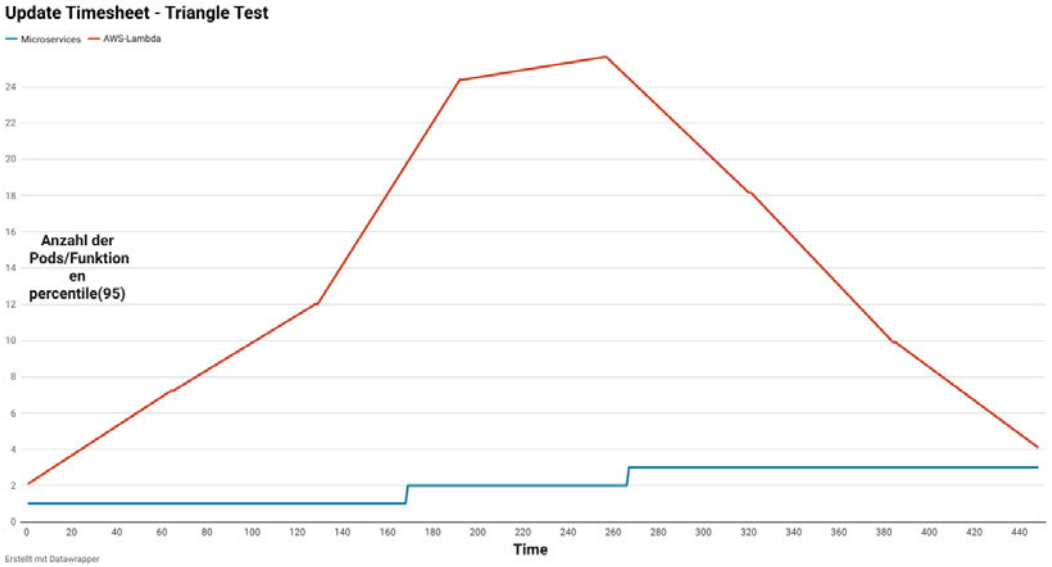

Abbildung 18: Microservices und AWS Lambda - Skalierbarkeit über die Zeit x des ”PUT /timesheet/<id>” Endpoints

5.4.3 Angepasste CPU- und Memory-Werte

In diesem Kapitel werden die Ergebnisse der restlichen 40 Tests vorgestellt. Bei diesen Tests wurden die Ressourcen-Werte angepasst und diese Tests werden als erfolgreiche Tests betrachtet. Die Kubernetes CPU- und Memory-Werte wurden bei dem ”requests” Parameter bei 145 MB und 0,07 CPU-Core bzw. 70 Milis belassen. Der ”limits” Parameter wurde auf 512 MB und 0,25 CPU-Core bzw. 250 Milis erhöht. In gleicher Weise wurde für die AWS Lambda Funktionen der Memory-Wert auf 512 MB erhöht.
Es ist auch wichtig zu erwähnen, dass im Gegensatz zu den anderen Tests die folgenden Tests aus zwei Durchläufen bestehen. In diesem Kapitel werden nur die Abbildungen des ”GET /projects” Endpoints gezeigt, da nur diese für die Diskussion wichtig waren, die übrigen Abbildungen finden sich im Anhang.

Incremental Tests		Min	Avg	Max	p(95)	Avg. RPS	suc req	unsuc req
Get All Projects	Microservices Durchlauf 1	27.68	38.29	220.32	54.17	75.9	34235	0
	AWS Lambda Durchlauf 1	37.66	58.22	1440	88.09	74.5	33605	0
	Microservices Durchlauf 2	29.39	36.77	200.26	48.75	76.03	34295	0
	AWS Lambda Durchlauf 2	40.71	54.71	1420	70.68	74.72	33701	0
Read Favourite Projects	Microservices Durchlauf 1	29.29	37.26	307.08	51.85	75.98	34270	0
	AWS Lambda Durchlauf 1	38.44	53.05	1150	74.74	74.85	33774	0
	Microservices Durchlauf 2	28.83	35.18	205.8	45.12	76.13	34336	0
	AWS Lambda Durchlauf 2	37.41	50.28	1060	63.07	75.09	33872	0
Create Timesheet	Microservices Durchlauf 1	31.17	43.32	164.01	63.85	75.57	34085	0
	AWS Lambda Durchlauf 1	42.86	61.25	1340	89.05	74.23	33482	0
	Microservices Durchlauf 2	31.04	40.2	323.01	52.55	75.79	34185	0
	AWS Lambda Durchlauf 2	41.27	54.59	1100	69.05	74.77	33723	0
Delete Timesheet	Microservices Durchlauf 1	27.84	44.47	371.28	66.86	75.47	34054	0
	AWS Lambda Durchlauf 1	39.85	57.71	1100	78.07	74.54	33625	0
	Microservices Durchlauf 2	31.82	41.09	237.83	54.8	75.74	34169	0
	AWS Lambda Durchlauf 2	42.11	57.19	1130	75.02	75.32	34162	4
Update Timesheet	Microservices Durchlauf 1	32.74	45.17	270.83	66.22	75.42	34016	0
	AWS Lambda Durchlauf 1	41.6	60.36	1220	83.72	74.24	33477	6
	Microservices Durchlauf 2	32.29	42.41	181.98	58.17	75.63	34110	0
	AWS Lambda Durchlauf 2	41.72	58.28	1210	78.58	74.47	33591	1

Tabelle 4: Microservices und AWS Lambda - Ergebnisse der incremental Tests

Triangle Tests		Min	Avg	Max	p(95)	Avg. RPS	suc req	unsuc req
Get All Projects	Microservices Durchlauf 1	27.28	37.95	188.26	54.48	77.27	34793	0
	AWS Lambda Durchlauf 1	39.78	56.82	3170	78.53	75.82	34172	0
	Microservices Durchlauf 2	29.36	35.91	307.99	48.74	77.50	34886	0
	AWS Lambda Durchlauf 2	40.82	53.76	1450	68.67	76.02	34285	0
Read Favourite Projects	Microservices Durchlauf 1	28.8	36.27	129	50.63	77.29	34848	0
	AWS Lambda Durchlauf 1	35.26	52.36	1100	71.6	76.16	34315	0
	Microservices Durchlauf 2	29.34	34.95	308.08	45.94	77.57	34921	0
	AWS Lambda Durchlauf 2	39.01	52.42	936.77	69.17	76.15	34323	0
Create Timesheet	Microservices Durchlauf 1	31.19	44.67	299.95	68.11	76.0	34562	0
	AWS Lambda Durchlauf 1	40.85	60.6	1950	85.47	75.51	34025	0
	Microservices Durchlauf 2	32.33	44.65	340.02	66.9	76.75	34568	0
	AWS Lambda Durchlauf 2	42.22	60.14	1620	83.98	75.54	34046	0
Delete Timesheet	Microservices Durchlauf 1	19.78	41.2	223.73	56.24	77.03	34698	2
	AWS Lambda Durchlauf 1	42.23	56.52	1590	73.83	75.91	34183	2
	Microservices Durchlauf 2	31.84	42.33	189.21	60.32	76.88	34664	2
	AWS Lambda Durchlauf 2	42.05	57.1	914.84	74.64	75.85	34160	4
Update Timesheet	Microservices Durchlauf 1	30.44	43.52	270.73	62.7	76.85	34612	1
	AWS Lambda Durchlauf 1	43.49	61.26	1200	84.74	75.54	34014	0
	Microservices Durchlauf 2	31.43	42.06	303.23	59.11	76.86	34663	1
	AWS Lambda Durchlauf 2	40.72	58.7	1360	81.12	75.66	34092	0

Tabelle 5: Microservices und AWS Lambda - Ergebnisse der triangle Tests

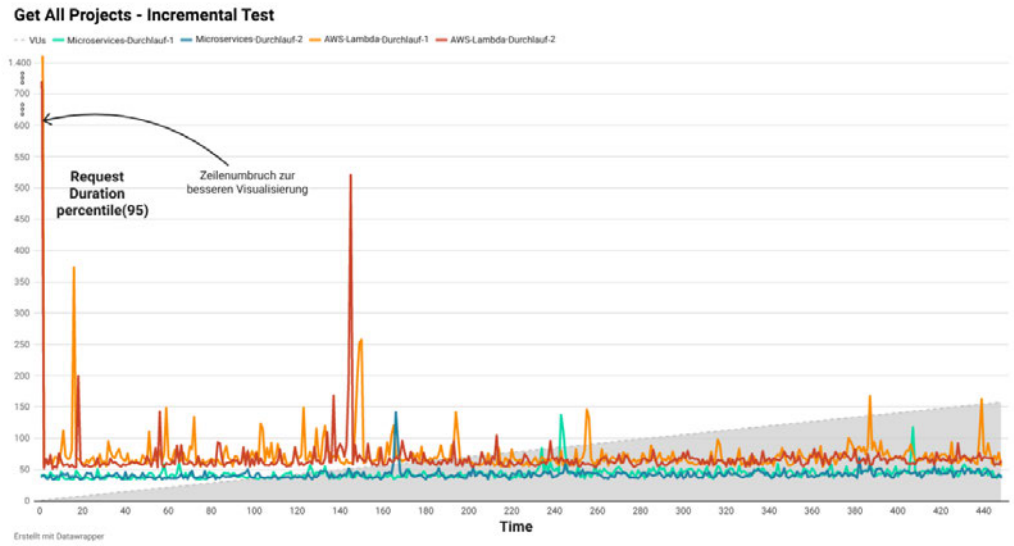

Abbildung 19: Microservices und AWS Lambda - Request-Duration als Percentile-95 über die Zeit x des ”GET /projects” Endpoints

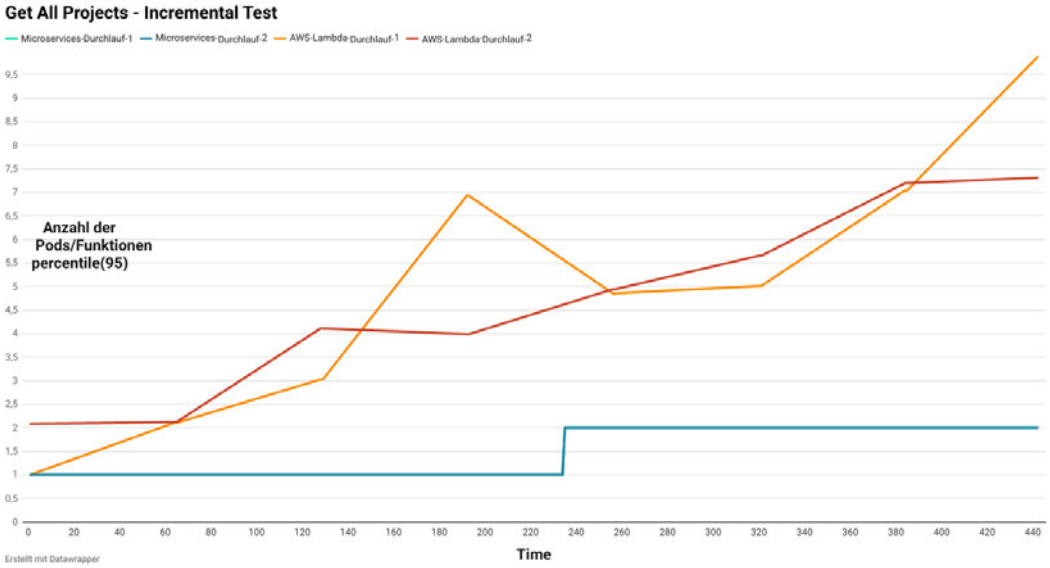

Abbildung 20: Microservices und AWS Lambda - Skalierbarkeit über die Zeit x des ”GET /projects” Endpoints

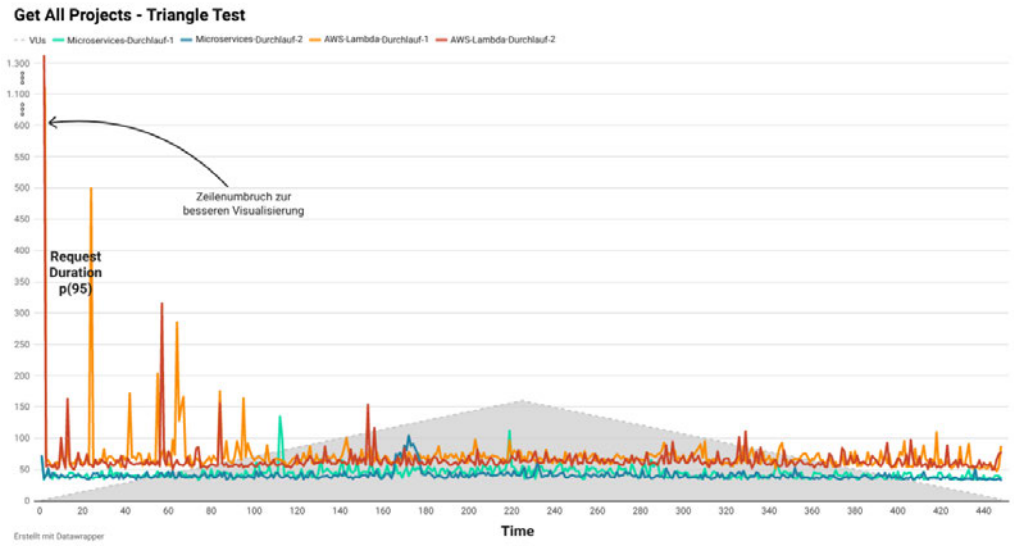

Abbildung 21: Microservices und AWS Lambda - Request-Duration als Percentile-95 über die Zeit x des ”GET /projects” Endpoints

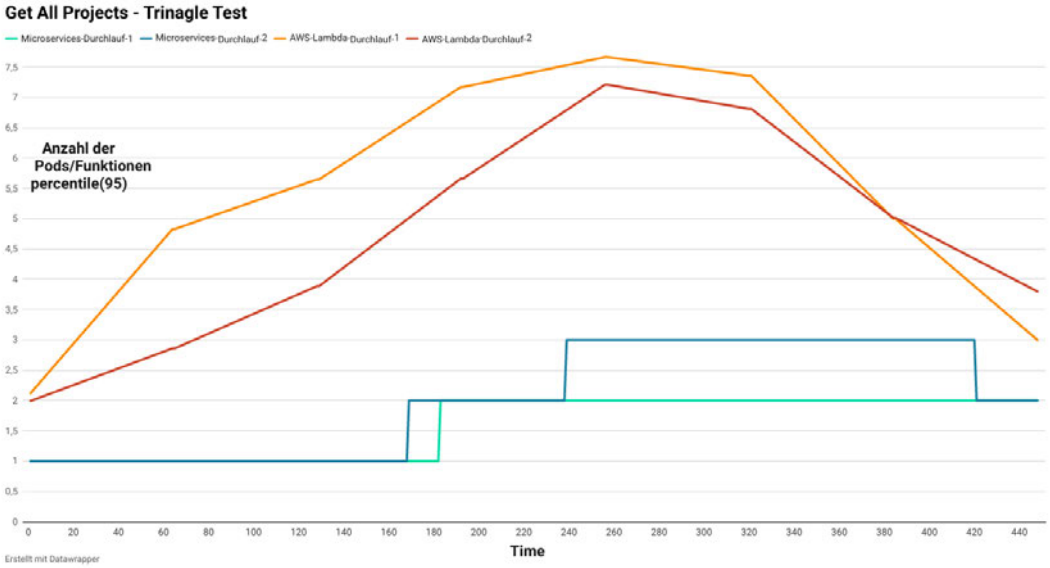

Abbildung 22: Microservices und AWS Lambda - Skalierbarkeit über die Zeit x des ”GET /projects” Endpoints

5.4.4 Kosten

Wie in Kapitel 5.2.3 "Verteilungssicht" zu sehen ist, verwenden beide Deploymentstrategien AWS. In diesem Kapitel werden die AWS-Kosten für beide Deploymentstrategien aufgeführt und beschrieben.
Da die Anwendungen auf dem AWS-Konto von HanseCom deployt wurden, können die Kosten für diese Anwendungen nicht von den übrigen AWS-Kosten des Unternehmens getrennt werden. Ein Beispiel hierfür ist AWS ECR: Viele Docker-Images werden in AWS ECR gespeichert, worunter auch die Docker-Images der Anwendungen dieser Arbeit fallen. Aus diesem Grund wurde die Kostenberechnung mit AWS Pricing Calculator durchgeführt. Da AWS Pricing Calculator verwendet wird, wird die Kostenberechnung nicht für Load Testing durchgeführt. Diese Berechnung wird für ein Szenario mit einem stabilen Traffic von 2.700.000 Requests pro Monat durchgeführt.
Die folgenden Services wurden in der Microservices-Deploymentstrategie verwendet:

- AWS ECR - pro Monat werden 11.664 MB an Docker-Images in AWS ECR gespeichert. Die Timesheets-Anwendung hat eine Größe von 364 MB und die Project-Anwendung von 365 MB. Es kann davon ausgegangen werden, dass sich die Anwendungen weiterentwickeln und dass jede Anwendung viermal pro Woche neu deployt wird. Das sind 32 Deployments pro Monat oder 16 Deployments pro Projekt. Das ergibt 11.664 MB pro Monat.
- AWS Load Balancer (external) - pro Monat werden 2.700.000 Requests gesendet. Das bedeutet, dass etwa 1.808 MB an Daten pro Monat an den Load-Balancer gesendet werden. Diese Zahl könnte ausgerechnet werden, da in einem Performance-Testing Durchlauf ca. 113 MB Daten an Load-Balancer gesendet werden. Ein Durchlauf mit fünf Endpoints ergibt ca. 170.000 Requests. Das bedeutet, dass ca. 1.808 MB Daten pro Monat an den Load-Balancer gesendet werden.
- AWS EKS - hat einen festen Preis pro Monat.
- AWS Aurora - der Datenbanktyp ist "db.t3.large", es handelt sich um einen Cluster mit zwei Instanzen: Writer und Reader.
- AWS EC2 - für das Load Testing wurden drei Instanzen vom Typ "r6.large" verwendet. Damit wurde sichergestellt, dass die Pods skaliert werden können und es keinen Bottleneck gibt.
- AWS VPC

Region	Service	Monatlich	Jährlich	Währung
EU (Frankfurt)	Amazon Aurora MySQL-Compatible	142.64	1711.68	USD
EU (Frankfurt)	Amazon Elastic Container Registry	1.1391	13.67	USD
EU (Frankfurt)	Application Load Balancer	19.94	239.28	USD
EU (Frankfurt)	Amazon EKS	73	876	USD
EU (Frankfurt)	Amazon Virtual Private Cloud (VPC)	0.0	0.0	USD
EU (Frankfurt)	Amazon EC2	220.95	2651.4	USD

Tabelle 6: Kosten des Microservices-Systems

Es ist zu erkennen, dass EC2-Instanzen und die Aurora-Datenbank viel Geld kosten. Dies geschieht aus dem Grund, dass bei dem Setup für das Testen dafür gesorgt wurde, dass die zwei Komponenten keinen Bottleneck im System darstellen. Für die Diskussion in Kapitel 5.5 hat diese Wahl keine große Auswirkung.
Die folgenden Services wurden in der Serverless-Deploymentstrategie verwendet:

- AWS API Gateway - 2.700.000 Requests werden über das Gateway gesendet. Dabei handelt es sich um REST-API-Endpoints.
- AWS Lambda - für AWS Lambda ist die Requestdauer für die Kostenberechnung wichtig. Jeder Request dauert etwa 58 ms. Diese Zeit wurde aus dem Durchschnitt des Load Testings ermittelt.
- AWS Aurora - der Datenbanktyp ist ”db.t3.large”, es handelt sich um einen Cluster mit zwei Instanzen: Writer und Reader.
- AWS VPC

Region	Service	Monatlich	Jährlich	Währung
EU (Frankfurt)	Amazon Virtual Private Cloud (VPC)	0	0	
EU (Frankfurt)	Amazon API Gateway	10.06	120.72	USD
EU (Frankfurt)	AWS Lambda	1.85	22.2	USD
EU (Frankfurt)	Amazon Aurora MySQL-Compatible	142.64	1711.68	USD

Tabelle 7: Kosten des Serverless-Systems

Die in den beiden Tabellen aufgeführten Kosten sind nur eine Schätzung der Kosten und können von den tatsächlichen Kosten abweichen. Die aufgeführten Kosten wurden ohne AWS Free Tier erstellt.

5.5 Diskussion

In Kapitel 5.4.1 ”Erhöhte CPU- und Memory-Werte” wurden die Ergebnisse der zwei Tests vorgestellt. In der Abbildung 10 ist zu sehen, dass keine Skalierung der Microservices-Anwendung stattfand. Es wurde versucht, eine Umgebung zu schaffen, in der Microservice-Anwendungen skalieren und mehrere AWS Lambda-Funktionen parallel ausgeführt werden, um zu bewerten, welche Deploymentstrategie eine bessere Ressilience und Skalierbarkeit bietet. Aus diesem Grund wurden die Ausführungszeit, CPU- und Memory-Werte angepasst.
In Kapitel 5.4.2 ”Geringe CPU- und Memory-Werte” sind in den Diagrammen große Ausschläge zu sehen, dies ist insbesondere in Abbildung 15 und 17 für die Microservice-Anwendung zu erkennen. Dies ist darauf zurückzuführen, dass der Microservice
-Anwendungen nur 0,07 CPU-Cores bzw. 70 Millis für die Parameter ”requests” und ”limits” zugewiesen wurden. Die Microservice-Anwendungen erreichten den ”limits”-Wert zu schnell und konnten nicht alle Requests in einer erwarteten Zeit von < 100 ms verarbeiten.

Tabelle 8 und 9 zeigen die CPU-Werte der Kubernetes-Pods während der triangle und incremental Tests des ”PUT /timesheet/<id>” Endpoints, die in der Abbildung 15 und 17 zu sehen sind. Die Tabellen zeigen den CPU-Wert der Pods in Abhängigkeit von der Anzahl der zu diesem Zeitpunkt ausgeführten VUs. Abbildung 15 zeig, dass in dem Bereich von 40 VUs bis zum Ende des Tests große Ausschläge zu erkennen sind. In Tabelle 9 ist zu erkennen, dass Pod 1 bereits ab 40 VUs mehr als die Hälfte seiner CPU-Ressourcen verbraucht. Bis zum Ende des Tests verbraucht Pod 1 weiterhin einen großen Teil seiner CPU-Ressourcen. Bei 80 VUs und 130 VUs kommt es zu einer Skalierung der Pods, aber den Pods wird weniger Traffic zugewiesen als den Pod 1.
Diese Beobachtung ist in Abbildung 17 und Tabelle 8 noch besser zu erkennen. In Abbildung 17 sind große Ausschläge in der Microservice-Anwendung im Bereich von 40 VUs bis zu 100 VUs bzw. 310 Sekunden zu sehen. In Tabelle 8 ist zu sehen, dass Pod 1 der Microservice-Anwendung ab 40 VUs einen großen Teil seiner CPU-Ressourcen verbraucht. Dies geschieht bis zu 100 VUs bzw. 310 Sekunden. Ab 100 VUs stabilisiert sich der Verbrauch der CPU-Ressourcen und der Pod 1 verbraucht weniger als 50% seiner Ressourcen. Auch hier zeigt sich das Problem, das in Tabelle 9 zu sehen war: Pod 2 und Pod 3 werden bei 120 VUs und 130 VUs erstellt, aber der Traffic wird nicht gerecht auf die Pods verteilt.
Es ist ersichtlich, dass die Microservices-Deploymentstrategie unter einem

Triangle Test			
VUs	**Pod 1**	**Pod 2**	**Pod 3**
1	0m	%	%
10	3m	%	%
20	7m	%	%
30	35m	%	%
40	47m	%	%
50	47m	%	%
60	47m	%	%
70	60m	%	%
80	60m	%	%
90	60m	%	%
100	57m	%	%
110	62m	%	%
120	62m	15m	%
130	66m	30m	%
140	66m	26m	%
150	64m	43m	%
160	66m	36m	%
150	71m	42m	%
140	69m	42m	%
130	65m	34m	4m
120	65m	27m	15m
110	60m	26m	15m
100	53m	24m	15m
90	49m	23m	14m
80	42m	23m	9m
70	42m	23m	9m
60	39m	20m	8m
50	35m	18m	7m
40	28m	18m	6m
30	25m	11m	6m
20	22m	6m	4m
10	22m	6m	2m
1	13m	4m	1m

Tabelle 8: CPU-Werte der Kubernetes Pods der Microservice-Anwendung beim Testen des ”PUT /timesheet/<id>” Endpoints mit Trinagle Tests

Incremental Test			
VUs	**Pod 1**	**Pod 2**	**Pod 3**
1	1m	%	%
10	1m	%	%
20	14m	%	%
30	32m	%	%
40	44m	%	%
50	47m	%	%
60	56m	%	%
70	56m	%	%
80	55m	22m	%
90	46m	18m	%
100	51m	35m	%
110	51m	34m	%
120	52m	33m	%
130	60m	36m	28m
140	55m	43m	3m
150	57m	36m	8m
158	61m	38m	9m

Tabelle 9: CPU-Werte der Kubernetes Pods der Microservice-Anwendung beim Testen des ”PUT /timesheet/<id>” Endpoints mit incremental Tests

Load-Balancing- und Trafficverteilungs-Problem leidet.
Es ist auch zu erkennen, dass die Kubernetes-Skalierbarkeits und Ressourcen-Konfiguration wichtige Rollen haben. Die Konfiguration muss abhängig vom Use Case der Anwendung gemacht werden. Das bedeutet, dass die Konfiguration immer abhängig vom Traffic der Anwendung sein muss. Für eine Microservice-Anwendung, bei der die Zahl der "Requests pro Sekunde" niedrig ist, für eine, bei der sie hoch ist, und für eine dritte, bei der sie nicht vorhersehbar ist, muss die Konfiguration unterschiedlich sein. Der Vorteil einer Serverless-Anwendung besteht darin, dass es keine Skalierbarkeitskonfiguration gibt bzw. diese vom AWS Lambda-Skalierbarkeitsalgorithmus übernommen wird.
Aus dieser Argumentation, kann festgestellt werden, dass die Kubernetes-Skalierbarkeits- und Ressourcen-Konfiguration aufwändiger als die von Serverless ist.

