MASTERARBEIT

FAAS VS. CAAS FÜR DIE ENTWICKLUNG VON REST APIS

ausgeführt am

Studiengang

Informationstechnologien und Wirtschaftsinformatik

Von:

Lukas David, BSc.

Personenkennzeichen: 2010320038

Betreuer:

FH-HON.PROF. ING. DIPL.-ING. (FH) DIPL.-ING. DR.TECHN.

MICHAEL GEORG GRASSER,

MBA MPA CMC

Graz, am 20. März 2022 .....

Unterschrift

EHRENWÖRTLICHE ERKLÄRUNG

Ich erkläre ehrenwörtlich, dass ich die vorliegende Arbeit selbstständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen nicht benützt und die benutzten Quellen wörtlich zitiert sowie inhaltlich entnommene Stellen als solche kenntlich gemacht habe.

.....

Unterschrift

KURZFASSUNG

Cloud Technologien wurden zur Bereitstellung von Infrastruktur für REST APIs in den letzten Jahren immer wichtiger. Das Container-as-a-Service (CaaS) Modell bei welchen Anwendungen virtualisiert in der Cloud laufen stellt hierbei einen der meistgenutzten Services dar. Dabei kommt es jedoch immer noch zu einem Overhead bei Verwaltung und Bereitstellung der Container, wodurch Entwicklungsressourcen verloren gehen. Eine noch leichtgewichtigere Alternative ist das neuere Function-as-a-Service (FaaS) Modell, welches den Verwaltungsaufwand reduzieren soll.

Ziel der Masterarbeit ist es herauszufinden, ob FaaS sich besser als CaaS für die Entwicklung von REST APIs eignet. Um die Modelle vergleichbar zu machen, werden die Kriterien Kosten sowie Performance herangezogen und es wird folgende Forschungsfrage gestellt: ‚Wie wirkt sich FaaS gegenüber CaaS auf Performance und Kostenentwicklung einer REST API aus?'.

Um die Forschungsfrage zu beantworten, wurde jeweils eine CaaS und eine FaaS API implementiert. Als Metrik für Performance wurde dabei die Response Time (RT) gewählt. Diese gibt die Zeit an, welche zwischen Senden des API Requests und dem Empfang der Response verstreicht. Die RT wurde automatisiert mittels Artillery Skripts gemessen, um gleichzeitige Nutzung durch 1.000 virtuelle Benutzer*innen zu simulieren und verschiedene Anwendungsfälle abzubilden. Anhand der RT Daten wurden anschließend Kosten für verschiedene Benutzerzahlen berechnet.

Die Ergebnisse der Performancemessungen zeigen, dass die RT von FaaS durchwegs höher ist als jene von CaaS. Für zeitkritische Applikationen kann FaaS daher nicht empfohlen werden. Bei den Kosten zeigt sich ein gemischtes Bild. Gerade bei wenigen API-Aufrufen entstehen bei FaaS kaum Kosten im Vergleich zu CaaS. Je höher die Anzahl der API Aufrufe, umso mehr verkehrt sich dieser Umstand jedoch ins Gegenteil, bei zwölf Millionen täglichen Aufrufen ist die FaaS Lösung schon beinahe dreimal so teuer wie CaaS. FaaS bietet sich also vor allem für kleinere Applikationen an, um einen Kompromiss zwischen Performance und Kosten zu finden.

ABSTRACT

Cloud technologies have become increasingly important infrastructure for REST APIs. The Container-as-a-Service (CaaS) model, running virtualized applications in the cloud, is particularly widely used. However, there is still an overhead in managing and deploying the containers, which means that development resources are lost. A more