Die Abbildungen 11, 13 und 15 zeigen, dass die AWS Lambda Anwendung zwar Ausschläge aufweist, diese aber nicht kontinuierlich, sondern nur in seltenen Fällen auftreten. AWS Lambda Funktionen haben das Problem nicht, denn wenn mehr Traffic auftritt, werden neue Funktionen erstellt, die eine Zeit lang "warm" bleiben (sie werden nicht beendet und sind für neue Requests bereit). In der Abbildung 17 ist das Gegenteil zu sehen, die AWS Lambda Anwendung hat kontinuierliche Ausschläge im Bereich zwischen 25 und 160 VUs. In der Abbildung 18 ist die Ursache für die Ausschläge erkennbar, die Anzahl der parallel ausgeführten Funktionen nimmt zu und bei etwa 160 VUs kommt es zu 25 parallel ausgeführten AWS Lambda Funktionen. Für jede der ausgeführten AWS Lambda Funktionen musste eine Umgebung gestartet werden. Beim Starten der Funktionen kommt es zu Cold-Start.
In Kapitel 5.4.3 "Angepasste CPU- und Memory-Werte" wurden die CPU- und Memory-Werte erneut angepasst, um eine Umgebung zu schaffen, in der bewertet werden kann, welche Deploymentstrategie die bessere Ressilience und Skalierbarkeit bietet. Durch die Anpassung der Werte war die Umgebung erfolgreich und daher wurden diese Tests in den vorherigen Kapiteln als "valide" Tests bezeichnet. Aus diesem Grund wurden auch zwei Durchläufe beim Testen durchgeführt.
Bei den Abbildungen im Kapitel 5.4.3 sind Cold-Starts auch zu erkennen. Beim ersten Request der Serverless-Anwendung ist in jedem Testlauf ein Cold-Start zu sehen. Ein Beispiel hierfür ist die Abbildung 19. Hier ist zu sehen, dass Cold-Start bei Sekunde 1 in AWS-Lambda-Durchlauf 1 und AWS-Lambda-Durchlauf 2 auftritt. AWS-Lambda-Durchlauf 1 benötigte etwa 1420 ms für die Bearbeitung des Requests und AWS-Lambda-Durchlauf 2 etwa 750 ms. **Daraus lässt sich schließen, dass die Serverless-**

Deploymentstrategie unter einem Problem des Cold-Starts leidet.

Eine weitere Beobachtung, die sich aus den Tabellen 4 und 5 ablesen lässt, ist dass die Microservices-Deploymentstrategie die Reqeuests schneller verarbeitet. Es ist zu erkennen, dass die Microservice-Anwendugen in jedem Test und jedem Durchlauf einen höheren Min-, Avg-, Max- und p(95)-Wert hatten als die Serverless-Anwendungen. **Dadurch kann festgestellt werden, dass die Microservices-Deploymentstrategie eine bessere Requestsbearbeitungsleistung als die Serverless-Deloymentstrategie aufweist.** Dabei ist wichtig zu erkennen, dass der Unterschied nicht groß ist und sich in der Praxis bei der meisten Anwendungen nicht bemerkbar machen würde.

Ein weiterer Vorteil der Serverless-Deploymentstrategie ist der, dass die Serverless-Deploymentstrategie agiler in Bezug auf Skalierbarkeit ist. In diesem Kapitel wurde erwähnt, dass die Microservice-Anwendungen einen CPU-Wert erreichen müssen, um skaliert werden zu können. Dies ist bei der Serverless-Deploymentstrategie nicht der Fall, die Skalierbarkeit hängt vom Traffic ab. Das bedeutet, dass eine neue Funktion ausgeführt wird, wenn neue Requests gesendet werden und keine Funktion vorhanden ist, die diese Requests bearbeiten kann.

In Kapitel 5.4.4 ”Kosten” wurden die Ergebnisse der Kostenberechnung der Deploymentstrategien aufgeführt. Bei der Microservices-Deploymentstrategie ist zu erkennen, dass sich die monatlichen Ausgaben auf etwa 457,6 USD belaufen. Jährlich entspricht dies etwa 5.492,03 USD. Die meisten Kosten verursachen folgende Services: AWS EC2, AWS Aurora und AWS EKS. Hier ist wichtig zu erwähnen, dass AWS EKS nicht eingesetzt werden muss, Kubernetes kann auch ”self-managed” werden. Bei der AWS Aurora-Datenbank kann ein Cluster mit weniger Ressourcen gewählt werden, dies wird auch für die monatliche Anzahl von Requests ausreichen. Die drei ”r6.large” EC2-Instanzen können gegen eine oder zwei kleinere Instanzen ausgetauscht werden. Dies führt dazu, dass die Microservices-Deploymentstrategie weniger kostet, es handelt sich jedoch immer noch um laufende Kosten. Das heißt, wenn die Anwendung keine Requests erhält, fallen trotzdem Kosten an, da sie auf einer AWS EC2-Instanz ständig läuft.
Bei der Serverless-Deploymentstrategie liegen die monatlichen Kosten bei 154,55 USD bzw. 1.854,6 USD pro Jahr. Hier ist zu sehen, dass die Kosten der Serverless -Deploymentstrategie deutlich geringer sind als die Kosten der Microservices -Deploymentstrategie. Der Grund dafür ist, dass die Serverless-Deploymentstrategie AWS EC2 und AWS EKS nicht verwendet. Trotz dieser Ergebnisse kann nicht festgestellt wer-

den, dass die Serverless-Deploymentstrategie allgemein weniger Kosten als die Microservices-Deloymentstrategie verursacht. Dies ist aus folgendem Grund nicht möglich: Die Kosten in der Microservices-Deploymentstrategie können reduziert werden. Wie bereits erwähnt, muss die AWS EKS nicht benutzt werden und die AWS EC2 Instanzen können mit anderen Instanztypen ausgetauscht werden. Um die genauen Kosten der beiden Deploymentstrategien zu vergleichen, müssen in der Zukunft weitere Experimente durchgeführt werden.
Es lässt sich jedoch feststellen, dass AWS Lambda nur dann Kosten verursacht, wenn die AWS Lambda-Funktionen ausgeführt werden. Dies steht im Gegensatz zu der Microservices-Deploymentstrategie. Wenn eine Microservice-Anwendung auf Kubernetes auf einer EC2-Instanz ausgeführt wird und keine Requests erhält, fallen trotzdem Kosten an.
Es ist ersichtlich, dass die Serverless-Deploymentstrategie, die AWS Lambda verwendet, für eine Anwendung mit geringerem Traffic weniger kostet als eine Microservices-Deploymentstrategie, die EC2 verwendet.
Ob eine Microservices-Deploymentstrategie für eine Anwendung mit höherem Traffic weniger Kosten verursacht, lässt sich anhand der Kostenberechnung in Kapitel 5.4.4 nicht feststellen.

In dem in Kapitel 3.1 beschriebenen Paper werden drei der sechs hier beschriebenen Erkenntnisse bestätigt. Das Paper kommt zu vier Erkenntnissen. Die vierte in dem Paper aufgeführte Erkenntnis unterscheidet sich von der Erkenntnis in dieser Arbeit. In dieser Arbeit wurde gezeigt, dass die Requestsbearbeitungsperformance der Microservice-Anwendung besser ist als bei Serverless-Anwendungen. Das Paper kommt zu anderen Ergebnissen. Da das Paper nicht genug Informationen über die Implementierung der Anwendungen enthält, kann nicht festgestellt werden, warum dies der Fall ist.

Aus den Erkenntnissen der Arbeit gehen bestimmte Use Cases hervor für die eine Deployentstrategie besser geeignet ist als die andere.
Durch die Erkenntnisse zeigt sich, dass die Microservices-Deploymentstrategie für zwei Use Cases geeignet ist:

- Real-Time-Data-Processing - Big-Data-Anwendungen erfassen große Mengen an unstrukturierten Daten. Diese Daten können z.B. aus sozialen Netzwerken stammen und müssen in Echtzeit verarbeitet werden. Hier kann eine Microservices-Deploymentstrategie verwendet werden, da die Serverless-Deploymentstrategie unter dem Problem des Cold Starts leidet. Das bedeutet, dass die Serverless

-Deploymentstrategie zu Beginn der Datenverarbeitung die Daten nicht in Echtzeit verarbeiten kann.

- Anwendungen, die auf Sekundenbruchteile angewiesen sind - hier eignet sich die Microservices-Deploymentstrategie, da sie eine bessere Requestsbearbeitungsleistung als die Serverless-Deploymentstrategie aufweist.

Es ergeben sich auch andere Use Cases für die Serverless-Deploymentstrategie. Diese sind im Folgenden aufgeführt:

- Anwendungen mit Traffic-Schwankungen - hier empfiehlt sich die Serverless -Deploymentstrategie, da sie einfacher zu konfigurieren und in Bezug auf die Skalierbarkeit agiler ist. Eine Konfiguration kann für alle Traffic-Schwankungen verwendet werden. Dies ist bei der Microservices-Deploymentstrategie nicht der Fall. Ein weiter Grund ist, dass die Microservices-Deploymentstrategie unter dem Load-Balancing- und Trafficverteilungs-Problem leidet. Dies führt zu Problemen, wenn der Traffic schwankt.

- Automatisch skalierbare APIs - für diese Anwendungen sind eine flexible Skalierbarkeit und eine einfache Konfiguration von Vorteil. Deshalb bietet sich für diesen Use Case die Serverless-Deploymentstrategie an.

- Job-Scheduler (CronJobs) - diese werden in den meisten Fällen nicht oft ausgeführt. Dies bedeutet, dass die Anwendungen nicht viel Traffic erzeugen. Wie aus den Ergebnissen hervorgeht, verursacht eine Serverless-Anwendung bei geringerem Traffic weniger Kosten als eine Microservice-Anwendung. Aus diesem Grund ist die Serverless-Deploymentstrategie für diesen Use Case geeignet.

Die Empfehlungen beziehen sich nur auf die Kosten und die Performance (Skalierbarkeit, Resilience und Cloud Latency) der Deploymentstrategien. Bei der Auswahl der Deploymentstrategie sollten auch andere Kriterien berücksichtigt werden, da in dieser Arbeit nur eine kleine Teilmenge der Kriterien angesprochen und untersucht wird.

6 Customer-Reporting-System

In diesem Kapitel werden die Implementierung und die Ergebnisse des Performance-Testings der in Kapitel 1.1. "Motivation, Zielsetzung und Fragestellung" erwähnten Event-driven Customer-Reporting-Anwendung vorgestellt. Da viele Konzepte bereits in Kapitel 5 "Employee-Time-Sheet-Management-Portal" vorgestellt wurden, werden diese Konzepte in diesem Kapitel nicht erneut erläutert. Aus diesem Grund wird den Lesenden empfohlen, Kapitel 5 zu lesen.
Im ersten Teil des Kapitels werden die Anforderungen an das System vorgestellt. Außerdem wird der Unterschied zwischen SQL und KSQL erläutert.
In den folgenden Kapiteln werden wie in Kapitel 5 zuerst der Architekturentwurf und die Architekturentscheidungen diskutiert. Im weiteren Verlauf des Kapitels wird das architektonische Setup und das Setup der Tests beschrieben. Weiterhin werden die Details der Tests besprochen und am Ende werden die Ergebnisse der Tests und die Kosten vorgestellt und diskutiert.

6.1 Anforderungen

Das Handy-Ticket-Deutschland-System verfügt über ein registrierungsfreies Ticketkaufsystem. Dieses System ermöglicht den Benutzer*innen, ein Ticket ohne vorherige Registrierung und Anmeldung zu kaufen. Für diese Anwendung müssen Customer-Reports erstellt werden. In dieser Arbeit wird der Prototyp des Systems erstellt, der die Customer-Reports erstellt und sie den Verkehrsunternehmen zur Verfügung stellt.
Um die Customer-Reports erstellen zu können, muss das Customer-Reporting-System über eine Datenquelle verfügen. Für das registrierungsfreie Ticketkaufsystem wird intern Kafka verwendet, wobei das System das Saga-Pattern benutzt und die Events in die Kafka-Topics schreibt. Die Datenquelle für das Customer-Reporting-System sind die vom registrierungsfreien Ticketkaufsystem erstellten Events.
Für das Customer-Reporting-System werden gemockte Events verwendet, die von einem zu diesem Zweck implementierten Service erstellt werden.

Um zu verstehen, wie die Customer-Reports aussehen, ist es zunächst notwendig zu definieren, was Customer-Reports sind. In dieser Arbeit werden die Customer-Reports als Statistiken definiert, die Informationen über die verkauften Tickets über einen Zeitraum x liefern.

Die Customer-Reports werden als CSV-Dateien erstellt, jedes Verkehrsunternehmen erhält seine eigenen Customer-Reports und kann die Daten der anderen Verkehrsunternehmen nicht einsehen. Dafür wird eine Authentifizierung implementiert, aber da dies nicht das Ziel der Arbeit ist, wird diese Anforderung im Prototypen nicht umgesetzt. Eine der wichtigsten Anforderungen ist, dass es keine Kund*innen- und Paymentdaten in den Customer-Reports geben soll.
Die folgenden Felder werden in den CSV-Dateien der Customer-Reports angefordert:

- Purchase-Order-ID
- Beleg-Typ
- Beleg-Nr.
- Ausgabedatum
- Gültigkeitsbeginn
- Gültigkeitsende
- Produkt-ID
- Preis
- Tarifzone
- Startort
- Zielort

Um die oben genannten Anforderungen zu erfüllen, muss ein Endpoint implementiert werden. Dieser Endpoint ist ein POST Endpoint mit der ”kvp” Pathvariable. Die Pathvariable ist die ID des Verkehrsunternehmens, es werden nur die Reports des Verkehrsunternehmens zurückgeliefert. Als Request Body muss ein JSON mit ”start_time” und ”end_time” angegeben werden. Diese beiden Werte definieren den Zeitraum für den die Customer-Reports erstellt werden sollen. Die Rückgabe des Endpoints ist eine CSV-Datei mit den oben genannten Feldern.
Definition des Endpoints: POST /reports/<kvp>
Body des Requests:
{”start_time” : ”2021-07-23T00:00:00Z”,
”end_time” : ”2021-08-23T23:59:59Z”
}

6.2 SQL und KSQL

Wie bereits erwähnt, werden Kafka-Events als Datenquelle für die Customer-Reports verwendet. Die Events müssen aggregiert werden. Zu diesem Zweck kann eine Anwendung implementiert oder KSQL verwendet werden. In dieser Arbeit wurde entschieden, die KSQL-Lösung zu verwenden.
Die Aggregation der Daten wird in KSQL anders durchgeführt als in einer traditionellen relationalen Datenbank. Der Unterschied wird in diesem Kapitel dargestellt.
Abbildung 23 zeigt ein Beispiel für ein ER-Modell einer relationalen Datenbank, das als Quelle für die Erstellung von Customer-Reports verwendet werden kann.

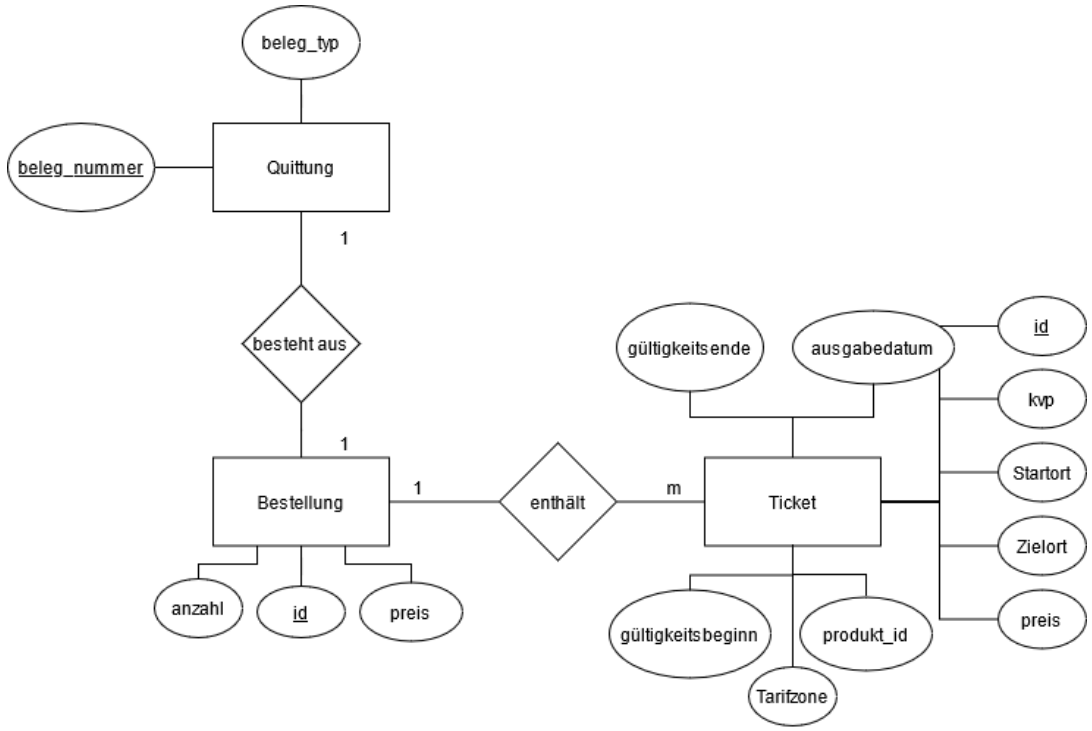

Abbildung 23: Beispiel eines ER-Modelles einer Datenbank, die für die Customer-Reporting benutzt werden kann

Es ist zu erkennen, dass das ER-Modell aus drei Entities besteht: Bestellung, Ticket und Quittung. Die Entities bestehen aus zwei Beziehungen; die Bestellung hat eine Beziehung zur Quittung und eine zum Ticket. Die Bestellung besteht aus einer Anzahl von Produkten bzw. Tickets mit einer ID und einem Preis. Die Quittung besteht nur aus der

Beleg-Nummer bzw. der Beleg-ID und dem Belegtyp. Das Ticket besteht aus folgenden Attributen: ID des Tickets, Ausgabedatum, Beginn und Ende der Gültigkeit, Tarifzone, Start- und Zielort, Preis, Produkt-ID und KVP. Das Attribut KVP definiert zu welchem Verkehrsunternehmen das Ticket gehört.
Folgend ist ein SQL-Statement zu sehen, mit welchem die Daten für die Customer-Reports aggregiert werden.

```
SELECT b.id, q.beleg_typ, q.beleg_nummer,
t.ausgabedatum, t.gueltigkeitsbeginn,
t.gueltigkeitsende, t.produkt_id,
t.preis, t.tarifzone, t.startort, t.zielort
FROM bestellung b
JOIN quittung q
    ON q.bestellung_id = b.id
JOIN ticket t
    ON t.bestellung_id = b.id
    AND kvp = 6016
    AND t.ausgabedatum
    between '2021-07-23 00:00:00'
    and '2021-07-23 23:59:59';
```

Im SQL-Statement ist zu sehen, dass die Daten, die sich in den Tabellen befinden, über einen JOIN aggregiert werden, die Aggregation folgt über einen Zeitraum und über das Attribut "kvp".
In KSQL muss der Ansatz von Pull- und Push-Queries benutzt werden, der in Kapitel 2 "Grundlagen" beschrieben wurde. Zunächst werden für die verschiedenen Topics (Ticket, Bestellung und Quittung) die Tabellen erstellt, die den Zustand der Daten ändern, wenn neue Events in das Topic geschrieben werden. In KSQL werden Tabellen oder Streams aus Kafka-Topics erstellt. Die anderen Tabellen mit Abfrageergebnissen werden aus diesen genannten Tabellen oder Streams erstellt. Eine neue Tabelle mit dem Namen "REPORTING_DATA" wird aus den vorherigen Tabellen (Ticket, Bestellung und Quittung) erstellt. Folgend ist der Code für die Erstellung der Tabelle zu sehen.

```
CREATE TABLE "REPORTING_DATA" AS
SELECT
    rec.RECEIPTTYP, rec.RECEIPTNUMBER,
```

```
        tic.PURCHASEORDERID AS PURCHASEORDERID,
        tic.TICKETDATA->"PURCHASEDATE" AS PURCHASEDATE,
        tic.TICKETDATA->"VALIDFROM" AS VALIDFROM,
        tic.TICKETDATA->"VALIDTO" AS VALIDTO,
        pur.TARIFFDATA->"PRODUCTID" AS PRODUCTID,
        tic.TICKETDATA->"PRICE" AS PRICE,
        pur.TARIFFDATA->"TARIFFZONEFROM" AS
        TARIFFZONEFROM,
        tic.TICKETDATA->"STARTZONE" AS STARTZONE,
        tic.TICKETDATA->"DESTINATIONZONE"
        AS DESTINATIONZONE,
        tic.TICKETDATA->"KVP" as KVP
    FROM "RECEIPT_THESIS_TABLE" rec
        JOIN "PURCHASE_ORDER_THESIS_TABLE" pur
        ON pur.PURCHASEORDERID = rec.PURCHASEORDERID
        JOIN "TICKET_CRATED_THESIS_TABLE" tic
        ON tic.PURCHASEORDERID = rec.PURCHASEORDERID;
```

Es ist zu erkennen, dass in KSQL auch JOIN zur Erstellung einer neuen Tabelle verwendet wird. Nach der Erstellung der Tabelle kann der folgende Query verwendet werden, um die Reporting-Daten zu erhalten.

```
SELECT *
FROM REPORTING_DATA
WHERE ROWTIME > 153163589079
    AND ROWTIME < 1731635890795
    AND KVP = 6126;
```

Es ist zu sehen, dass im Query der Zeitraum und KVP verwendet werden, um die Daten zu aggregieren.
Die konzeptionellen Unterschiede zwischen einer relationalen Datenbank und KSQL sind groß, aber es zeigt sich, dass KSQL auch SQL-Syntax verwendet.
Während der Implementierung wurde festgestellt, dass KSQL viele Einschränkungen hat. Die KSQL-Dokumentation bestätigt [IncoJ], dass KSQL die Topic-Record-Name-Strategy nicht unterstützt. Dies bedeutet, dass mehrere Events im selben Topic nicht von KSQL verarbeitet werden können.
Außerdem wurde festgestellt, dass das Produkt KSQL in ksqlDB umbenannt wurde und

somit zwei Produkte existieren [Kre19]. Die ksqlDB hat eine andere Versionierung als KSQL [Con21]. Die persönliche Meinung des Autors dieser Arbeit ist, dass die Dokumentation der beiden Produkte nicht gut voneinander getrennt ist und dass die zwei unterschiedlichen Versionierungen Probleme verursachen. Es gibt mehrere Dockerhub-Repositories für die Projekte ([Inc21b], [Inc21d] und [Inc21c]), aber nur ein Github-Repository [Inc21e] für beide Projekte bzw. KSQL wird nicht weiterentwickelt und das Github-Repository von KSQL ist zum ksqlDB-Repository geworden. Diese Gründe haben dazu geführt, dass die Dokumentation von ksqlDB für KSQL und ein falsches Docker-Image am Anfang der Implementierung verwendet wurde.

6.3 Setup und Entwurf

In diesem Kapitel werden das Setup der Anwendungen und die Deloymentstrategien vorgestellt. Den Lesenden wird empfohlen, das Kapitel 5.2 "Setup und Entwurf" zu lesen, da viele Konzepte und Diskussionen bereits in diesem Kapitel behandelt wurden und in diesem Kapitel nicht weiter ausgeführt werden.
Für das Customer-Reporting-System wurden zwei Deploymentstrategien verwendet: die Serverless- und Microservices-Deploymentstrategien. Für die Microservices
-Deploymentstrategie wurden Kubernetes und Docker eingesetzt, und für die Serverless-Deploymentstrategie wurden AWS Lambda zusammen mit Serverless-Framework und AWS API Gateway verwendet. Beide Anwendungen verwenden Apache Kafka zusammen mit KSQL zur Aggregation der Events. Der Kafka-Cluster besteht aus drei Kafka-Brokern und dieser wurde mit 53,7 GB Speicherplatz ausgestattet. Die Speicherung der Events im Cluster wurde aus dem Grund konfiguriert, dass die Events nicht gelöscht werden. Dies ist für eine Reporting-Anwendung wichtig, da der Zugriff auf ältere Events für die Erstellung von Customer-Reports gewährleistet sein muss. Für die Events wurden drei Topics angelegt, wobei jedes Topic aus acht Partitionen besteht.
In der KSQL-Konfiguration wurden dem Service 1 CPU-Core bzw. 1.000 Milis und 2,15 GB RAM zugewiesen.
Die beiden Anwendungen verwendeten den Kubernetes- bzw.
AWS Lambda Skalierungsalgorithmus, der bereits in Abschnitt 5.2 erläutert wurde.
Die folgenden Ressourcen wurden für die Anwendungen verwendet:

- Microservices
  - requests

        * RAM - 145 MB
        * CPU - 0.07 CPU-Core bzw. 70 Milis
    - limits
        * RAM - 512 MB
        * CPU - 0,25 CPU-Core bzw. 250 Milis
- Serverless
    - RAM - 512 MB

Der Source-Code der beiden Anwendungen wurde in Python geschrieben. Diese Entscheidung wurde getroffen, weil Python über ausgezeichnete Libraries für die Verarbeitung von CSV-Dateien verfügt. Für die Implementierung des Endpoints wurde das Flask-Framework verwendet. Die beiden Source-Codes unterscheiden sich nur in einem Punkt. Der Microservice-Anwendung wurde ein Healthcheck-Endpoint "/actuator/health" hinzugefügt, der intern von Kubernetes zur Skalierung der Services verwendet wird.

6.3.1 Bausteinsicht

In diesem Kapitel wird die Bausteinsicht der beiden Anwendungen anhand von Komponentendiagrammen dargestellt und diskutiert.

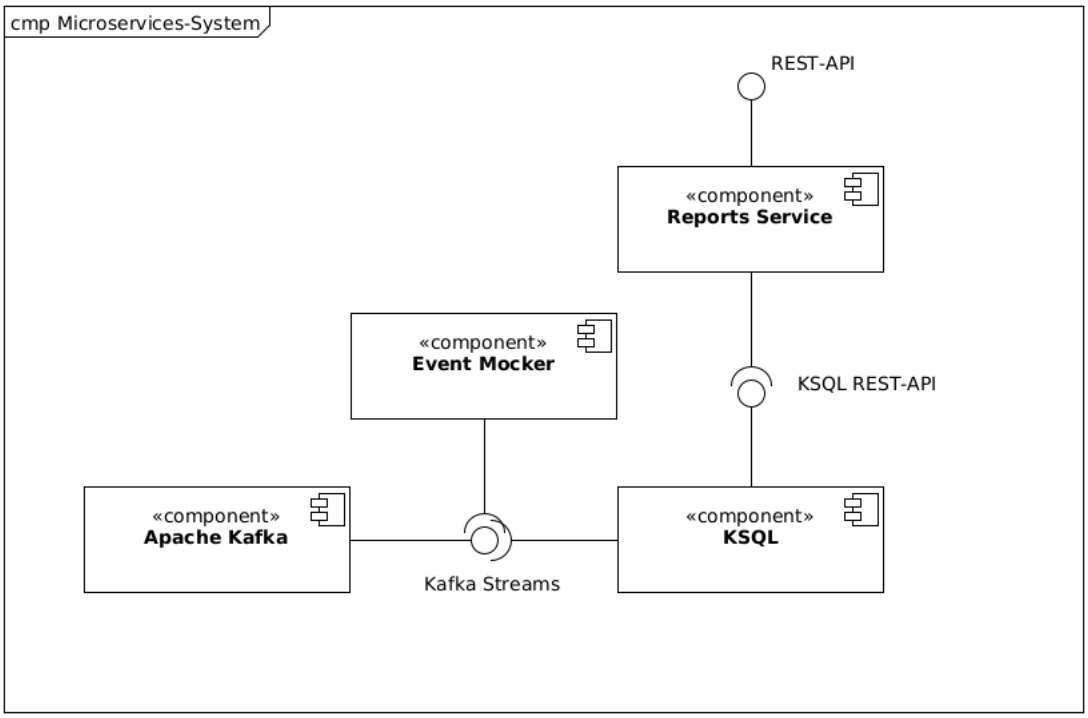

Abbildung 24: Komponentendiagramm Microservices-System, UML nach [OMG17]

Abbildung 24 zeigt das Komponentendiagramm des Microservices-Systems. Das System besteht aus vier Komponenten: Reports-Service, KSQL, Apache Kafka und Event Mocker. Der Einstiegspunkt des Systems ist der Reports-Service, der den Endpoint ”POST /reports/<kvp>” bereitstellt. Der Reports-Service ist mit der REST-API des KSQLs verbunden. KSQL verwendet Kafka-Streams, um die Events aus Kafka zu lesen. Zum Mocken der Events wurde die Anwendung Event Mocker verwendet, die vor dem Testen nur einmal ausgeführt wurde; alle Tests wurden mit denselben Events durchgeführt.

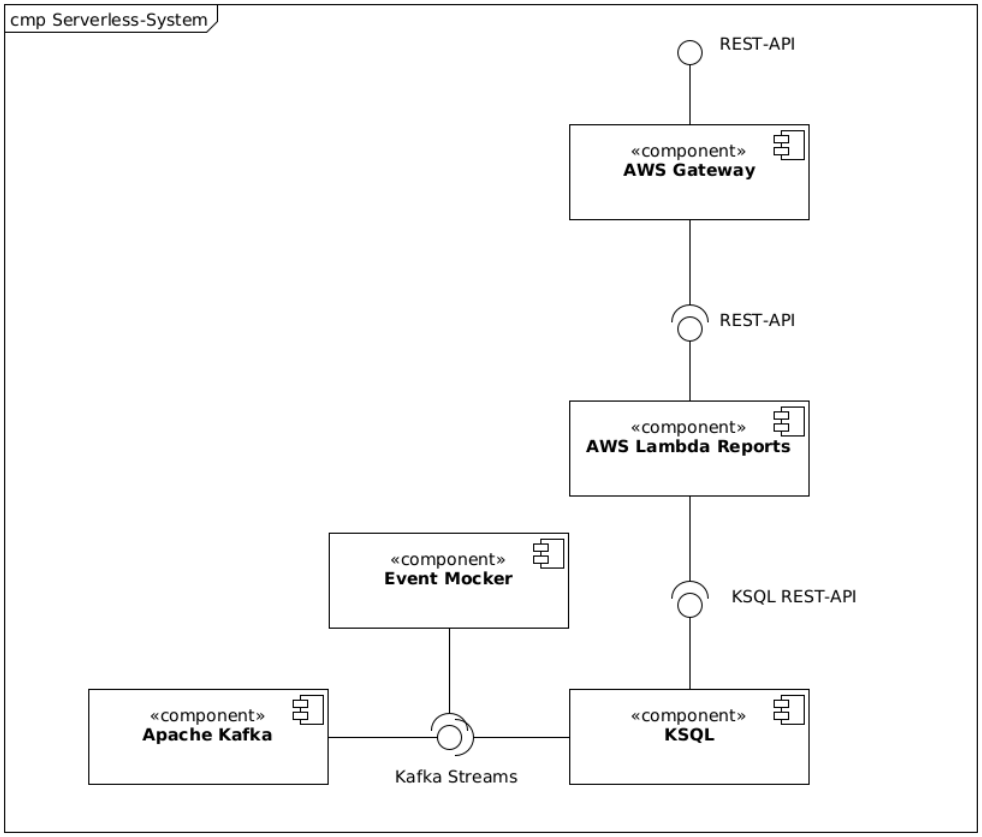

Abbildung 25: Komponentendiagramm Serverless-System, UML nach [OMG17]

Abbildung 25 zeigt das Komponentendiagramm des Serverless-Systems. Der Unterschied zum Microservices-System besteht darin, dass die AWS Lambda Reports-Anwendung eine Serverless-Anwendung ist und ein weiterer Unterschied besteht darin, dass der Endpoint ”POST /reports/<kvp>” über das AWS Gateway public gemacht wird. Das bedeutet, dass die Requests über das AWS API Gateway an die AWS Lambda Reports-Anwendung gesendet werden.

6.3.2 Laufzeitsicht

In diesem Kapitel werden Sequenzdiagramme verwendet, um die Interaktionen zwischen den Komponenten darzustellen. Die Diagramme beschreiben keine Edge-Cases und keine Validierung der Requests, sie beschreiben nur den Happy-Path der Anwendung.

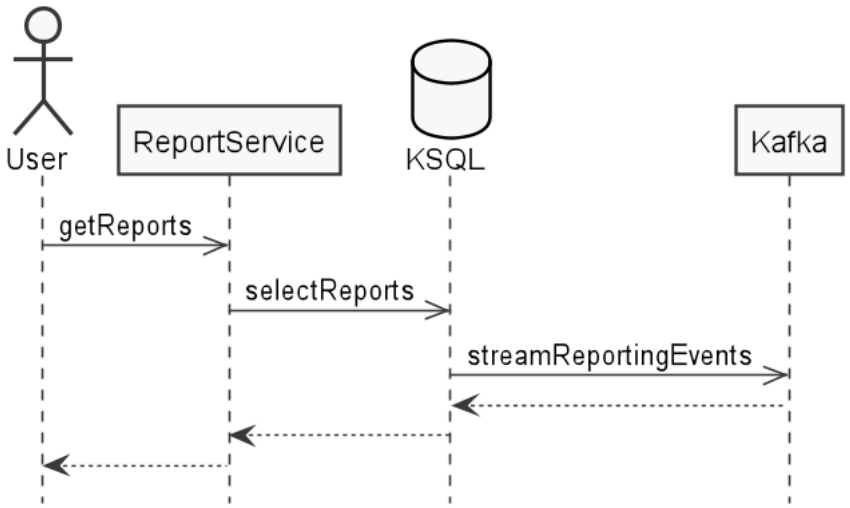

Abbildung 26: Sequenzdiagramm des 'POST /reports/<kvp>" Endpoints in der Microservice-Anwendung, UML nach [OMG17]

Die Abbildung 26 zeigt das Sequenzdiagramm für den "POST /reports/<kvp>" Endpoint der Microservice-Anwendung. Um den Endpoint zu verwenden, muss der User einen Request an die Komponente "ReportService" senden. Die Komponente "ReportService" sendet ein Request an die KSQL REST-API, um die Customer-Reporting-Daten zu erhalten. KSQL verwendet Kafka-Streams, um die Tabellen anzupassen, wenn sich die Events in Kafka ändern. Außerdem sendet KSQL einen JSON Response mit den Customer-Reporting-Daten an "ReportService" zurück. Die Komponente "ReportService" verarbeitet die empfangenen Daten und speichert sie in einer CSV-Datei, die als Response an den User gesendet wird.

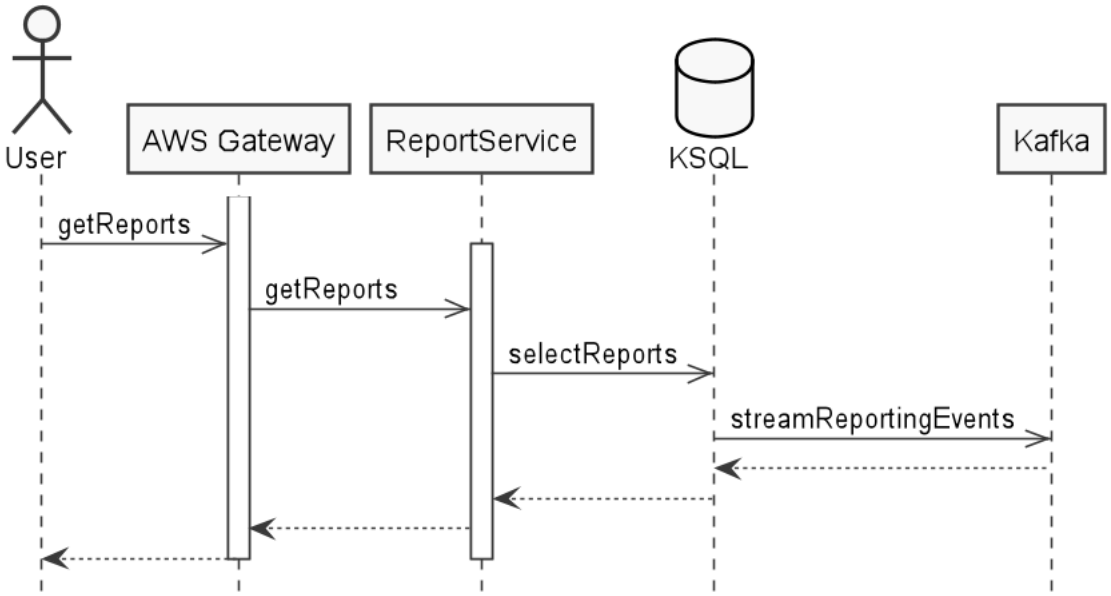

Abbildung 27: Sequenzdiagramm des 'POST /reports/<kvp>" Endpoints in der Serverless-Anwendung, UML nach [OMG17]

Die Abbildung 27 zeigt denselben Endpoint der Serverless-Anwendung. Es ist zu erkennen, dass die Serverless-Deploymentstrategie, eine weitere Komponente enthält. Dabei handelt es sich um die Komponente "AWS Gateway". Der User sendet den Request erst an das AWS Gateway, welches den Request an die Anwendung "ReportService" weiterleitet.

6.3.3 Verteilungssicht

In diesem Kapitel wird die Verteilungssicht der Anwendungen beschrieben. Für die Microservice-Anwendung wird das Deployment mit Kubernetes und Docker beschrieben, für die Serverless-Anwendung mit AWS Lambda und Serverless-Framework. Viele Deployment-Komponenten und -Konzepte wurden bereits in Kapitel 5.2.3 "Verteilungssicht" vorgestellt.

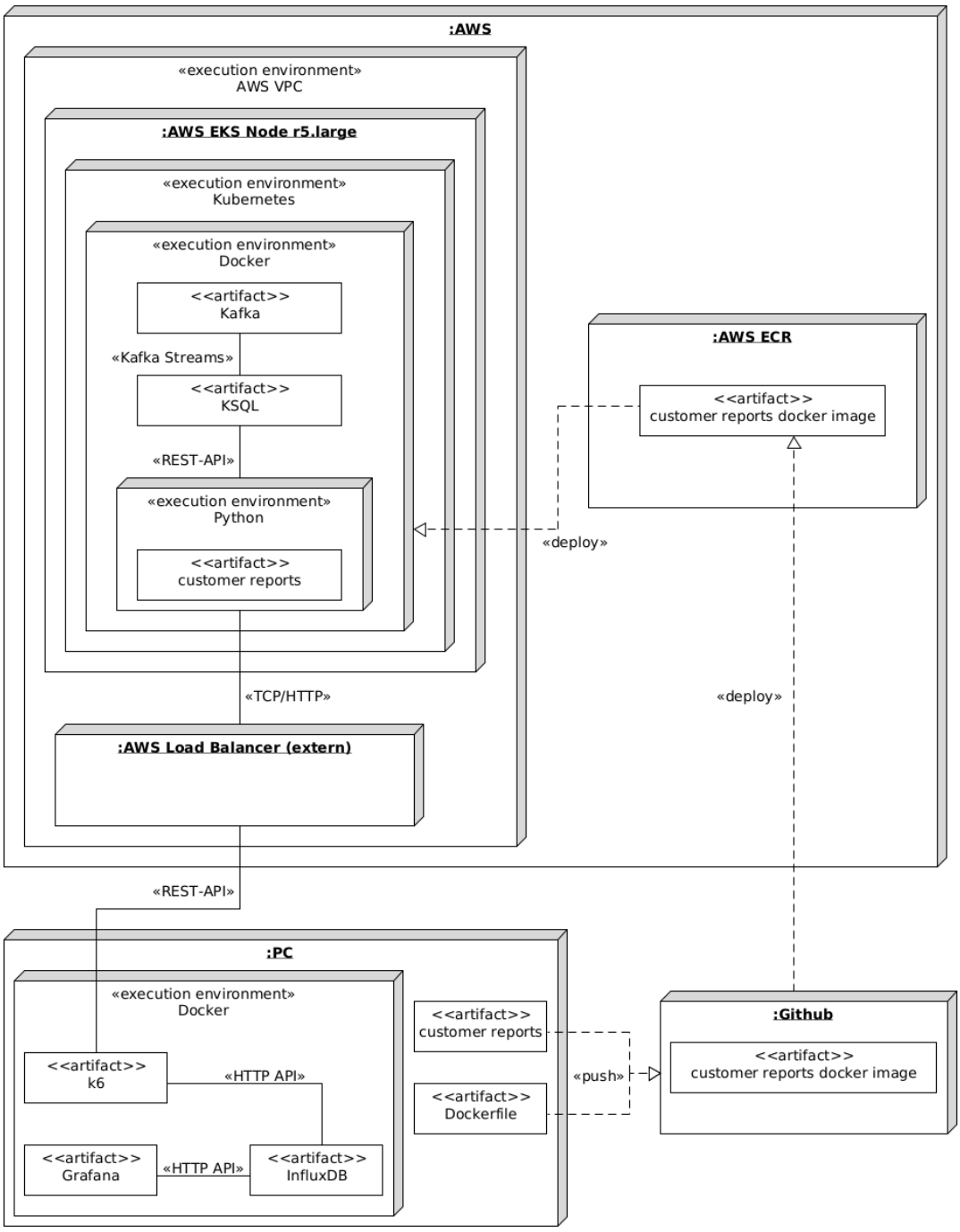

Abbildung 28: Deploymentdiagramm der Microservice-Anwendungen bei der Microservices-Deploymentstrategie mit Kubernetes, UML nach [OMG17]

In Abbildung 28 sind die Konzepte der Nodes: PC, Github, AWS ECR, AWS Load Balancer (extern) und AWS EKS Node gleich zu den Konzepten aus Kapitel 5.2.3. Die Komponenten haben die gleiche Rolle im System. Der Unterschied ist, dass die Komponenten aus Abbildung 28 ”customer reports” Anwendung, KSQL und Kafka auf Kubernetes deployt wurden. Die ”customer reports” Anwendung besteht aus dem zuvor beschriebenen ”POST /reports/<kvp> ” Endpoint. Die Anwendung befindet sich in einem Docker-Container mit einer Python-Umgebung und ist über eine REST-API mit KSQL verbunden. KSQL erhält die Event-Daten von Kafka über die Kafka-Streams-Schnittstelle.

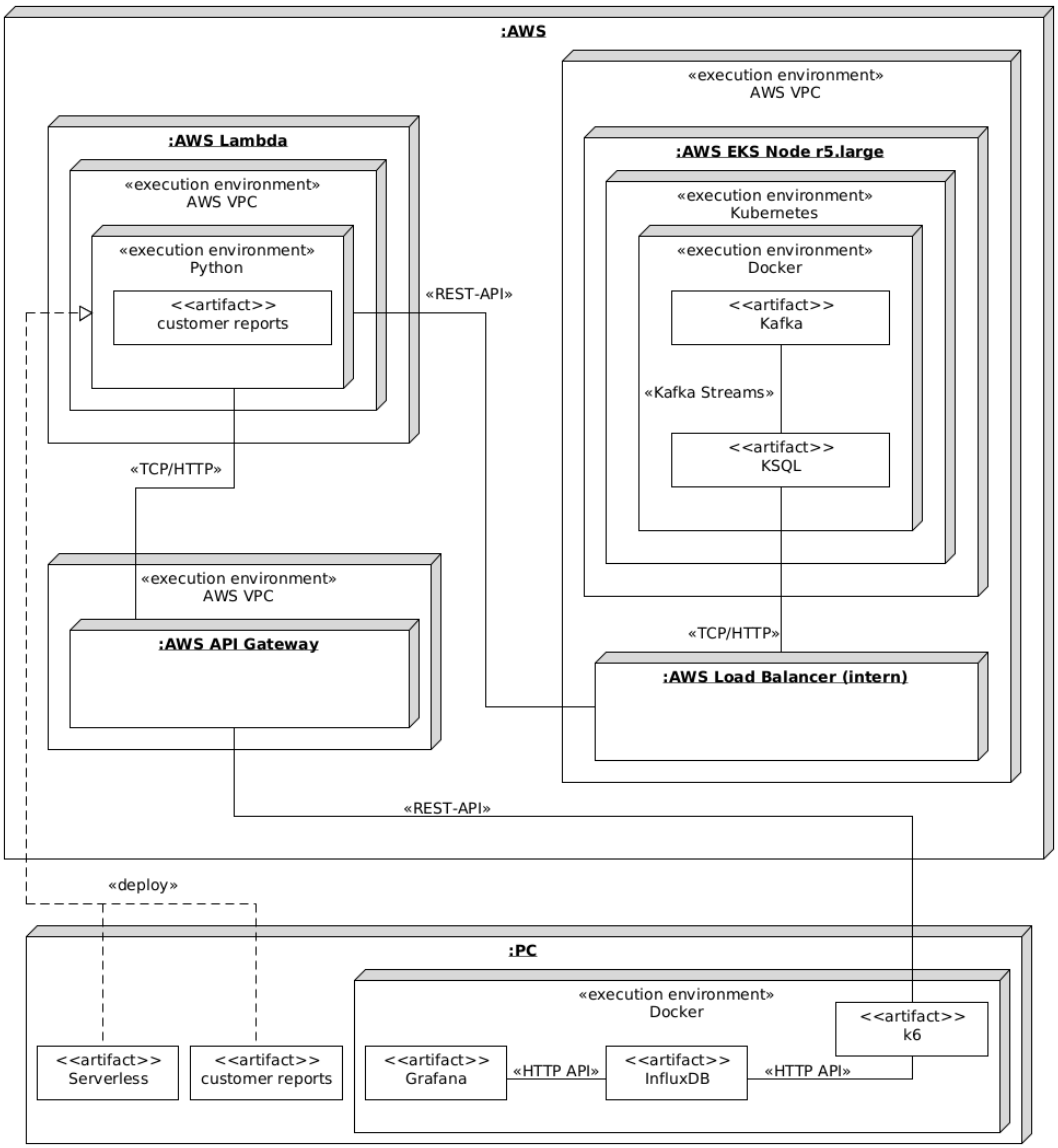

Abbildung 29: Deploymentdiagramm der Serverless-Anwendungen bei der Serverless -Deploymentstrategie mit AWS Lambda und Serverless-Framework, UML nach [OMG17]

Abbildung 29 zeigt das Deploymentdiagramm der Serverless-Anwendung, die mit der Serverless-Deploymentstrategie deployt wurde. Dies bedeutet, dass die Anwendungen als AWS Lambda Funktionen unter Verwendung des Serverless-Framework deployt wurden. Das Konzept der Verteilungssicht einer Serverless-Anwendung bei den Nodes: PC, AWS

API Gateway und AWS Lambda wurden in Kapitel 5.2.3 beschrieben. In Abbildung 29 ist im Gegensatz zu Abbildung 28 zu erkennen, dass die ”customer reports” Anwendung keine Requests direkt an KSQL sendet. Der Grund dafür ist, dass sich die AWS Lambda Funktion ”customer reports” nicht in derselben VPC wie KSQL befindet und dass die Funktion nicht im Kubernetes Cluster deployt wurde. KSQL befindet sich im Kubernetes Cluster und kann nur innerhalb des Clusters erreicht werden. Daher kann die ”customer reports” Microservice-Anwendung aus Abbildung 28 auf KSQL zugreifen, da die Anwendung auch im Kubernetes-Cluster deployt wurde. Um KSQL in der Serverless-Anwendung aus Abbildung 29 zu erreichen, muss KSQL mit einem Load-Balancer verbunden werden. Hierfür wurde ein interner Load-Balancer verwendet. Ein interner Load-Balancer leitet den Verkehr an die Instanz in privaten Subnetzen weiter. Das bedeutet, dass die AWS -Lambda-Funktionen Zugriff auf das private Subnetz haben müssen, wofür die Konfiguration des VPC angepasst werden müsste.

6.4 Methodik

In diesem Kapitel wird das Verfahren zur Durchführung der Tests und zur Erhebung der Daten für das ”Customer-Reporting-System” beschrieben.
Für das System wurden acht Tests durchgeführt. Die Anzahl der Tests ergibt sich wie folgt:
$2\ Durchlaeufe\ *\ ((1\ Endpoint\ *\ 2\ Deploymentstrategien(Serverless\ und\ Microservices)\ *2\ Testarten))$
Für beide Anwendungen musste eine Konfiguration der Ressourcen vorgenommen werden. In Kubernetes wurden der Microservice-Anwendung 145 MB Memory und 0,07 CPU-Core bzw. 70 Milis für den Parameter ”requests” und 512 MB Memory und 0.,25 CPU-Core bzw. 250 Milis für den Parameter ”limits” zugewiesen. Der Serverless-Anwendung wurden 512 MB Memory zugewiesen.
Beim Testen wurden zwei verschiedene Testarten durchgeführt: incremental und triangle. Weitere Informationen zu den Testarten, z. B. wie viele VUs verwendet wurden, sind in Kapitel 4 und Kapitel 5.3 zu finden.
Die Events wurden vor dem Testen mit dem Event Mocker erstellt und während des Tests nicht mehr verändert. Für jede Anwendung wurden zwei Durchläufe durchgeführt. In jedem Durchlauf wurden beide Testarten ausgeführt. Zuerst wurde incremental für die Microservices-Deploymentstrategie und danach für die Serverless-Anwendung durchgeführt. Am Ende der zwei Tests wurde eine Pause gemacht, um sicherzustellen,

dass die AWS Lambda Funktionen kalt sind. Nach den incremental Tests wurden die triangle Tests gemacht, wobei zunächst die Microservices- und danach die Serverless-Deploymentstrategie getestet wurden.
Nach jedem Test wurden Daten gesammelt, diese wurden verarbeitet und werden im folgenden Kapitel als Ergebnisse vorgestellt.

6.5 Ergebnisse

In diesem Kapitel werden die Ergebnisse der Tests und der Kosten vorgestellt.
Nach der Durchführung der Tests wurden die Daten gesammelt und gefiltert. Aus den gefilterten Daten wurden Diagramme und Tabellen erstellt. Für jede Testart wurden zwei Diagramme und eine Tabelle erstellt. Weitere Informationen zu den Diagrammen und Tabellen und wie man sie liest, sind in Kapitel 5.4 ”Ergebnisse” zu finden.
Außer den Testergebnissen werden auch die Kosten der Deploymentstrategien dargestellt. Diese Kosten beziehen sich auf die Kosten von AWS und werden in Kapitel 6.6 erläutert.

6.5.1 Customer-Reports Incremental Test

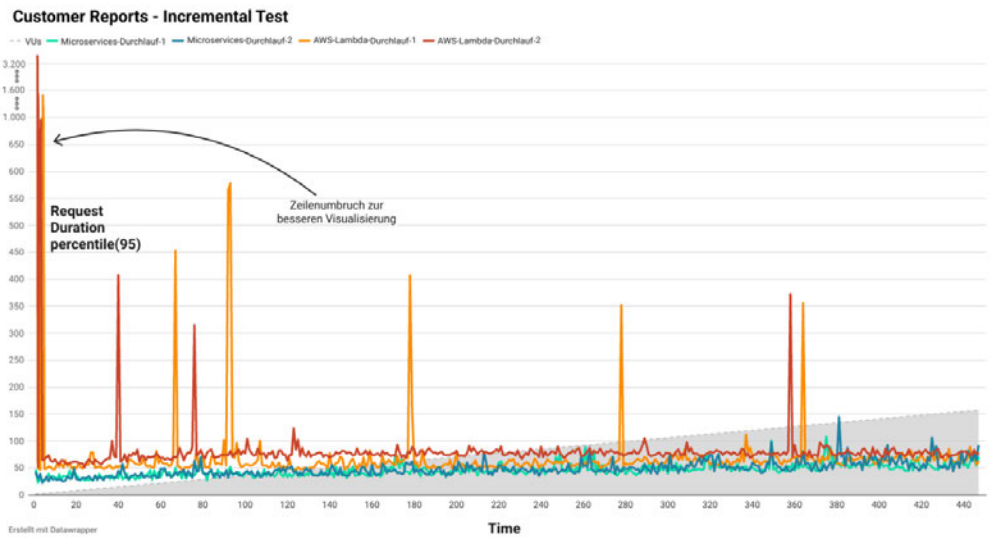

Abbildung 30: Microservices und AWS Lambda - Request-Duration als Percentile-95 über die Zeit x des ”POST /reports/<kvp>” Endpoints

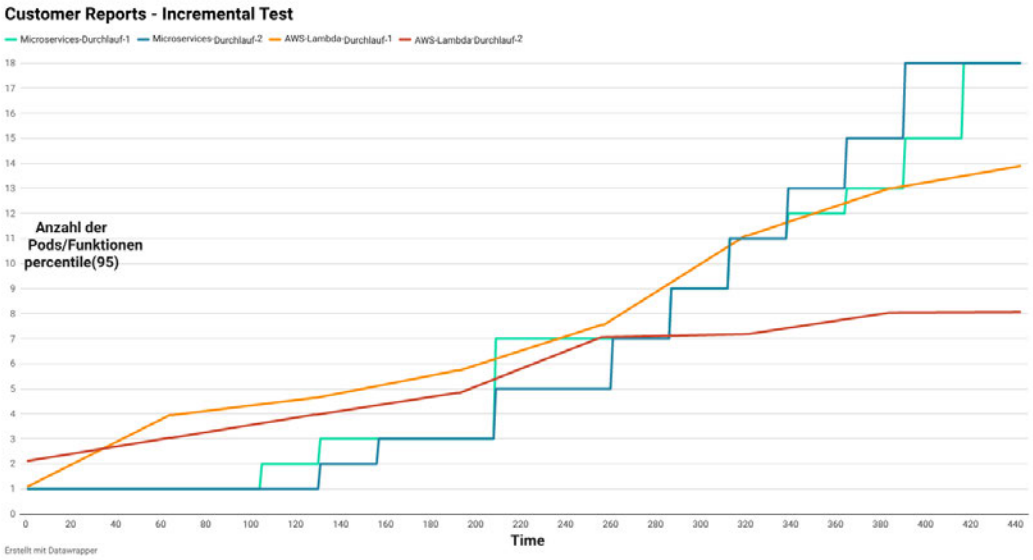

Abbildung 31: Microservices und AWS Lambda - Skalierbarkeit über die Zeit x des ”POST /reports/<kvp>” Endpoints

Incremental Tests	Min	Avg	Max	p(95)	Avg. RPS	suc req	unsuc req
Microservices Durchlauf 1	12.05	36.84	221.43	62.49	76.06	34024	282
AWS Lambda Durchlauf 1	34.39	51.62	3070	74	74.91	33787	0
Microservices Durchlauf 2	12.09	39.16	244.94	72.33	75.85	33909	306
AWS Lambda Durchlauf 2	35.24	59.48	3210	81.78	74.39	33554	0

Tabelle 10: Microservices und AWS Lambda - Ergebnisse der incremental Tests

Die Abbildungen und die Tabelle zeigen die Ergebnisse der vier incremental Tests. Zwei Tests wurden für die Serverless-Deploymentstrategie und zwei für die Microservices-Deploymentstrategie durchgeführt.

6.5.2 Customer-Reports Triangle Test

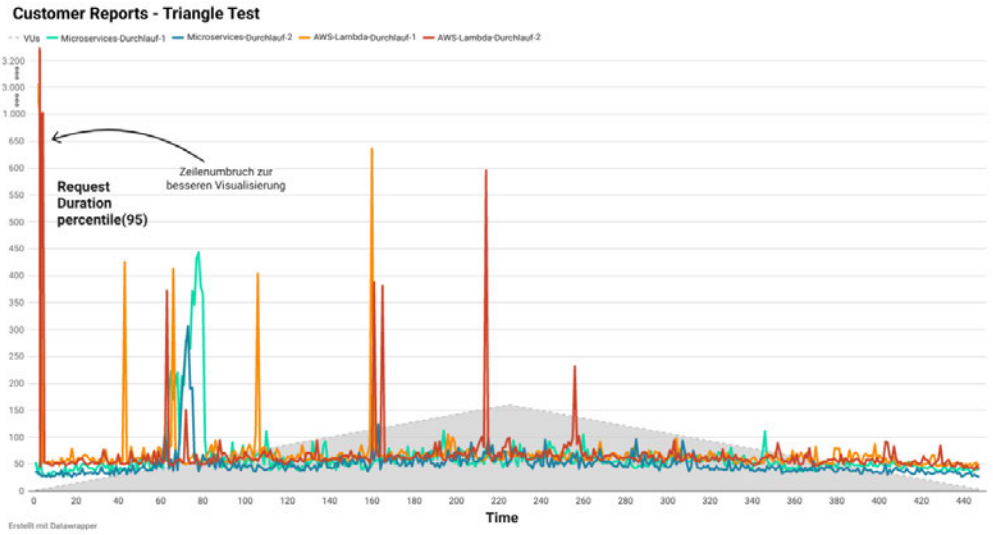

Abbildung 32: Microservices und AWS Lambda - Request-Duration als Percentile-95 über die Zeit x des ”POST /reports/<kvp>” Endpoints

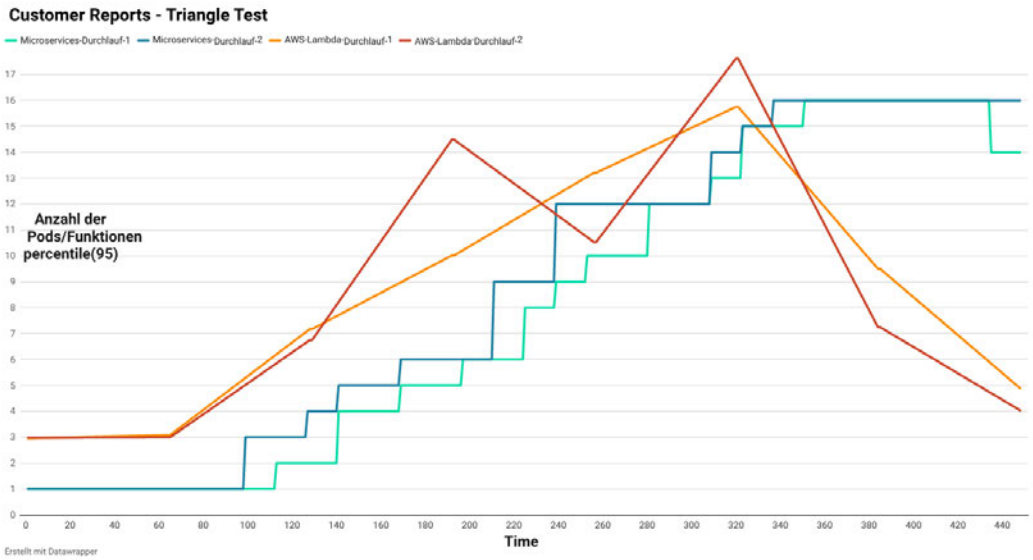

Abbildung 33: Microservices und AWS Lambda - Skalierbarkeit über die Zeit x des ”POST /reports/<kvp>” Endpoints

Triangle Tests	Min	Avg	Max	p(95)	Avg. RPS	suc req	unsuc req
Microservices Durchlauf 1	12.19	44.18	508.11	79.92	76.78	34287	311
AWS Lambda Durchlauf 1	29.49	53.28	3000	75.58	76.06	34282	0
Microservices Durchlauf 2	12.1	38.3	388.53	66.38	77.28	34492	303
AWS Lambda Durchlauf 2	34.63	53.59	3270	76.79	76.06	34265	0

Tabelle 11: Microservices und AWS Lambda - Ergebnisse der triangle Tests

Im Gegensatz zu den Abbildungen in Kapitel 6.5.1 zeigen die Diagramme und die Tabelle in diesem Kapitel die Ergebnisse der vier triangle Tests. Zwei Tests wurden für die Serverless-Deploymentstrategie und zwei für die Microservices-Deploymentstrategie durchgeführt.

6.5.3 Kosten

In diesem Kapitel werden die AWS-Kosten aufgeführt und beschrieben. Informationen über den AWS Pricing Calculator und die Anzahl der Request in der Kostenberechnung sowie die Begründung der Entscheidungen finden sich in Kapitel 5.4.4 ”Kosten”.
Für diese Berechnung wurden, wie auch in Kapitel 5.4.4, 2.700.000 Requests pro Monat berücksichtigt. In einer Customer-Reporting-Anwendung würde es nicht vorkommen, dass pro Monat 2.700.000 Requests getätigt werden. In der Realität würden weniger als 1.000 Requests pro Monat gemacht werden. Aber da die Ergebnisse der Kostenberechnung der Customer-Reporting-Anwendung mit den Ergebnissen aus Kapitel 5.4.4 und den Ergebnissen des Papers aus Kapitel 3.1 verglichen werden sollen, wird die Kostenberechnung dieser Anwendung mit 2.700.000 Requests pro Monat durchgeführt.
Die folgenden Services wurden in der Microservices-Deploymentstrategie verwendet:

- AWS ECR - die Customer-Reporting-Anwendung ist 110 MB groß. Es wird davon ausgegangen, dass sie viermal pro Woche deployt wird. Daraus ergeben sich 16 Deployments pro Monat, was bedeutet, dass 1.760 MB pro Monat in AWS ECR gespeichert werden.
- AWS Load Balancer (external) - pro Monat werden 2.700.000 Requests gesendet. Aus dem Load Testing ging hervor, dass bei einem Test 90 MB Daten an den Load-Balancer gesendet wurden. Ein Test besteht aus etwa 34.000 Requests. Das bedeutet, dass 7.200 MB pro Monat an den Load-Balancer gesendet werden.
- AWS EKS - fester Preis pro Monat.
- AWS EC2 - um sicherzustellen, dass es keinen Bottleneck im System gibt, wurden drei EC2-Instanzen des Typs ”r6.large” verwendet. Damit wird sichergestellt, dass die Pods skalieren können.
- AWS VPC

Region	Service	Monatlich	Jährlich	Währung
EU (Frankfurt)	Amazon Elastic Container Registry	0.1719	2.06	USD
EU (Frankfurt)	Application Load Balancer	19.94	239.28	USD
EU (Frankfurt)	Amazon EKS	73	876	USD
EU (Frankfurt)	Amazon Virtual Private Cloud (VPC)	0	0	USD
EU (Frankfurt)	Amazon EC2	220.95	2651.4	USD

Tabelle 12: Kosten des Microservices-Systems

Die folgenden Services wurden in der Serverless-Deploymentstrategie verwendet:

- AWS API Gateway - genau wie in Kapitel 5.5 ist dies ein REST-API-Endpoint. Pro Monat werden 2.700.000 Requests über diesen Endpoint gesendet.
- AWS Lambda - die Dauer eines Requests beträgt 51 ms. Der Preis hängt von der Dauer der Ausführung ab.
- AWS Load Balancer (intern) - genau wie bei dem Load-Balancer aus der Microservices-Deploymentstrategie, werden 7.200 MB Daten pro Monat über diesen Load-Balancer gesendet. Es handelt sich um einen internen Load-Balancer, was aber keinen Einfluss auf den Preis hat.
- AWS EKS - fester Preis pro Monat.
- AWS EC2 - auch hier werden drei Instanzen des EC2-Typs "r6.large" verwendet.
- AWS VPC

Region	Service	Monatlich	Jährlich	Währung
EU (Frankfurt)	Application Load Balancer	19.94	239.28	USD
EU (Frankfurt)	Amazon EKS	73	876	USD
EU (Frankfurt)	Amazon Virtual Private Cloud (VPC)	0	0	USD
EU (Frankfurt)	Amazon EC2	220.95	2651.4	USD
EU (Frankfurt)	Amazon API Gateway	10.06	120.72	USD
EU (Frankfurt)	AWS Lambda	1.7	20.4	USD

Tabelle 13: Kosten des Serverless-Systems

Bei dieser Kostenberechnung sind die in den beiden Tabellen aufgeführten Kosten nur eine Schätzung der Kosten und können von den tatsächlichen Kosten abweichen. Die aufgeführten Kosten wurden ohne AWS Free Tier erstellt.

6.6 Diskussion

In diesem Kapitel werden die Ergebnisse aus Kapitel 6.5 diskutiert. Außerdem wird versucht, die Ergebnisse der Diskussion aus Kapitel 5.5 wiederzugeben.
Die Abbildungen 31 und 33 zeigen den ersten Unterschied zu den Ergebnissen aus Kapitel 5.4. Es ist zu erkennen, dass die Pods der Microservice-Anwendung deutlich höher skaliert wurden als die Pods der Microservice-Anwendung aus dem Use Case "Employee-Time-Sheet-Management-Portals". Dies geschieht aus dem Grund, dass die Anwendungen API-Requests an KSQL senden. Nachdem sie die Antwort erhalten haben, werden die Daten in einer CSV-Datei gespeichert. Dies sorgt dafür, dass die Microservice-Anwendung eine Menge CPU-Ressourcen verbraucht und aus diesem Grund skaliert wird.

Aus den Abbildungen ist ersichtlich, dass bei der Microservices-Deploymentstrategie fast keine Ausschläge auftreten, außer in Abbildung 32. In dieser Abbildung ist bei den beiden Microservices-Durchläufen ein großer Ausschlag zu erkennen. Der Grund dafür ist, dass zu diesem Zeitpunkt nur ein Pod die Requests bearbeitet hat. Die Anwendung war nicht skaliert und verbrauchte zu viele CPU-Ressourcen. Im ersten Durchlauf verbrauchte der Pod während der Ausführung von 60 VUs etwa 243 Milis CPU. Dies entspricht einer Auslastung von 97,2 % dessen, was dem Pod zur Verfügung steht. Im zweiten Durchlauf wurden 206 Milis CPU verbraucht, was einer Auslastung von 82,4 % der Ressourcen entspricht. Dies führte dazu, dass die Requests langsamer verarbeitet wurden.
Dieser Fall zeigt auch, wie wichtig es ist, den Kubernetes-Autoscaler für jeden einzelnen Use Case individuell zu konfigurieren. Somit wird die in Kapitel 5.5 aufgestellte Behauptung bestätigt: **Die Skalierbarkeits- und Ressourcen-Konfiguration von Kubernetes ist aufwendiger als bei Serverless.**

Die Tabellen 14 und 15 zeigen den Verbrauch von CPU-Ressourcen durch die Pods. Es wurden aus folgendem Grund nicht alle Ergebnisse in die Tabellen aufgenommen. Im incremental Test wurden die Pods bei 158 VUs auf 18 Pods skaliert, was in einer Tabelle für Unordnung sorgen würde.
Die beiden Tabellen zeigen, dass der Traffic nicht gleichmäßig auf die Pods verteilt ist. In Tabelle 14 ist zu erkennen, dass Pod 4 bei 80 VUs etwa 55 % mehr CPU-Ressourcen als Pod 2 verbraucht. In Tabelle 15 sind ähnliche Ergebnisse zu sehen.
Hier wird die zweite Behauptung aus Kapitel 5.5 bestätigt: **Die Microservices -Deploymentstrategie leidet unter einem Load-Balancing- und Trafficverteilungs-Problem.**

Incremental Test							
VUs	**Pod 1**	**Pod 2**	**Pod 3**	**Pod 4**	**Pod 5**	**Pod 6**	**Pod 7**
1	2m	%	%	%	%	%	%
10	14m	%	%	%	%	%	%
20	77m	%	%	%	%	%	%
30	121m	%	%	%	%	%	%
40	89m	113m	%	%	%	%	%
50	77m	77m	76m	%	%	%	%
60	91m	92m	110m	%	%	%	%
70	103m	65m	72m	%	%	%	%
80	64m	51m	90m	113m	110m	72m	62m
90	80m	70m	54m	62m	59m	73m	62m
100	79m	101m	70m	79m	62m	72m	69m

Tabelle 14: CPU-Werte der Kubernetes Pods der Microservice-Anwendung beim Testen mit Incremental Tests

Incremental Test								
VUs	**Pod 1**	**Pod 2**	**Pod 3**	**Pod 4**	**Pod 5**	**Pod 6**	**Pod 7**	**Pod 8**
1	2m	%	%	%	%	%	%	%
10	6m	%	%	%	%	%	%	%
20	58m	%	%	%	%	%	%	%
30	58m	%	%	%	%	%	%	%
40	117m	%	%	%	%	%	%	%
50	168m	%	%	%	%	%	%	%
60	243m	%	%	%	%	%	%	%
70	200m	%	%	%	%	%	%	%
80	184m	173m	%	%	%	%	%	%
90	184m	173m	%	%	%	%	%	%
100	147m	106m	148m	137m	%	%	%	%
110	114m	123m	146m	121m	%	%	%	%
120	121m	123m	110m	116m	127m	%	%	%
130	121m	92m	110m	93m	93m	%	%	%
140	121m	102m	124m	91m	88m	102m	%	%
150	120m	85m	103m	91m	91m	102m	%	%
160	109m	102m	94m	100m	87m	99m	111m	108m

Tabelle 15: CPU-Werte der Kubernetes Pods der Microservice-Anwendung beim Testen mit triangle Tests

Es bestätigt sich auch eine weitere Erkenntnis aus der Diskussion in Kapitel 5.5: **Die Serverless-Deploymentstrategie leidet unter einem Problem des Cold-Starts.** Dies lässt sich anhand der Abbildungen 30 und 32 erkennen. Die Abbildungen zeigen, dass während der ersten Sekunde Ausschläge bei den Durchläufen des AWS Lambda zu sehen sind. Dies wird als Cold-Start der AWS Lambda Funktionen bezeichnet.

Die Tabellen zeigen, dass die Microservices-Deploymentstrategie bei allen Tests eine bessere Requestsbearbeitungsleistung aufweist als die Serverless-Deloymentstrategie. Dies ist nur bei einem Durchlauf eines Tests nicht der Fall. Für den triangle Test im ersten Durchlauf übertrifft die Serverless-Deploymentstrategie die Microservices-Deploymentstrategie bei der p(95)-Metrik. Dies ist jedoch nur ein kleiner Teil des Tests und in allen anderen Tests ist das Gegenteil zu beobachten.
Dadurch lässt sich bestätigen, dass die Microservices-Deploymentstrategie eine bessere Requestsbearbeitungsleistung als die Serverless-Deloymentstrategie aufweist.

In Kapitel 6.5.3 wurden die Ergebnisse der Kostenberechnung für die beiden Deploymentstrategien vorgestellt.
Es ist zu erkennen, dass der Setup der Microservices-Deploymentstrategie etwa 314 USD pro Monat bzw. 3.768,74 USD pro Jahr kostet. Das Setup der Serverless-Deploymentstrategie kostet 325,65 USD pro Monat bzw. 3.907,80 USD pro Jahr.
In diesem Beispiel ist zu erkennen, dass das Setup der Serverless-Deploymentstrategie mehr Geld als das Setup der Microservices-Deploymentstrategie kostet. Das Ergebnis ist das Gegenteil des Ergebnisses aus Kapitel 5.4.4. Genau wie in Kapitel 5.5, es kann nicht festgestellt werden, dass die Serverless- oder Microservices-Deploymentstrategie generell mehr kostet als die andere Deploymentstrategie. Wie in Kapitel 5.5 beschrieben, können diese Kosten reduziert werden. Um die genauen Kosten der beiden Deploymentstrategien zu vergleichen, müssen in Zukunft weitere Experimente durchgeführt werden.
Dennoch wird die Annahme bestätigt, dass die Serverless-Deploymentstrategie mit AWS Lambda weniger Kosten verursacht als eine Microservices -Deploymentstrategie mit EC2 für eine Anwendung mit geringerem Traffic.

Ein weiteres Ergebnis ist in Tabelle 10 und 11 zu sehen. Bei der Microservices -Deploymentstrategie waren etwa 300 Requests in jedem Test und jedem Durchlauf erfolglos. Es kann nicht festgestellt werden, warum dieses Ergebnis auftritt. Es gibt keine Daten über die erfolglosen Requests. Um diese Frage zu beantworten, müssen weitere

Tests durchgeführt werden, die sich genau an die erfolglosen Requests richten.

Die Erkenntnisse aus diesem Kapitel entsprechen den Erkenntnissen aus Kapitel 5.5, sodass auch die Empfehlungen mit denen aus Kapitel 5.5 übereinstimmen. Die Microservices-Deploymentstrategie eignet sich für Real-Time-Data-Processing-Anwendungen und Anwendungen, die auf Sekundenbruchteile angewiesen sind. Im Gegensatz dazu wird die Serverless-Deploymentstrategie für folgende Use Cases empfohlen: Anwendungen mit Traffic-Schwankungen, automatisch skalierbare APIs sowie Job-Scheduler (CronJobs). Die Kriterien für die Empfehlung sind die Kosten und die Performance (Skalierbarkeit, Resilience und Cloud Latency) der Deploymentstrategien. Dies ist nur ein kleiner Teil der Kriterien, die für die Entscheidung über eine Deploymentstrategie herangezogen werden. Bei der Auswahl der Deploymentstrategie sollten zudem auch andere Kriterien berücksichtigt werden, da diese Arbeit sich nur auf die zuvor genannten Kriterien bezieht.

7 Fazit

Ziel der Arbeit war es, zu untersuchen, wie sich die Performance und Kosten zwischen einer Serverless-Deplyomentstrategie mit AWS Lambda und einer Microservices -Deploymentstrategie mit Docker und Kubernetes unterscheiden. In dieser Arbeit wurde die Performance durch Skalierbarkeit, Resilience und Cloud Latency definiert.
Anhand der Ergebnisse dieser Arbeit können Empfehlungen für Use Cases gegeben werden, in denen eine Deploymentstrategie der anderen vorgezogen werden kann.
Zwei Use Cases wurden zur Bewertung der Performance und Kosten herangezogen: "Employee-Time-Sheet-Managemen-Portal" und "Customer-Reporting-System". Für diese Systeme wurden Performance-Tests durchgeführt, um die Performance der Deploymentstrategien zu messen. Zur Berechnung der Kosten wurde der AWS Pricing Calculator verwendet.
Die Experimente führten zu folgenden Ergebnissen:

- Microservices-Deploymentstrategie leidet unter einem Load-Balancing- und Trafficverteilungs-Problem.
- Kubernetes-Skalierbarkeits- und Ressourcen-Konfiguration ist aufwendiger als die von Serverless.
- Microservices-Deploymentstrategie hat eine bessere Requestsbearbeitungsleistung als die Serverless-Deploymentstrategie.
- Serverless-Deploymentstrategie leidet unter dem Problem des Cold-Starts.
- Serverless-Deploymentstrategie ist agiler in Bezug auf Skalierbarkeit als die Microservices-Deploymentstrategie.
- Serverless-Deploymentstrategie, die AWS Lambda verwendet, für eine Anwendung mit geringerem Traffic, verursacht weniger Kosten als eine Microservices -Deploymentstrategie, die EC2 verwendet.

Aus diesen Erkenntnissen lassen sich Empfehlungen für Use Cases ableiten, in denen eine Deploymentestrategien besser geeignet ist als die andere.
Die Microservices-Deploymentstrategie eignet sich für folgende Use Cases:

- Real-Time-Data-Processing
- Anwendungen, die auf Sekundenbruchteile angewiesen sind

Im Gegensatz dazu eignet sich die Serverless-Deploymentstrategie für die folgenden Use Cases:

- Anwendungen mit Traffic-Schwankungen
- Automatisch skalierbare APIs
- Job-Scheduler (CronJobs)

Die Kriterien für die Empfehlungen wurden aus dieser Arbeit genommen. Dies sind die Kosten und die Performance (Skalierbarkeit, Resilience und Cloud Latency) der Deploymentstrategien, welche nur einen kleinen Teil der Kriterien darstellen, die bei der Entscheidung für eine Deploymentstrategie herangezogen werden sollten. Auch andere Kriterien sollten bei der Auswahl einer Deploymentstrategie berücksichtigt werden.
Aus den Ergebnissen der Arbeit geht hervor, dass nicht jede Deploymentstrategie für jeden Use Case geeignet ist. Bei der Auswahl einer Deploymentstrategie sollten die Use Cases berücksichtigt werden. Dabei muss sich folgende Frage gestellt werden: "Welche Anforderungen sind für das System relevant?" Anhand der Anforderungen können die Kriterien für die Deploymentstrategie erstellt werden. Anhand der Kriterien kann eine Auswahl der richtigen Deploymentstrategie für den Use Case getroffen werden.

7.1 Ausblick

Aus den Ergebnissen der Arbeit haben sich weitere Fragen ergeben, diese sollten durch weitere Arbeit und Forschung beantwortet werden.
In der Arbeit wurde festgestellt, dass die Serverless-Deploymentstrategie, die AWS Lambda verwendet, für eine Anwendung mit geringerem Traffic, weniger Kosten verursacht als eine Microservices-Deploymentstrategie, die EC2 verwendet. Es stellt sich jedoch die Frage, welche der beiden Deploymentstrategien weniger Kosten verursacht, wenn die Anwendung viel Traffic gesendet bekommt. In weiteren Experimenten kann diese Frage beantwortet werden - dazu muss ein Fokus auf die Kostenberechnung gelegt werden.
Eine weitere Unklarheit sind die erfolglosen Requests auf das Customer-Report-System, die in Kapitel 6.6 diskutiert wurden. Es ist nicht möglich festzustellen, warum dies geschieht. Es liegen keine Daten über die erfolglosen Requests vor. Um diese Frage zu beantworten, müssen weitere Tests mit genauem Fokus auf die erfolglosen Requests durchgeführt werden.
Eine Eigenschaft von Deploymentstrategien, die genauer untersucht werden sollte, ist

die Resilience. Um die Resilience genauer zu betrachten, kann eine Skalierbarkeitsgrenze für die Anzahl der Pods und Funktionen gesetzt werden. Das bedeutet, dass die Deploymentstrategien nur bis zu einer bestimmten Anzahl von Requests skalieren können. Dieses Experiment würde Auskunft darüber geben, welche Deploymentstrategie resilienter ist. Da dieses Experiment in dieser Arbeit nicht durchgeführt wurde, sollten weitere Arbeiten diesen Aufbau und die Frage genauer untersuchen.

Literatur

[AVSTK18] Abdollahi Vayghan, Leila; Saied, Mohamed A.; Toeroe, Maria ; Khendek, Ferhat: Deploying Microservice Based Applications with Kubernetes: Experiments and Lessons Learned. DOI 10.1109/CLOUD.2018.00148. In: *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*, 2018 S. 970–973

[AWS21a] AWS: *Lambda function scaling - AWS Lambda*. https://docs.aws.amazon.com/lambda/latest/dg/invocation-scaling.html. Version: 09 2021. – zuletzt zugegriffen am 14.12.2021

[AWS21b] AWS: *What is Amazon EC2? - Amazon Elastic Compute Cloud*. https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/concepts.html. Version: 05 2021. – zuletzt zugegriffen am 25.12.2021

[AWS21c] AWS: *What is AWS Pricing Calculator? - AWS Pricing Calculator*. https://docs.aws.amazon.com/pricing-calculator/latest/userguide/what-is-pricing-calculator.html. Version: 04 2021. – zuletzt zugegriffen am 25.12.2021

[AWSoJa] AWS: *Amazon API Gateway | API Management | Amazon Web Services*. https://aws.amazon.com/api-gateway/. Version: o.J. – zuletzt zugegriffen am 30.10.2021

[AWSoJb] AWS: *Amazon CloudWatch - Application and Infrastructure Monitoring*. https://aws.amazon.com/cloudwatch/. Version: o.J. – zuletzt zugegriffen am 25.12.2021

[AWSoJc] AWS: *Elastic Load Balancing*. https://aws.amazon.com/elasticloadbalancing/. Version: o.J. – zuletzt zugegriffen am 30.10.2021

[AWSoJd] AWS: *Free Cloud Computing Services - AWS Free Tier*. https://aws.amazon.com/free/. Version: o.J. – zuletzt zugegriffen am 25.12.2021

[AWSoJe] AWS: *Fully Managed Container Registry – Amazon Elastic Container Registry – Amazon Web Services*. https://aws.amazon.com/ecr/. Version: o.J. – zuletzt zugegriffen am 30.10.2021

[AWSoJf] AWS: *Fully Managed Container Solution – Amazon Elastic Container Service (Amazon ECS) - Amazon Web Services*. https://aws.amazon.com/ecs/. Version: o.J. – zuletzt zugegriffen am 25.12.2021

[AWSoJg] AWS: *Introduction to Amazon Aurora - Relational Database Built for the Cloud*. https://aws.amazon.com/rds/aurora/. Version: o.J. – zuletzt zugegriffen am 25.12.2021

[AWSoJh] AWS: *Logically Isolated Virtual Private Cloud—Amazon VPC – Amazon Web Services*. https://aws.amazon.com/vpc/. Version: o.J. – zuletzt zugegriffen am 30.10.2021

[AWSoJi] AWS: *Managed Kubernetes Service – Amazon EKS – Amazon Web Services*. https://aws.amazon.com/eks/. Version: o.J. – zuletzt zugegriffen am 30.10.2021

[AWSoJj] AWS: *What is AWS*. https://aws.amazon.com/what-is-aws/. Version: o.J. – zuletzt zugegriffen am 25.12.2021

[AzuoJa] Azure, Microsoft: *What Is a Virtual Machine and How Does It Work*. https://azure.microsoft.com/en-us/overview/what-is-a-virtual-machine/#overview. Version: o.J. – zuletzt zugegriffen am 25.12.2021

[AzuoJb] Azure, Microsoft: *What is Virtualization - Definition*. https://azure.microsoft.com/en-us/overview/what-is-virtualization/. Version: o.J. – zuletzt zugegriffen am 25.12.2021

[Bes21] Beswick, James: *Operating Lambda: Performance optimization – Part 1*. https://aws.amazon.com/blogs/compute/operating-lambda-performance-optimization-part-1/. Version: 04 2021. – zuletzt zugegriffen am 23.11.2021

[BGO+16] Burns, Brendan; Grant, Brian; Oppenheimer, David; Brewer, Eric ; Wilkes, John: Borg, Omega, and Kubernetes: Lessons Learned from Three Container-Management Systems over a Decade. DOI 10.1145/2898442.2898444. In: *Queue* 14 (2016), jan, Nr. 1, S. 70–93. ISSN 1542–7730

[Con21] Confluent Inc.: *KSQL versus ksqlDB - ksqlDB Documentation.* https://docs.ksqldb.io/en/latest/operate-and-deploy/ksql-vs-ksqldb/. Version: 01 2021. – zuletzt zugegriffen am 28.12.2021

[CP17] Casalicchio, Emiliano; Perciballi, Vanessa: Auto-Scaling of Containers: The Impact of Relative and Absolute Metrics. DOI 10.1109/FAS-W.2017.149. In: *2017 IEEE 2nd International Workshops on Foundations and Applications of Self* Systems (FAS*W)*, 2017 S. 207–214

[Dal21] Daly, Jeremy: *GitHub - serverless-mysql: A module for managing MySQL connections at SERVERLESS scale.* https://github.com/jeremydaly/serverless-mysql. Version: 10 2021. – zuletzt zugegriffen am 12.10.2021

[Doc21] Docker: *Overview of Docker Compose.* https://docs.docker.com/compose/. Version: 05 2021. – zuletzt zugegriffen am 25.12.2021

[DocoJa] Docker: *Docker overview.* https://docs.docker.com/get-started/overview/. Version: o.J. – zuletzt zugegriffen am 14.12.2021

[DocoJb] Docker: *What is a Container?* https://www.docker.com/resources/what-container. Version: o.J. – zuletzt zugegriffen am 14.12.2021

[FJG20] Fan, Chen-Fu; Jindal, Anshul ; Gerndt, Michael: Microservices vs Serverless: A Performance Comparison on a Cloud-native Web Application. DOI 10.5220/0009792702040215, 2020 S. 204–215

[Fow19] Fowler, Martin: *Microservices Guide.* https://martinfowler.com/microservices/. Version: 08 2019. – zuletzt zugegriffen am 14.09.2021

[GaroJ] Gartner: *Definition of Latency - Gartner Information Technology Glossary.* https://www.gartner.com/en/information-technology/glossary/latency. Version: o.J. – zuletzt zugegriffen am 25.12.2021

[Glu20] Gluck, Adam: *Introducing Domain-Oriented Microservice Architecture.* https://eng.uber.com/microservice-architecture/. Version: 07 2020. – zuletzt zugegriffen am 14.09.2021

[Gon19] Gonchar, Grygoriy: *Microservices and Kafka — Part 1 - The (eBay Kleinanzeigen and mobile.de) Tech Blog*. https://ebaytech.berlin/microservices-and-kafka-part-1-614767d27b20. Version: 10 2019. – zuletzt zugegriffen am 14.09.2021

[Gra21] Grafana: *GitHub - k6: A modern load testing tool, using Go and JavaScript - https://k6.io*. https://github.com/grafana/k6. Version: 09 2021. – zuletzt zugegriffen am 16.09.2021

[GraoJ] Grafana: *Introduction to Grafana*. https://grafana.com/docs/grafana/latest/introduction/. Version: o.J. – zuletzt zugegriffen am 25.12.2021

[Her15] Herzog, Jared: Software Architecture in Practice Third Edition Written by Len Bass, Paul Clements, Rick Kazman. DOI 10.1145/2693208.2693252. In: *SIGSOFT Softw. Eng. Notes* 40 (2015), feb, Nr. 1, S. 51–52. ISSN 0163–5948

[Inc21a] Inc., Confluent: Building Stream Processing Applications with Confluent. https://www.confluent.io/white-paper/stream-processing-made-easy-confluent-cloud-ksql/. (2021), S. 9

[Inc21b] Inc., Confluent: *Docker Hub - CP KSQL Server*. https://hub.docker.com/r/confluentinc/cp-ksql-server. Version: 12 2021. – zuletzt zugegriffen am 28.12.2021

[Inc21c] Inc., Confluent: *Docker Hub - CP KSQLDB Server*. https://hub.docker.com/r/confluentinc/cp-ksqldb-server. Version: 12 2021. – zuletzt zugegriffen am 28.12.2021

[Inc21d] Inc., Confluent: *Docker Hub - KSQLDB Server*. https://hub.docker.com/r/confluentinc/ksqldb-server. Version: 12 2021. – zuletzt zugegriffen am 28.12.2021

[Inc21e] Inc., Confluent: *GitHub - confluentinc/ksql: The database purpose-built for stream processing applications*. https://github.com/confluentinc/ksql. Version: 12 2021. – zuletzt zugegriffen am 28.12.2021

[Inc21f] Inc., Confluent: *KSQL | Confluent Platform 5.3.0*. https://docs.confluent.io/5.3.0/ksql/docs/index.html. Version: 12 2021. – zuletzt zugegriffen am 17.12.2021

[IncoJ] Inc., Confluent: *Formats, Serializers, and Deserializers | Confluent Documentation*. https://docs.confluent.io/platform/current/schema-registry/serdes-develop/index.html#limitations. Version: o.J. – zuletzt zugegriffen am 23.11.2021

[Inf21] Influxdata: *GitHub - influxdb: Scalable datastore for metrics, events, and real-time analytics*. https://github.com/influxdata/influxdb. Version: 11 2021. – zuletzt zugegriffen am 25.12.2021

[ISToJa] ISTQB: *ISTQB Glossary*. https://glossary.istqb.org/en/search/performance%20testing. Version: o.J. – zuletzt zugegriffen am 16.09.2021

[ISToJb] ISTQB: *ISTQB Glossary*. https://glossary.istqb.org/en/search/load%20testing. Version: o.J. – zuletzt zugegriffen am 16.09.2021

[JSSS+19] Jonas, Eric; Schleier-Smith, Johann; Sreekanti, Vikram; Tsai, Chia-Che; Khandelwal, Anurag; Pu, Qifan; Shankar, Vaishaal; Carreira, Joao; Krauth, Karl; Yadwadkar, Neeraja; Gonzalez, Joseph E.; Popa, Raluca A.; Stoica, Ion ; Patterson, David A.: *Cloud Programming Simplified: A Berkeley View on Serverless Computing*. 2019

[JY18] Jambunathan, Baskaran; Yoganathan, Kalpana: Architecture Decision on using Microservices or Serverless Functions with Containers. DOI 10.1109/ICCTCT.2018.8551035. In: *2018 International Conference on Current Trends towards Converging Technologies (ICCTCT)*, 2018 S. 1–7

[KafoJ] Kafka, Apache: *Introduction - Everything you need to know about Kafka in 10 minutes*. https://kafka.apache.org/intro. Version: o.J. – zuletzt zugegriffen am 17.12.2021

[Kha19] Khaku, Ammar: *How Netflix microservices tackle dataset pub-sub - Netflix TechBlog*. https://netflixtechblog.com/how-netflix-microservices-tackle-dataset-pub-sub-4a068adcc9a. Version: 10 2019. – zuletzt zugegriffen am 14.09.2021

[Kho21] Khononov, V.: *Learning Domain-Driven Design: Aligning Software Architecture and Business Strategy*. O'Reilly Media, Incorporated https://books.google.de/books?id=jSVhzgEACAAJ. ISBN 9781098100131

[Kol17] Kolesnikov, Dmitry: *Using Microservices to Power Fashion Search and Discovery*. https://engineering.zalando.com/posts/2017/02/using-microservices-to-power-fashion-search-and-discovery.html. Version: 02 2017. – zuletzt zugegriffen am 14.09.2021

[Kre19] Kreps, J.K: *Introducing ksqlDB*. https://www.confluent.io/blog/intro-to-ksqldb-sql-database-streaming/. Version: 11 2019. – zuletzt zugegriffen am 28.12.2021

[Kub21a] Kubernetes: *Autoscaler/cluster-autoscaler · Kubernetes/autoscaler*. https://github.com/kubernetes/autoscaler/tree/master/cluster-autoscaler. Version: 12 2021. – zuletzt zugegriffen am 14.12.2021

[Kub21b] Kubernetes: *Autoscaler/vertical-pod-autoscaler · Kubernetes/autoscaler*. https://github.com/kubernetes/autoscaler/tree/master/vertical-pod-autoscaler. Version: 03 2021. – zuletzt zugegriffen am 14.12.2021

[Kub21c] Kubernetes: *Horizontal Pod Autoscaling*. https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/. Version: 12 2021. – zuletzt zugegriffen am 14.12.2021

[Kub21d] Kubernetes: *Kubernetes Components*. https://kubernetes.io/docs/concepts/overview/components/. Version: 10 2021. – zuletzt zugegriffen am 14.12.2021

[Kub21e] Kubernetes: *Overview of kubectl*. https://kubernetes.io/docs/reference/kubectl/overview/. Version: 11 2021. – zuletzt zugegriffen am 25.12.2021

[Kub21f] Kubernetes: *The Kubernetes API*. https://kubernetes.io/docs/concepts/overview/kubernetes-api/. Version: 11 2021. – zuletzt zugegriffen am 14.12.2021

[Kub21g] Kubernetes: *Understanding Kubernetes Objects*. https://kubernetes.io/docs/concepts/overview/working-with-objects/kubernetes-objects/. Version: 08 2021. – zuletzt zugegriffen am 14.12.2021

[Kub21h] Kubernetes: *What is Kubernetes?* https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/. Version: 07 2021. – zuletzt zugegriffen am 14.12.2021

[LF14] Lewis, James; Fowler, Martin: *Microservices.* https://martinfowler.com/articles/microservices.html. Version: 03 2014. – zuletzt zugegriffen am 14.09.2021

[LRC+18] Lloyd, Wes; Ramesh, Shruti; Chinthalapati, Swetha; Ly, Lan ; Pallickara, Shrideep: Serverless Computing: An Investigation of Factors Influencing Microservice Performance. DOI 10.1109/IC2E.2018.00039. In: *2018 IEEE International Conference on Cloud Engineering (IC2E)*, 2018 S. 159–169

[LSB17] Leung, Andrew; Spyker, Andrew ; Bozarth, Tim: Titus: Introducing Containers to the Netflix Cloud: Approaching Container Adoption in an Already Cloud-Native Infrastructure. DOI 10.1145/3155112.3158370. In: *Queue* 15 (2017), oct, Nr. 5, S. 53–77. ISSN 1542–7730

[LSF18] Lee, Hyungro; Satyam, Kumar ; Fox, Geoffrey: Evaluation of Production Serverless Computing Environments. DOI 10.1109/CLOUD.2018.00062. In: *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*, 2018 S. 442–450

[MB17] McGrath, Garrett; Brenner, Paul R.: Serverless Computing: Design, Implementation, and Performance. DOI 10.1109/ICDCSW.2017.36. In: *2017 IEEE 37th International Conference on Distributed Computing Systems Workshops (ICDCSW)*, 2017 S. 405–410

[Mic21] Microsoft: *Monoliths to microservices using domain-driven design - Azure Architecture Center.* https://docs.microsoft.com/en-us/azure/architecture/microservices/migrate-monolith. Version: 11 2021. – zuletzt zugegriffen am 25.12.2021

[MMK17] Murray, Acklyn; Mejias, M. ; Keiller, P.: Resilience Methods within the Software Development Cycle, 2017

[Mol09] Molyneaux, Ian: *The Art of Application Performance Testing: Help for Programmers and Quality Assurance.* 1st. O'Reilly Media, Inc., 2009. ISBN 0596520662

[Mys21] MYSQLJS: *GitHub - mysql: A pure node.js JavaScript Client implementing the MySQL protocol.* https://github.com/mysqljs/mysql. Version: 10 2021. – zuletzt zugegriffen am 12.10.2021

[New15] NEWMAN, Sam: *Building Microservices: Designing Fine-Grained Systems.* 1st. O'Reilly Media, 2015. – 280 S. ISBN 978–1491950357

[NSP17] NARKHEDE, Neha; SHAPIRA, Gwen ; PALINO, Todd: *Kafka: The Definitive Guide Real-Time Data and Stream Processing at Scale.* 1st. O'Reilly Media, Inc., 2017. ISBN 1491936169

[NT20] NUPPONEN, Jussi; TAIBI, Davide: Serverless: What it Is, What to Do and What Not to Do. DOI 10.1109/ICSA-C50368.2020.00016. In: *2020 IEEE International Conference on Software Architecture Companion (ICSA-C)*, 2020 S. 49–50

[OBKN09] OREN BEN-KIKI, Clark E.; NET, Ingy döt: *YAML Ain't Markup Language (YAML^{TM}) Version 1.2.* https://yaml.org/spec/1.2.1/. Version: 10 2009. – zuletzt zugegriffen am 25.12.2021

[OMG17] OMG: *OMG Unified Modeling Language (OMG UML), Version 2.5.1.* formal/17-12-05. Needham, MA: Object Management Group, Dezember 2017. https://www.omg.org/spec/UML/2.5.1/

[Pal21] PALLETS: *GitHub - flask: The Python micro framework for building web applications.* https://github.com/pallets/flask. Version: 09 2021. – zuletzt zugegriffen am 25.12.2021

[RH20] RED HAT, Inc.: *What is a REST API?* https://www.redhat.com/en/topics/api/what-is-a-rest-api. Version: 05 2020. – zuletzt zugegriffen am 25.12.2021

[Sba17] SBARSKI, P.: *Serverless Architectures on AWS: With examples using AWS Lambda.* Manning Publications https://books.google.de/books?id=TfsFvgAACAAJ. ISBN 9781617293825

[Ser21] SERVERLESS: *GitHub - serverless: Serverless Framework – Build web, mobile and IoT applications with serverless architectures using AWS Lambda, Azure Functions, Google CloudFunctions more!* https://github.com/serverless/serverless. Version: 10 2021. – zuletzt zugegriffen am 14.12.2021

[Tec15] TECHOPEDIA: *Network Traffic*. https://www.techopedia.com/definition/29917/network-traffic. Version: 04 2015. – zuletzt zugegriffen am 25.12.2021

[Tec17] TECHOPEDIA: *Scalability*. https://www.techopedia.com/definition/9269/scalability. Version: 01 2017. – zuletzt zugegriffen am 25.12.2021

[Tec21] TECHOPEDIA: *Structured Query Language (SQL)*. https://www.techopedia.com/definition/1245/structured-query-language-sql. Version: 01 2021. – zuletzt zugegriffen am 25.12.2021

[VisoJ] VISUALISING INFORMATION FOR ADVOCACY: *Datawrapper*. https://visualisingadvocacy.org/node/673.html. Version: o.J. – zuletzt zugegriffen am 25.12.2021

Abbildungsverzeichnis

Tabellenverzeichnis

Anhang

7.2 Sequenz Diagramme aus dem Kapitel 5.2.2 ”Laufzeitsicht”

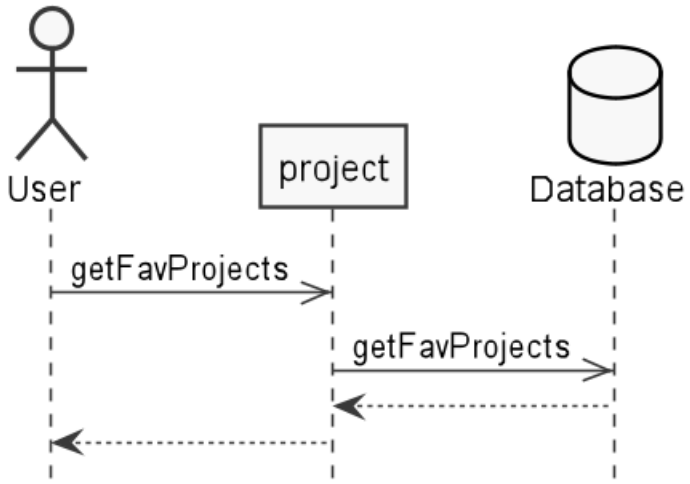

Abbildung 34: Sequenzdiagramm des getFavProjects Endpoints in der Microservice-Anwendung, UML nach [OMG17]

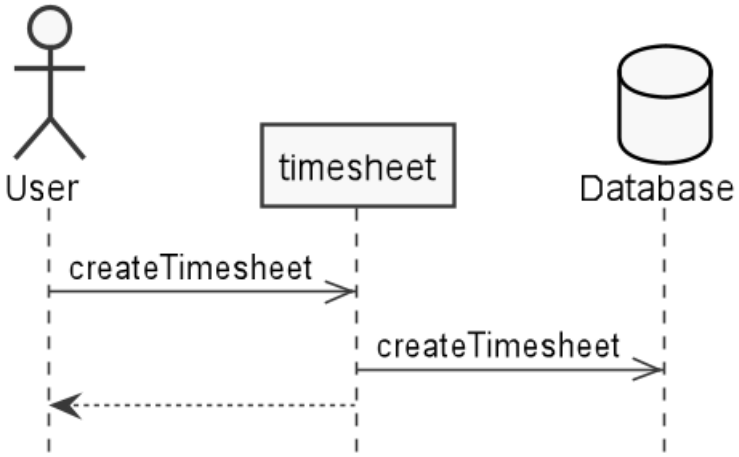

Abbildung 35: Sequenzdiagramm des createTimesheet Endpoints in der Microservice-Anwendung, UML nach [OMG17]

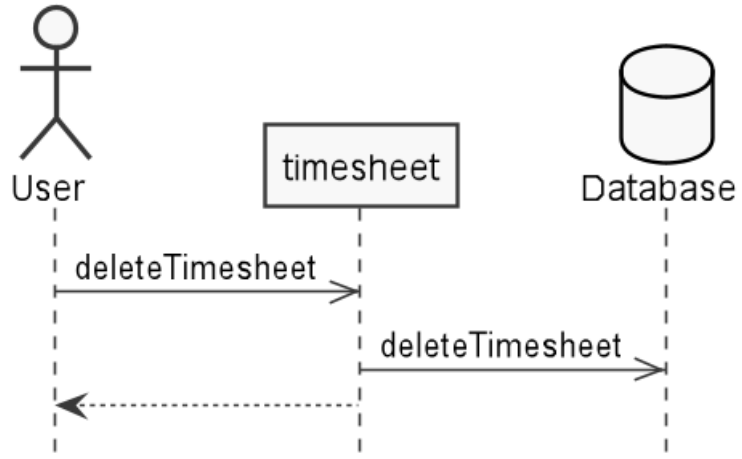

Abbildung 36: Sequenzdiagramm des deleteTimesheet Endpoints in der Microservice-Anwendung, UML nach [OMG17]

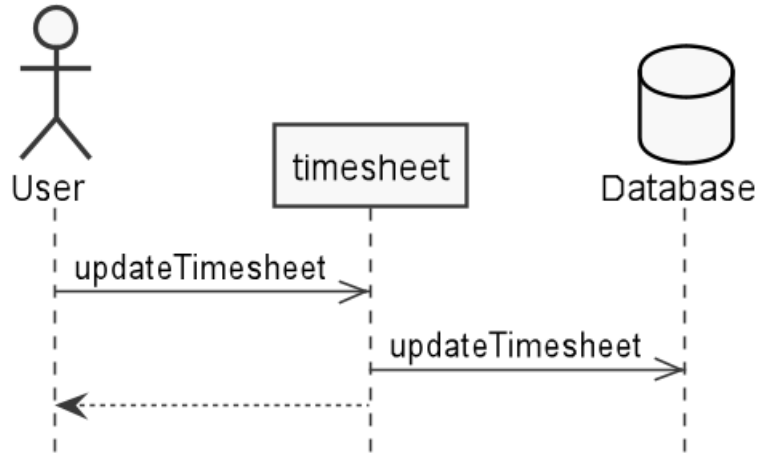

Abbildung 37: Sequenzdiagramm des updateTimesheet Endpoints in der Microservice-Anwendung, UML nach [OMG17]

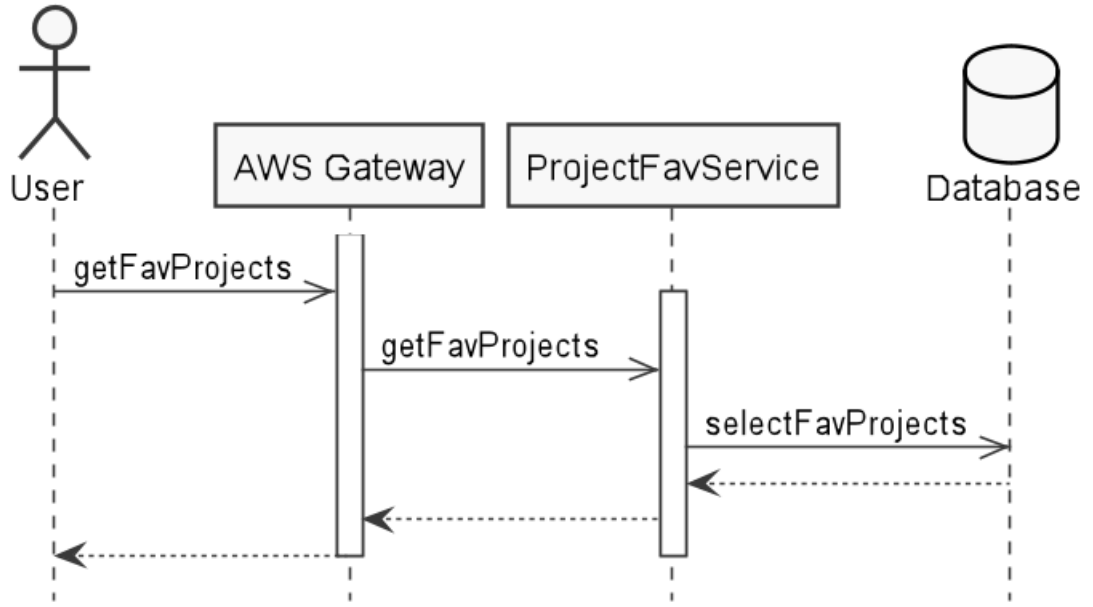

Abbildung 38: Sequenzdiagramm des getFavProjects Endpoints in der Serverless-Anwendung, UML nach [OMG17]

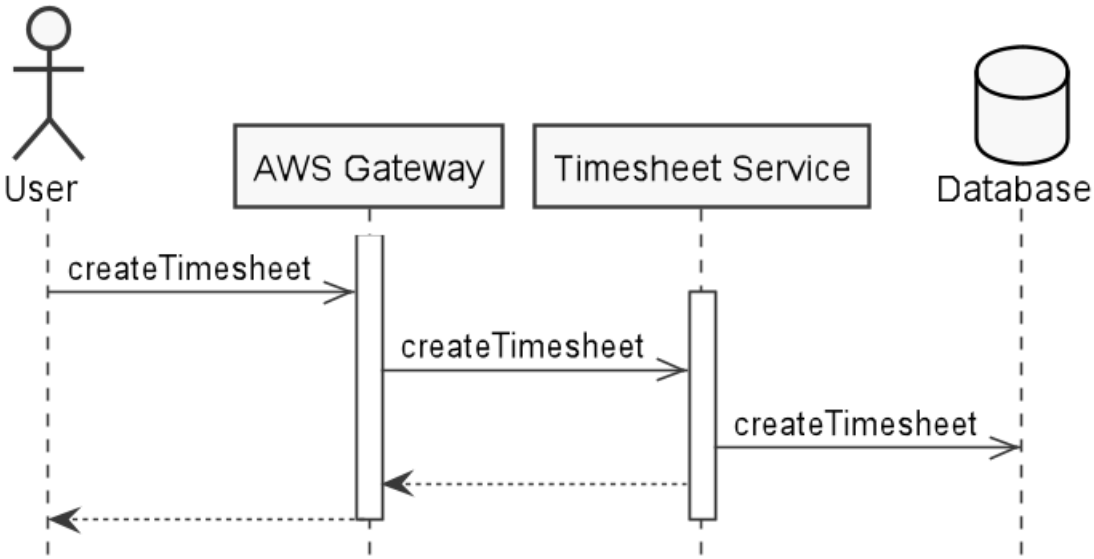

Abbildung 39: Sequenzdiagramm des createTimesheet Endpoints in der Serverless-Anwendung, UML nach [OMG17]

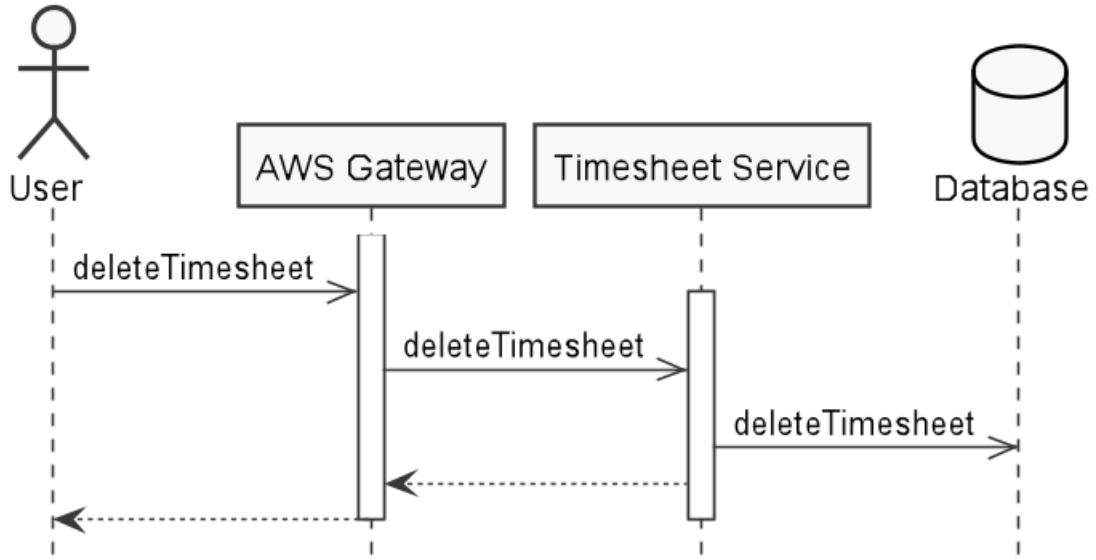

Abbildung 40: Sequenzdiagramm des deleteTimesheet Endpoints in der Serverless-Anwendung, UML nach [OMG17]

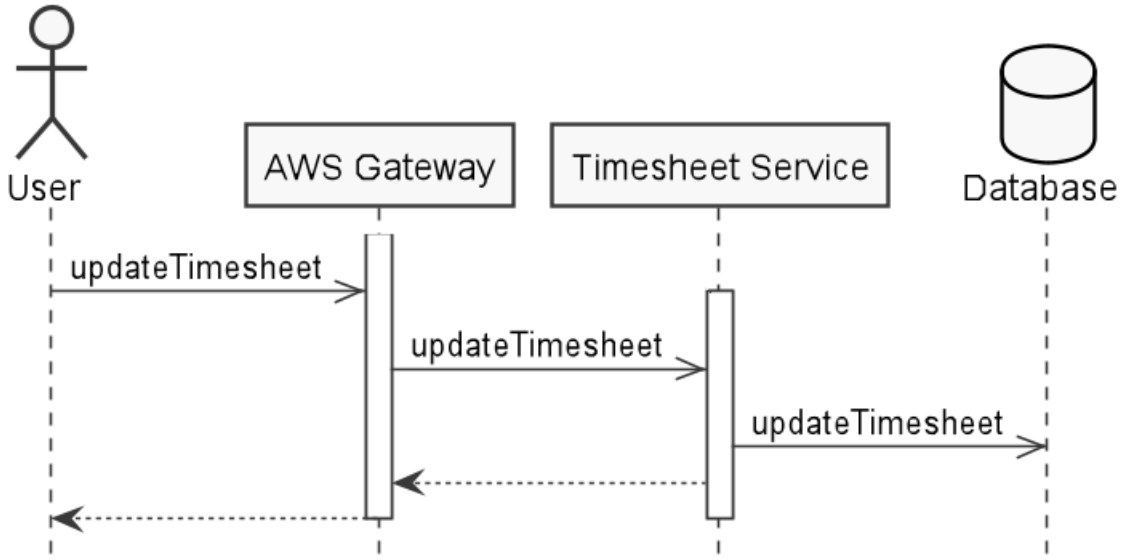

Abbildung 41: Sequenzdiagramm des updateTimesheet Endpoints in der Serverless-Anwendung, UML nach [OMG17]

7.3 Ergebnisse der Tests aus dem Kapitel 5.4.2 ”Geringe CPU- und Memory-Werte”

Incremental Tests		Min	Avg	Max	p(95)	Avg. RPS	suc req	unsuc req
Read Favourite Projects	Microservices	6.18	24.34	398.69	51.41	76.96	34715	0
	AWS Lambda	14.95	49.01	2300	84.66	75.15	33896	0
Create Timesheet	Microservices	0.98	36.56	721.79	87.68	76.06	34301	5
	AWS Lambda	28.9	46.15	2003	65.5	75.39	34006	0
Delete Timesheet	Microservices	16.32	49.41	691.34	125.09	74.87	33791	0
	AWS Lambda	28.61	46.39	2140	69.52	75.28	33956	0

Tabelle 16: Microservices und AWS Lambda - Ergebnisse der incremental Tests

Triangle Tests		Min	Avg	Max	p(95)	Avg. RPS	suc req	unsuc req
Read Favourite Projects	Microservices	13.83	28.02	448.69	80.77	77.97	35141	2
	AWS Lambda	24.82	44.15	2110	64.24	76.83	34612	0
Create Timesheet	Microservices	14.51	129.48	1100	496.7	71.05	31984	6
	AWS Lambda	29.42	62.39	2210	137.4	75.48	33998	0
Delete Timesheet	Microservices	16.49	40.4	707.2	105.46	76.95	34683	3
	AWS Lambda	28.8	47.34	2810	76.05	76.52	34450	2

Tabelle 17: Microservices und AWS Lambda - Ergebnisse der triangle Tests

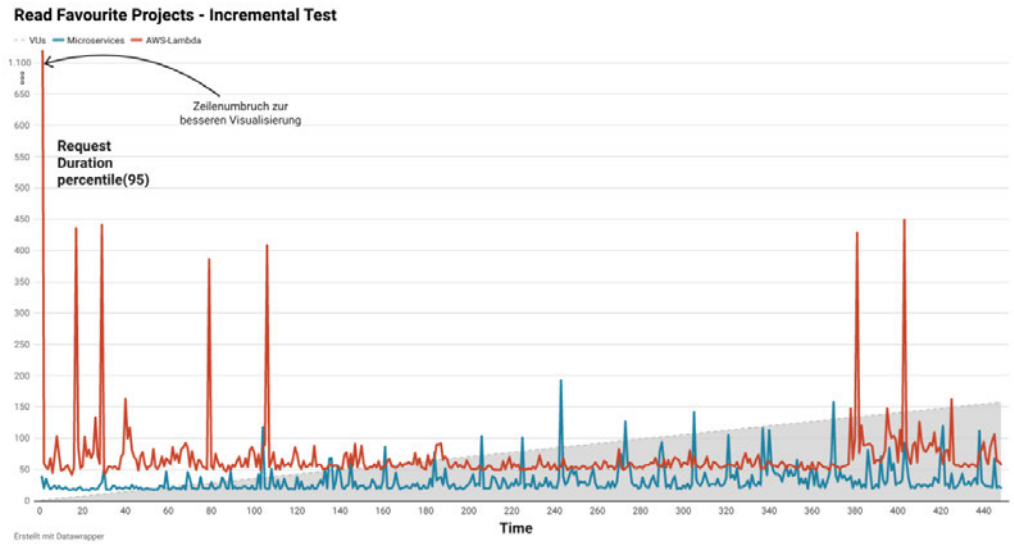

Abbildung 42: Microservices und AWS Lambda - Request-Duration als Percentile-95 über die Zeit x des ”POST /projects/<id>” Endpoints

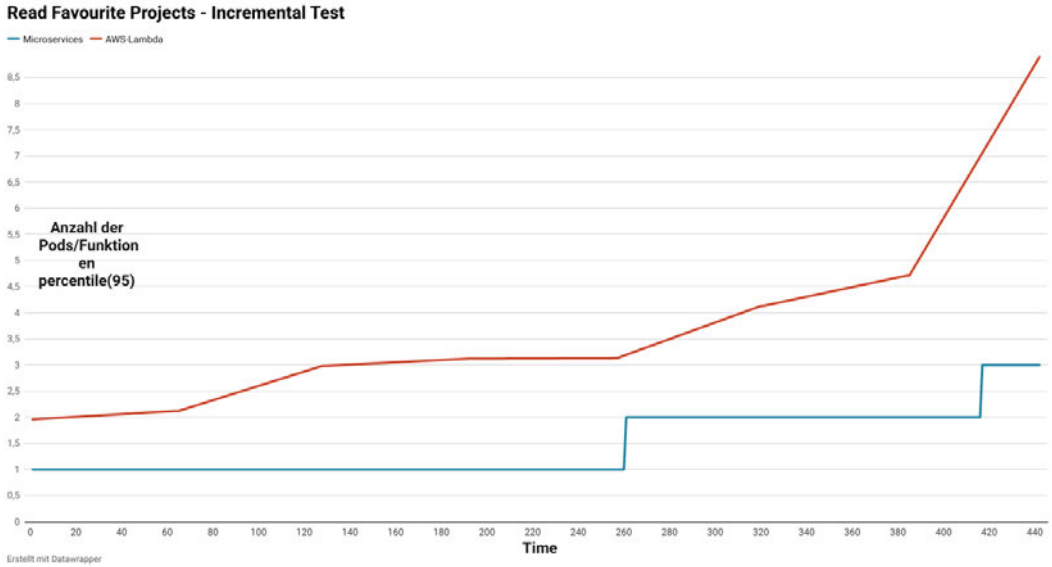

Abbildung 43: Microservices und AWS Lambda - Skalierbarkeit über die Zeit x des ”POST /projects/<id>” Endpoints

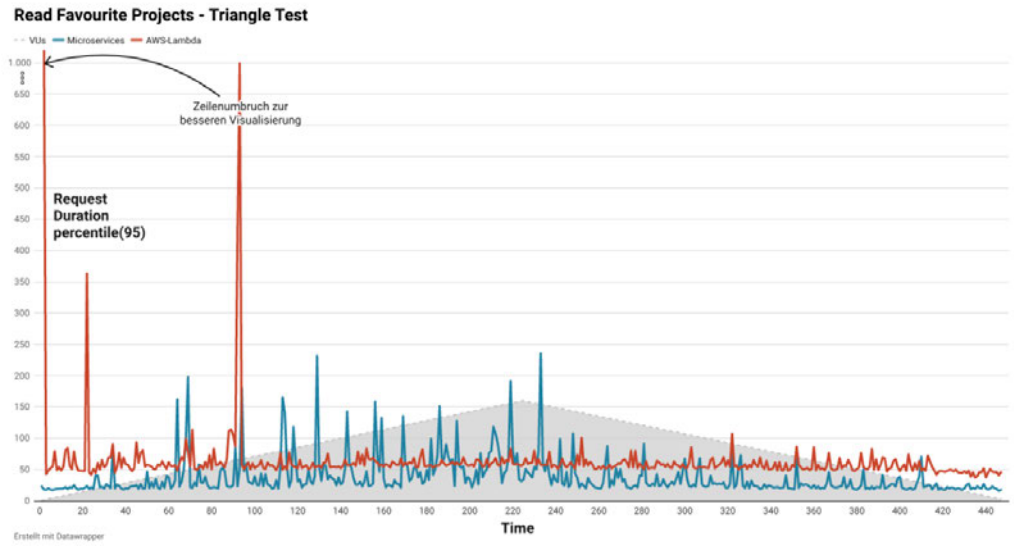

Abbildung 44: Microservices und AWS Lambda - Request-Duration als Percentile-95 über die Zeit x des ”POST /projects/<id>” Endpoints

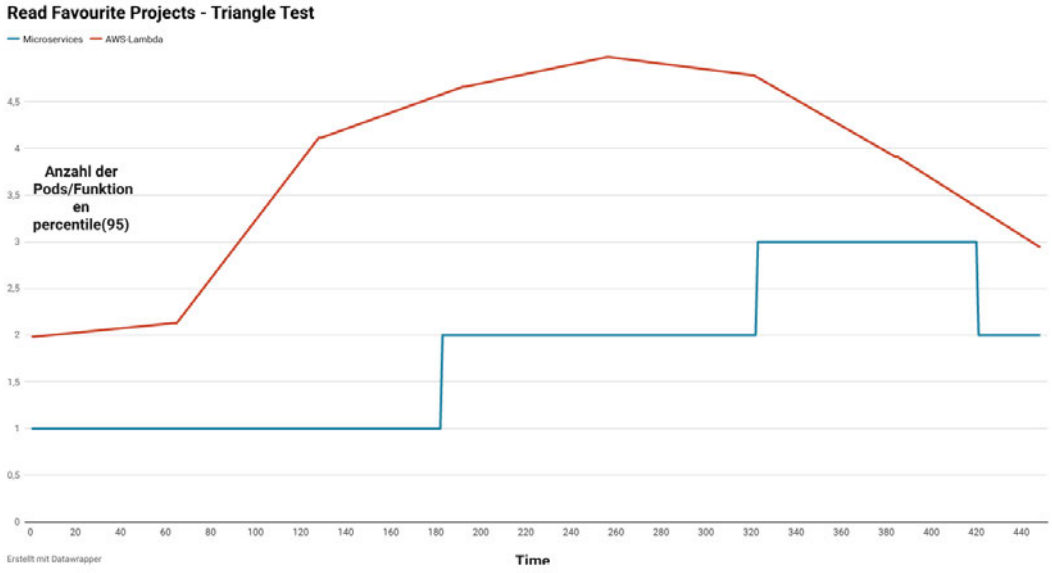

Abbildung 45: Microservices und AWS Lambda - Skalierbarkeit über die Zeit x des ”POST /projects/<id>” Endpoints

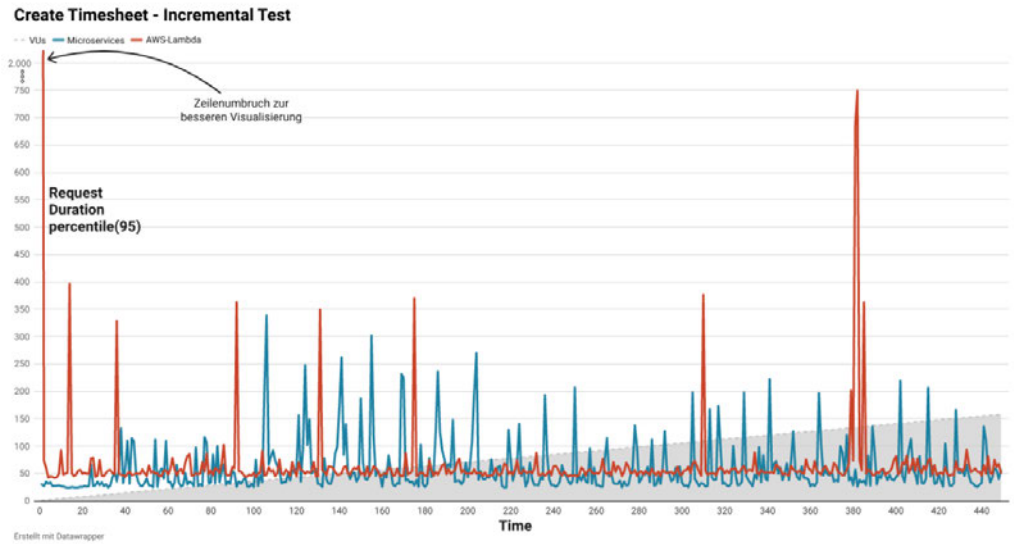

Abbildung 46: Microservices und AWS Lambda - Request-Duration als Percentile-95 über die Zeit x des ”POST /timesheet” Endpoints

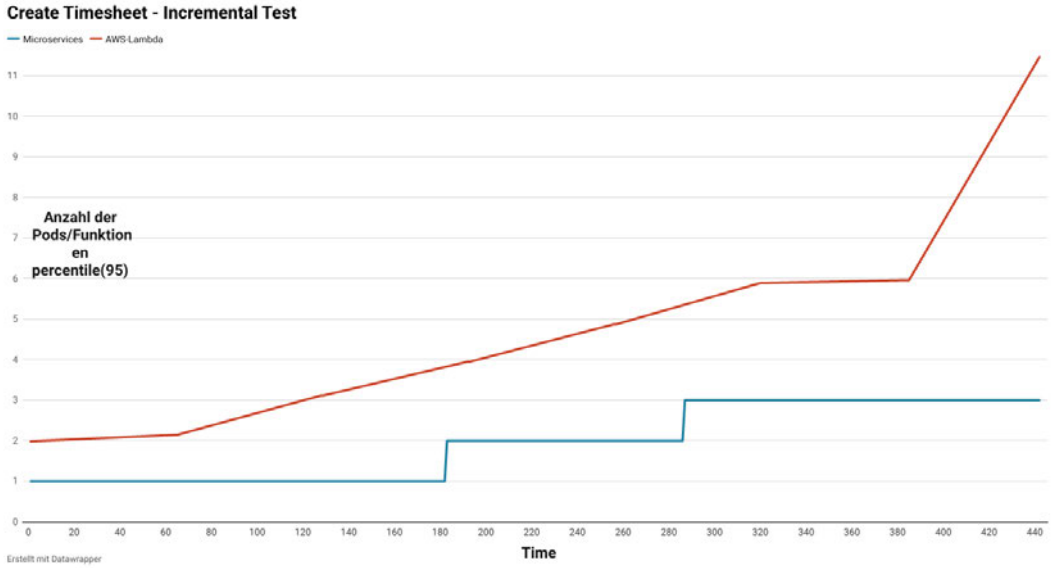

Abbildung 47: Microservices und AWS Lambda - Skalierbarkeit über die Zeit x des ”POST /timesheet” Endpoints

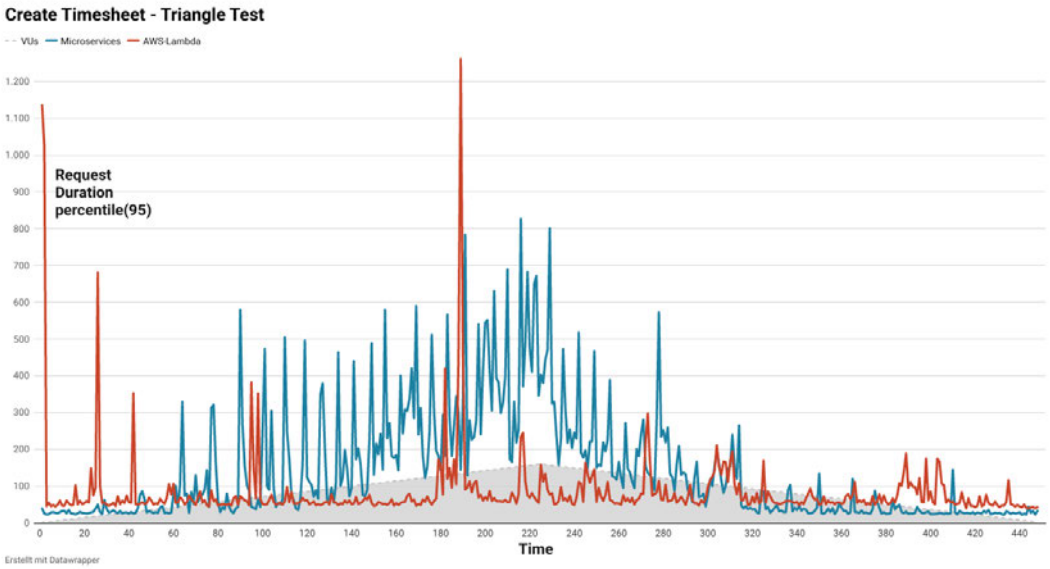

Abbildung 48: Microservices und AWS Lambda - Request-Duration als Percentile-95 über die Zeit x des ”POST /timesheet” Endpoints

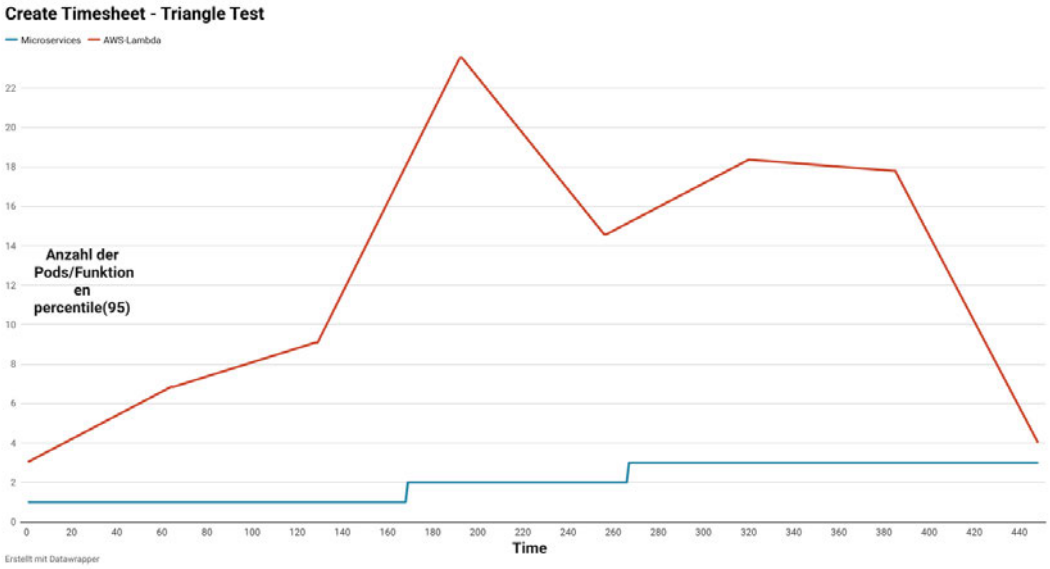

Abbildung 49: Microservices und AWS Lambda - Skalierbarkeit über die Zeit x des ”POST /timesheet” Endpoints

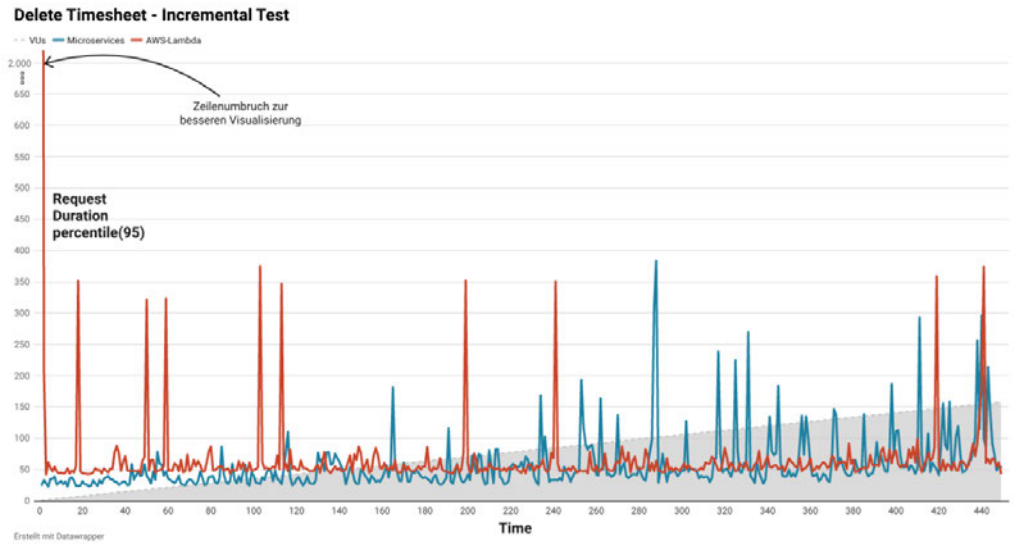

Abbildung 50: Microservices und AWS Lambda - Request-Duration als Percentile-95 über die Zeit x des ”DELETE /timesheet/<id>” Endpoints

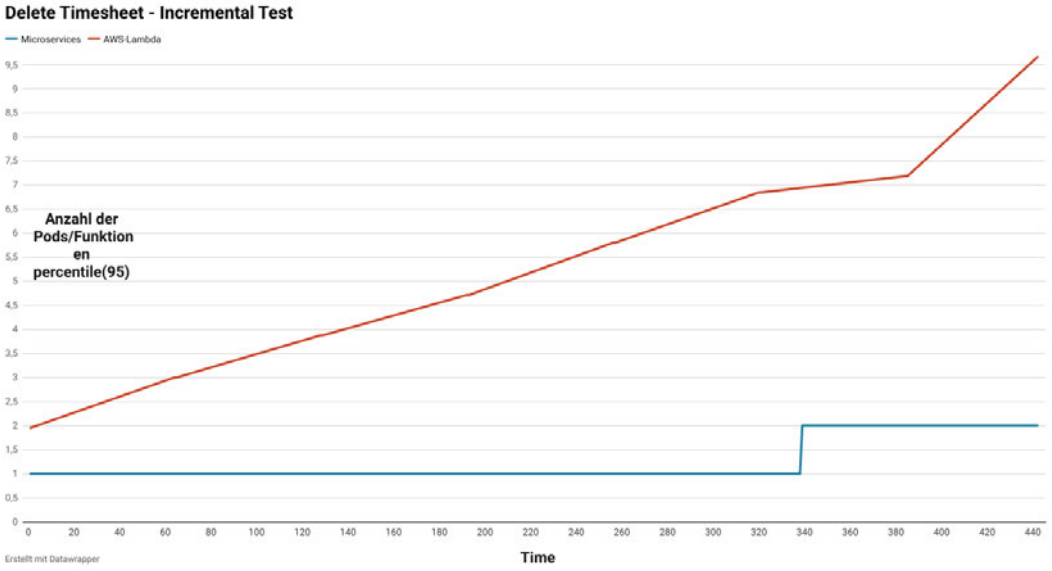

Abbildung 51: Microservices und AWS Lambda - Skalierbarkeit über die Zeit x des ”DELETE /timesheet/<id>” Endpoints

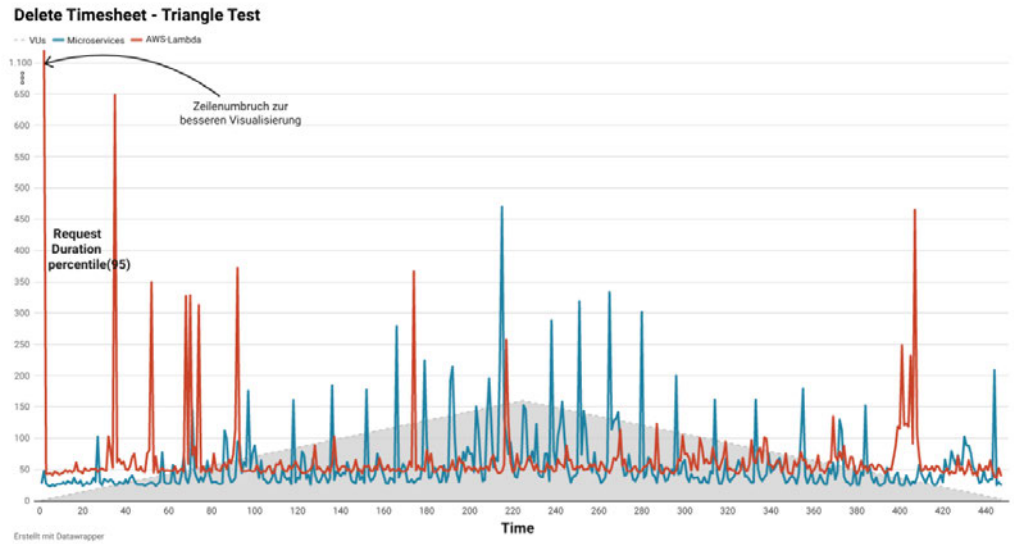

Abbildung 52: Microservices und AWS Lambda - Request-Duration als Percentile-95 über die Zeit x des ”DELETE /timesheet/<id>” Endpoints

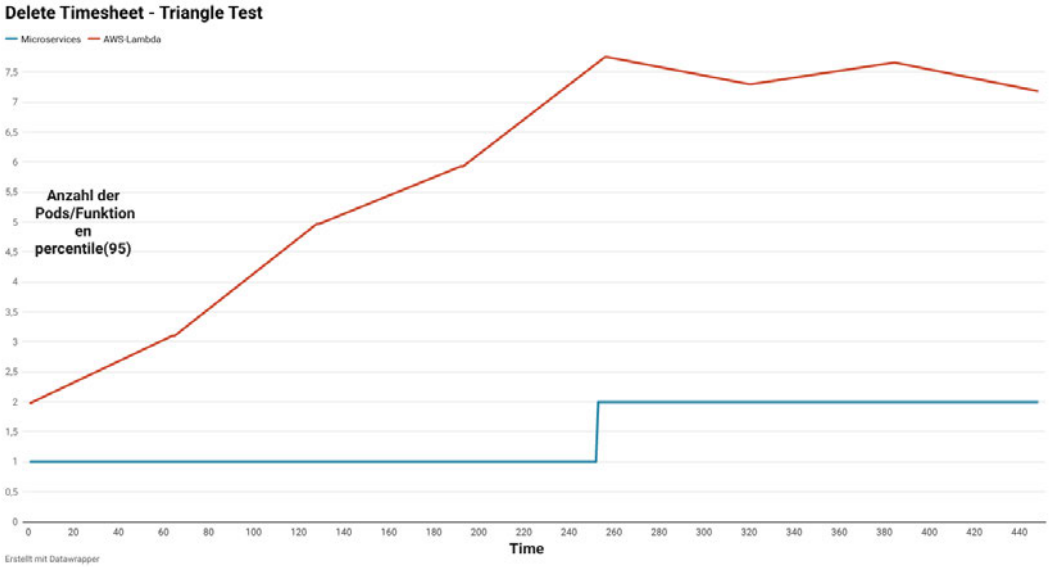

Abbildung 53: Microservices und AWS Lambda - Skalierbarkeit über die Zeit x des ”DELETE /timesheet/<id>” Endpoints

7.4 Ergebnisse der Tests aus dem Kapitel 5.4.3 ”Angepasste CPU- und Memory-Werte”

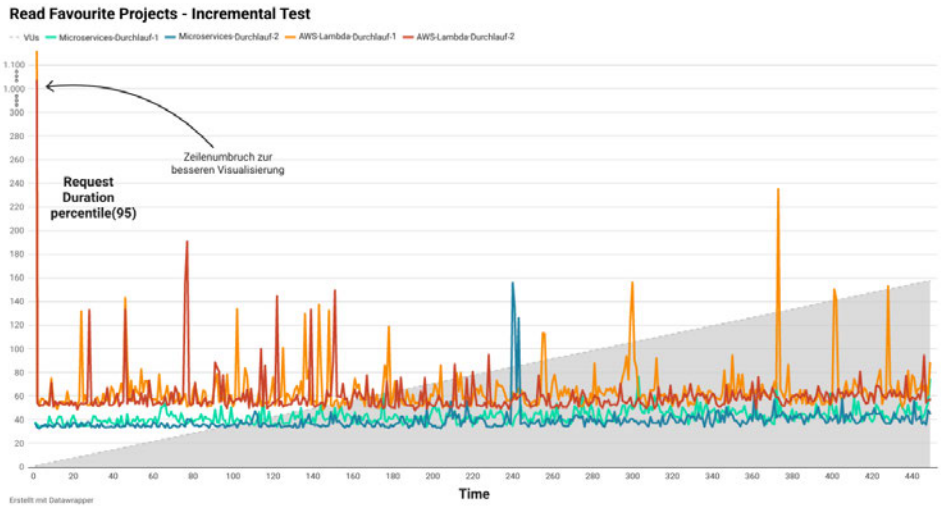

Abbildung 54: Microservices und AWS Lambda - Request-Duration als Percentile-95 über die Zeit x des ”POST /projects/<id>” Endpoints

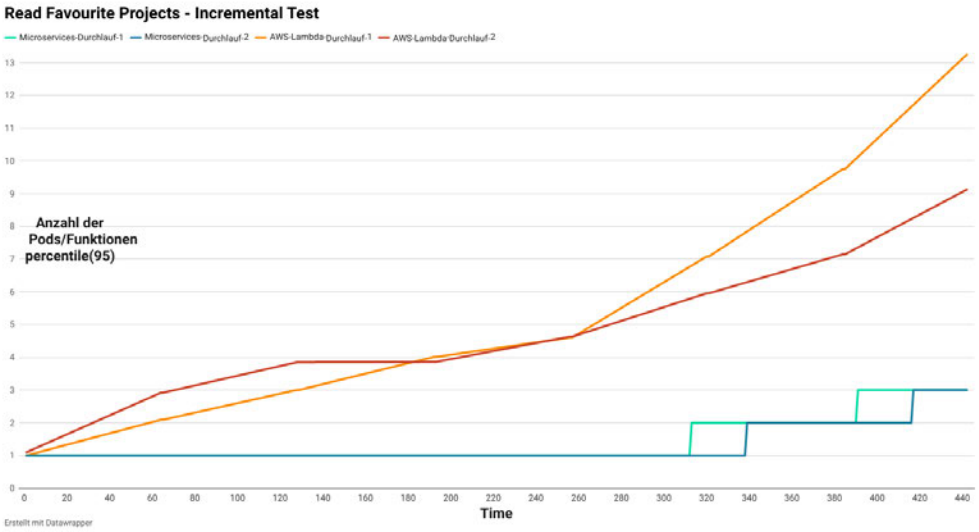

Abbildung 55: Microservices und AWS Lambda - Skalierbarkeit über die Zeit x des ”POST /projects/<id>” Endpoints

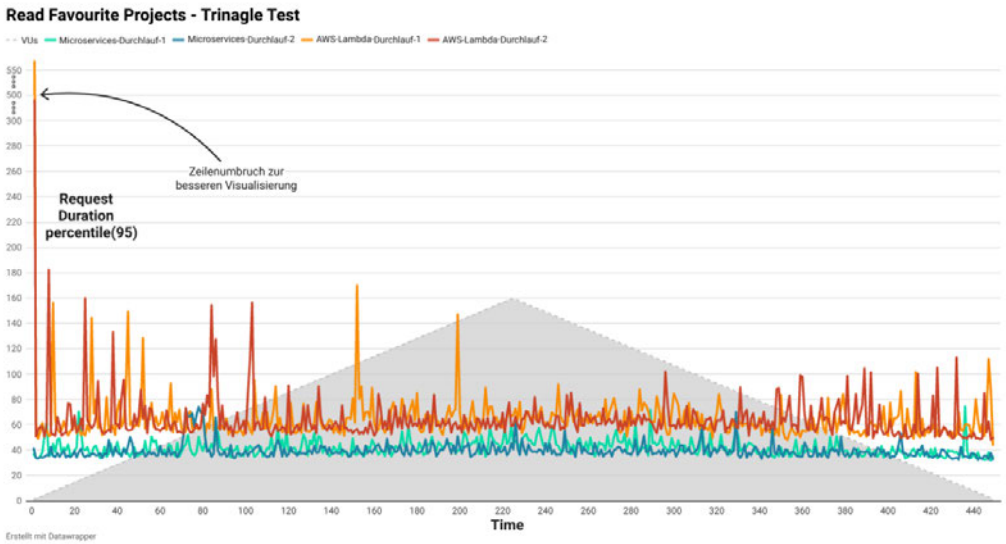

Abbildung 56: Microservices und AWS Lambda - Request-Duration als Percentile-95 über die Zeit x des ”POST /projects/<id>” Endpoints

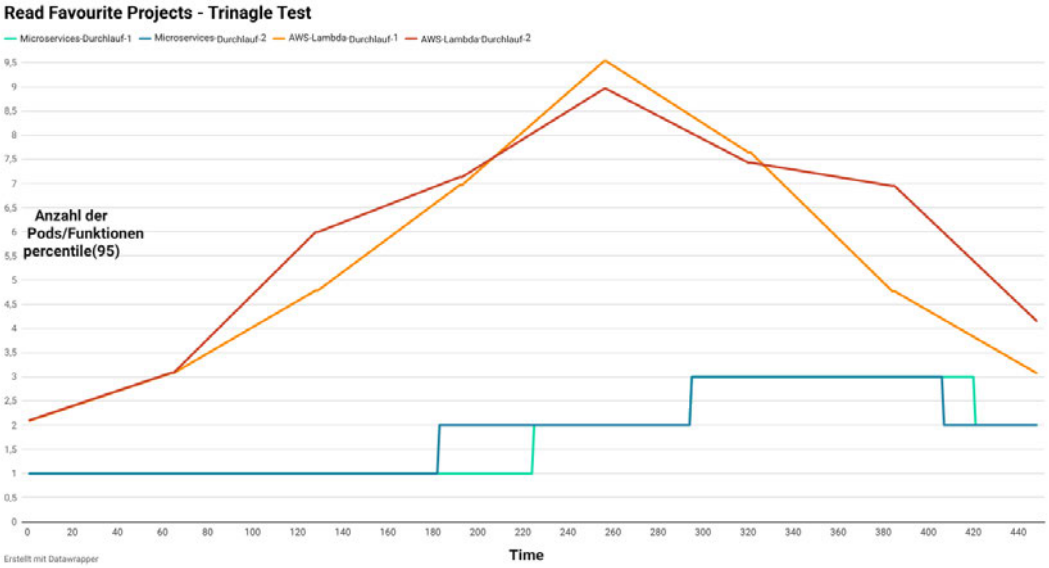

Abbildung 57: Microservices und AWS Lambda - Skalierbarkeit über die Zeit x des ”POST /projects/<id>” Endpoints

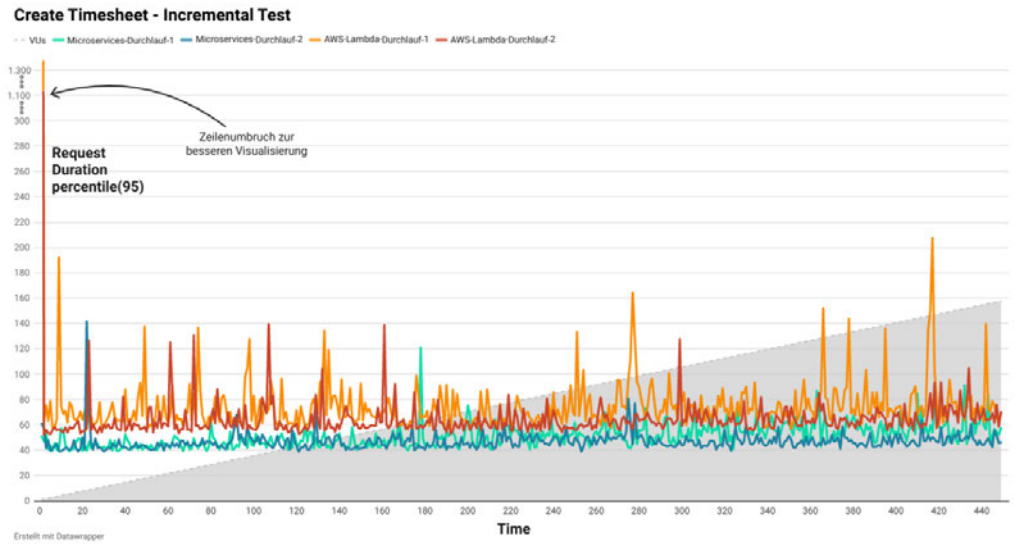

Abbildung 58: Microservices und AWS Lambda - Request-Duration als Percentile-95 über die Zeit x des ”POST /timesheet” Endpoints

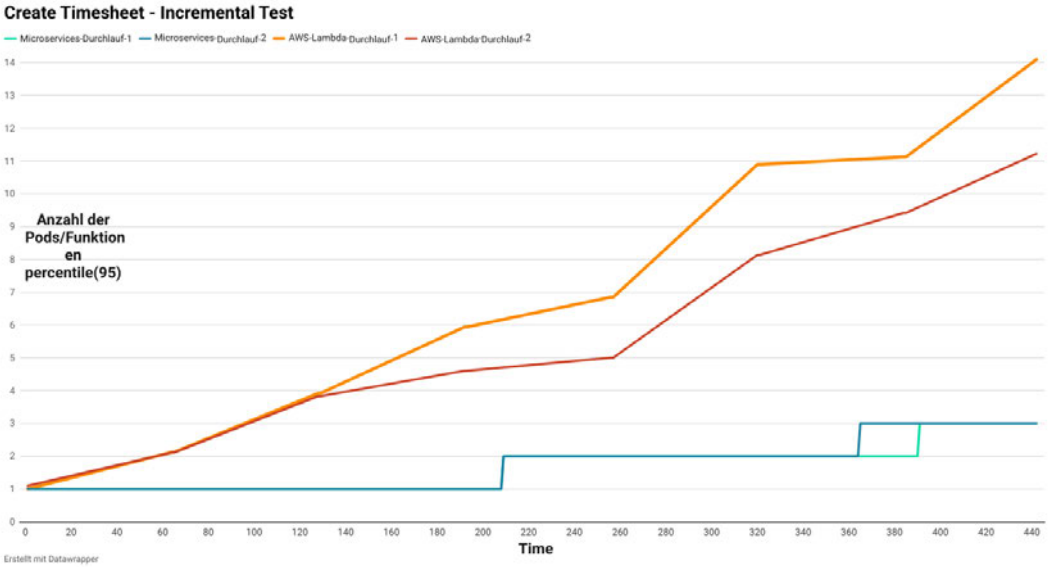

Abbildung 59: Microservices und AWS Lambda - Skalierbarkeit über die Zeit x des ”POST /timesheet” Endpoints

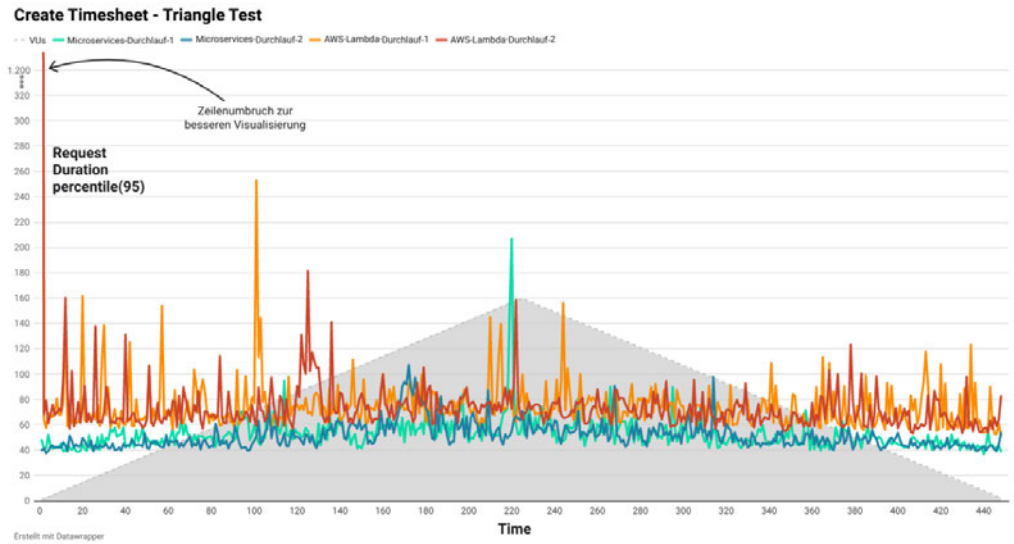

Abbildung 60: Microservices und AWS Lambda - Request-Duration als Percentile-95 über die Zeit x des ”POST /timesheet” Endpoints

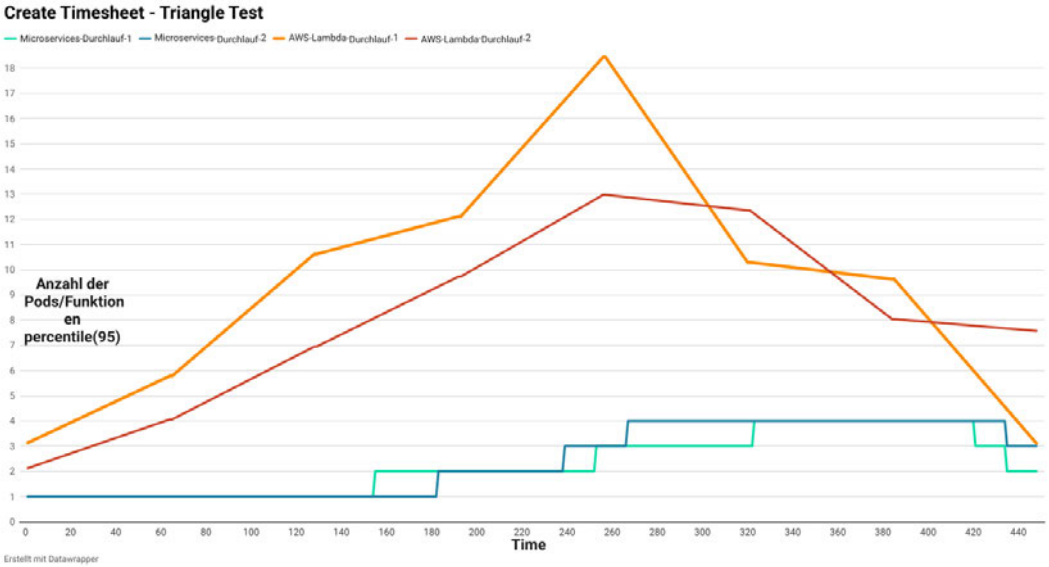

Abbildung 61: Microservices und AWS Lambda - Skalierbarkeit über die Zeit x des ”POST /timesheet” Endpoints

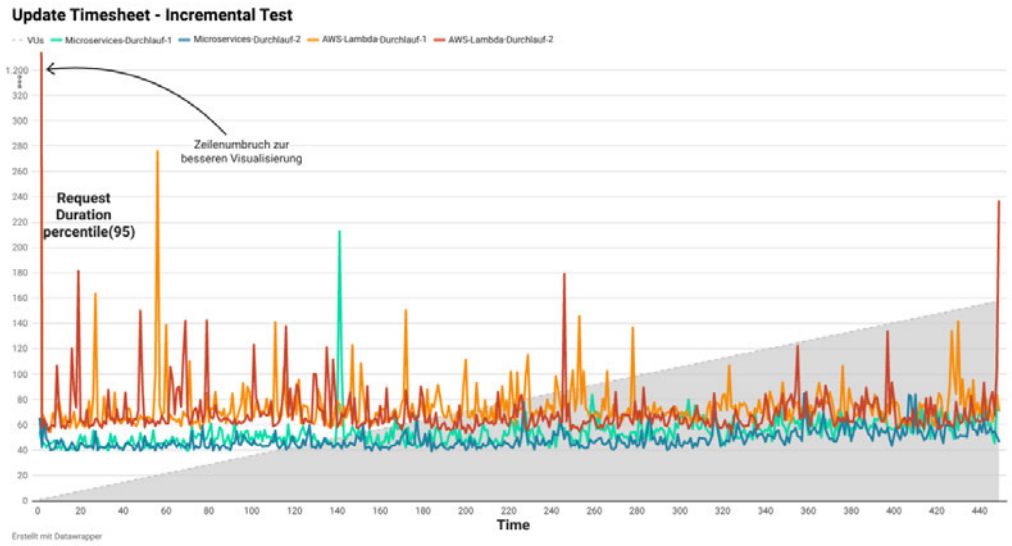

Abbildung 62: Microservices und AWS Lambda - Request-Duration als Percentile-95 über die Zeit x des ”PUT /timesheet/<id>” Endpoints

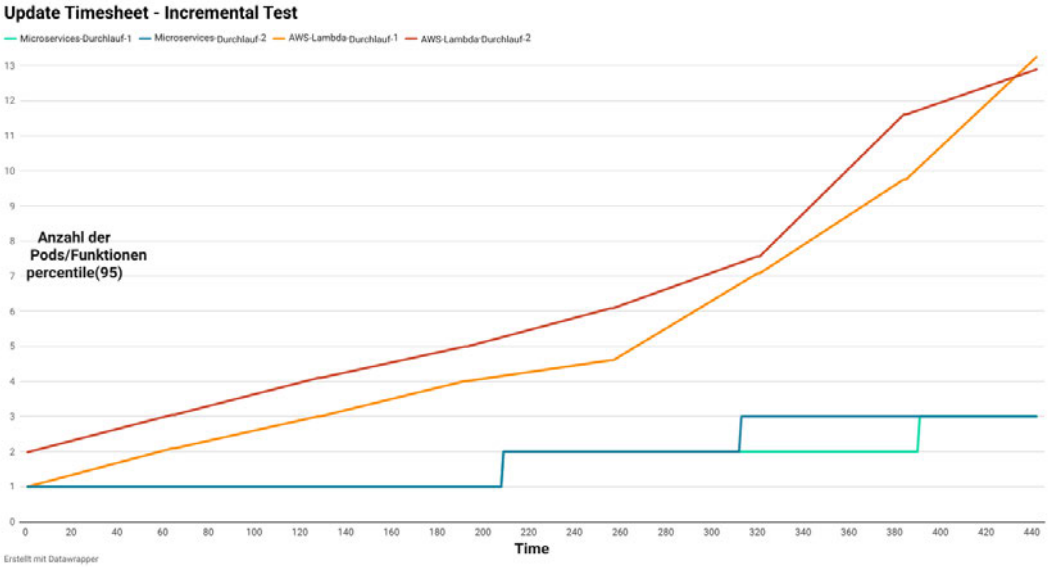

Abbildung 63: Microservices und AWS Lambda - Skalierbarkeit über die Zeit x des ”PUT /timesheet/<id>” Endpoints

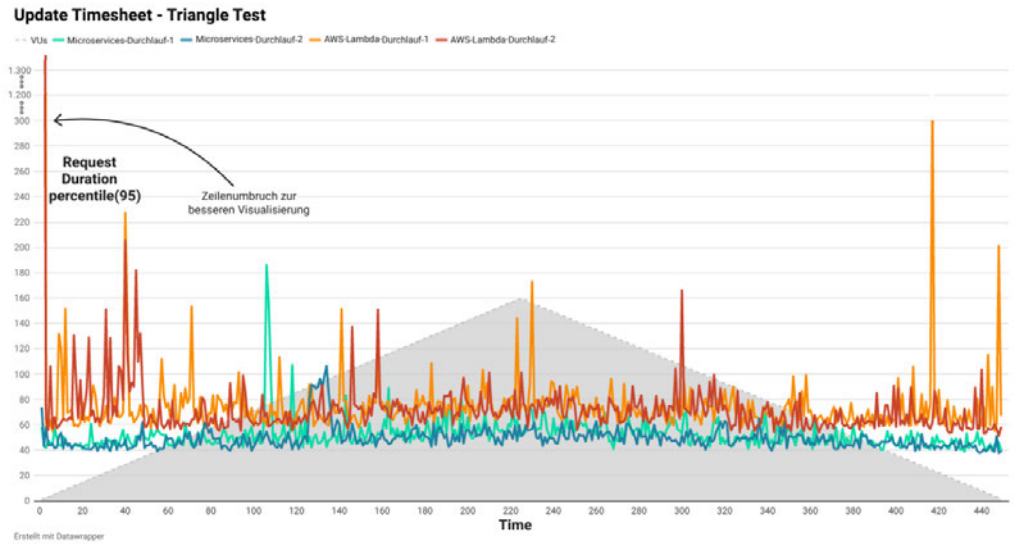

Abbildung 64: Microservices und AWS Lambda - Request-Duration als Percentile-95 über die Zeit x des ”PUT /timesheet/<id>” Endpoints

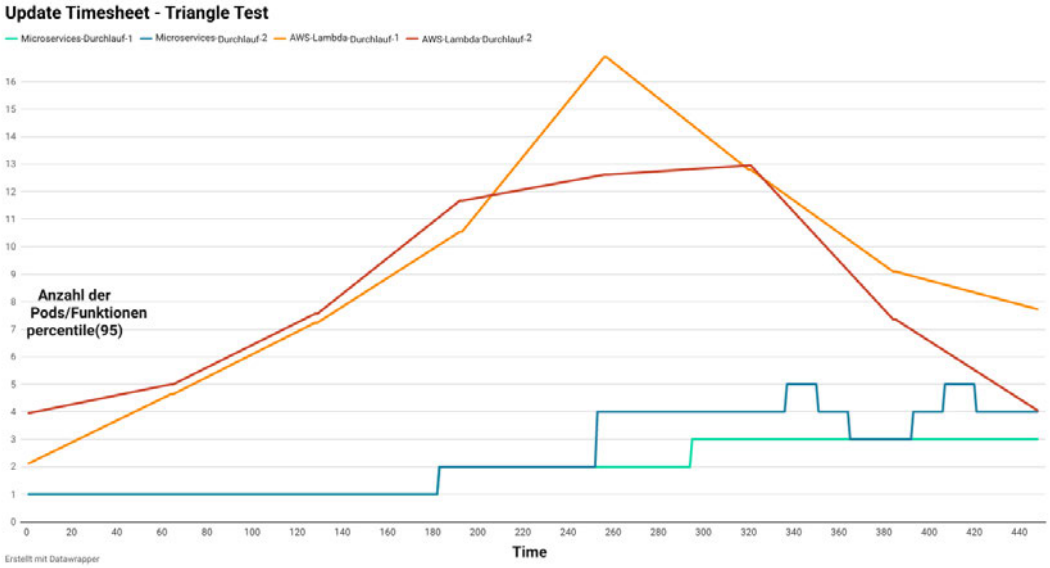

Abbildung 65: Microservices und AWS Lambda - Skalierbarkeit über die Zeit x des ”PUT /timesheet/<id>” Endpoints

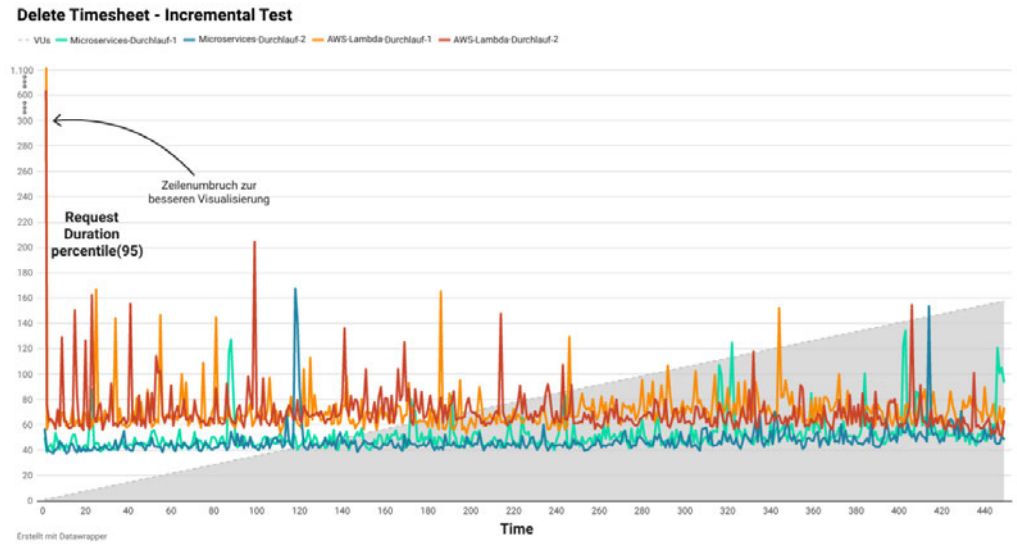

Abbildung 66: Microservices und AWS Lambda - Request-Duration als Percentile-95 über die Zeit x des ”DELETE /timesheet/<id>” Endpoints

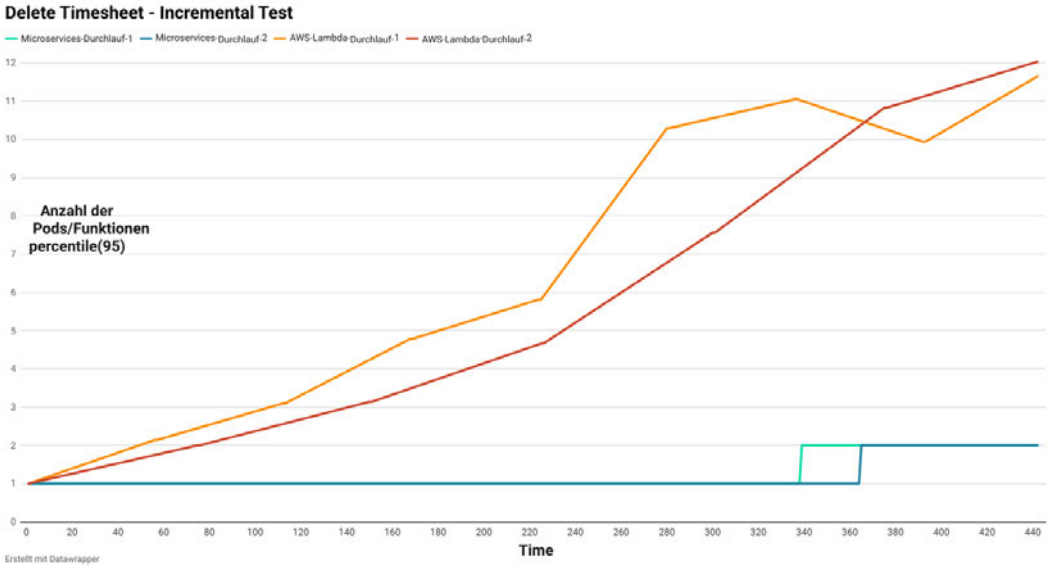

Abbildung 67: Microservices und AWS Lambda - Skalierbarkeit über die Zeit x des ”DELETE /timesheet/<id>” Endpoints

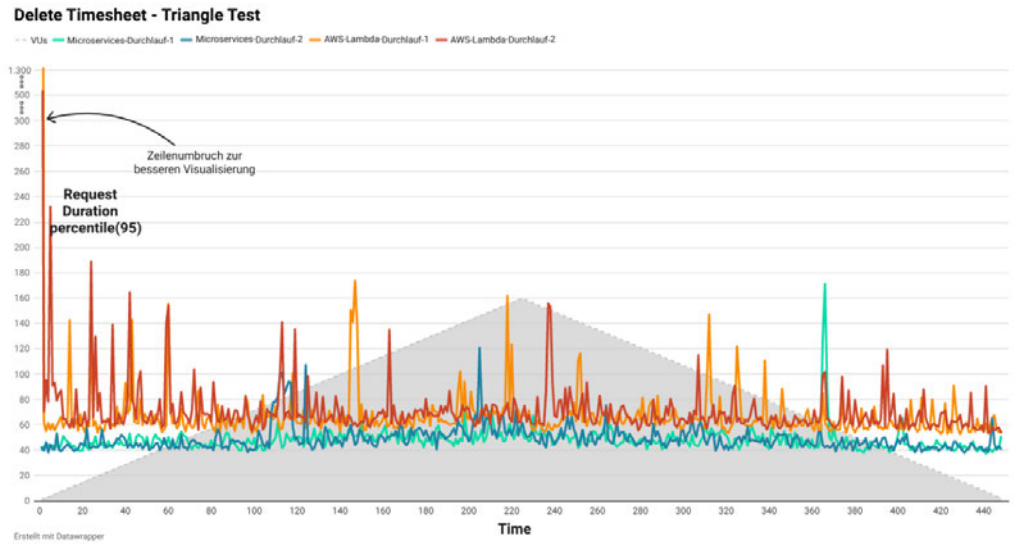

Abbildung 68: Microservices und AWS Lambda - Request-Duration als Percentile-95 über die Zeit x des ”DELETE /timesheet/<id>” Endpoints

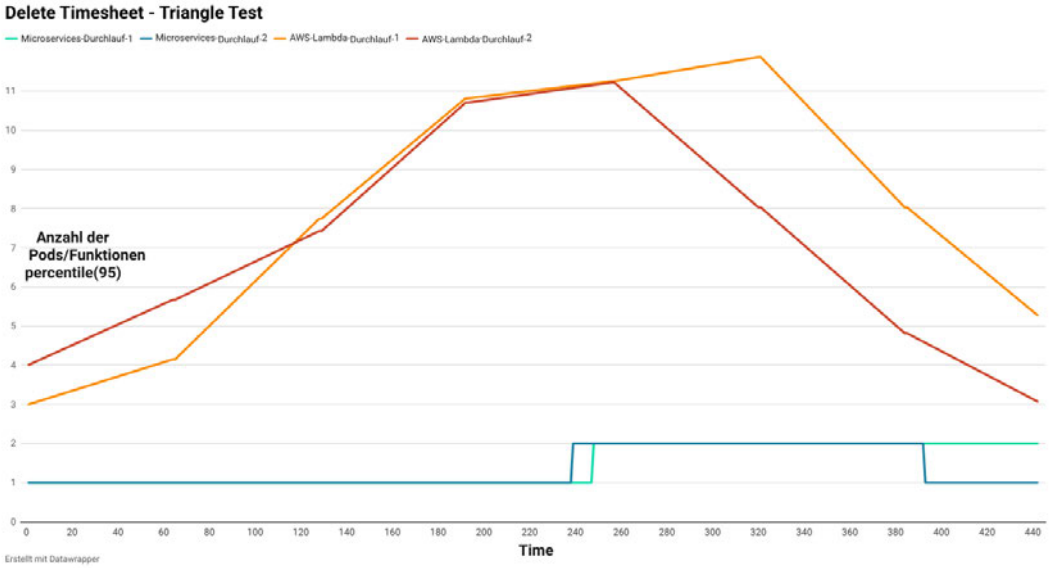

Abbildung 69: Microservices und AWS Lambda - Skalierbarkeit über die Zeit x des ”DELETE /timesheet/<id>” Endpoints

Erklärung zur selbstständigen Bearbeitung einer Abschlussarbeit

Gemäß der Allgemeinen Prüfungs- und Studienordnung ist zusammen mit der Abschlussarbeit eine schriftliche Erklärung abzugeben, in der der Studierende bestätigt, dass die Abschlussarbeit „— bei einer Gruppenarbeit die entsprechend gekennzeichneten Teile der Arbeit [(§ 18 Abs. 1 APSO-TI-BM bzw. § 21 Abs. 1 APSO-INGI)] — ohne fremde Hilfe selbständig verfasst und nur die angegebenen Quellen und Hilfsmittel benutzt wurden. Wörtlich oder dem Sinn nach aus anderen Werken entnommene Stellen sind unter Angabe der Quellen kenntlich zu machen."

Quelle: § 16 Abs. 5 APSO-TI-BM bzw. § 15 Abs. 6 APSO-INGI

Erklärung zur selbstständigen Bearbeitung der Arbeit

Hiermit versichere ich,

Name: _____

Vorname: _____

dass ich die vorliegende Bachelorarbeit – bzw. bei einer Gruppenarbeit die entsprechend gekennzeichneten Teile der Arbeit – mit dem Thema:

Realisierung und Gegenüberstellung einer serverless und einer containerbasierten Microservice-Architektur für das Client-Reporting

ohne fremde Hilfe selbständig verfasst und nur die angegebenen Quellen und Hilfsmittel benutzt habe. Wörtlich oder dem Sinn nach aus anderen Werken entnommene Stellen sind unter Angabe der Quellen kenntlich gemacht.

Ort	Datum	Unterschrift im Original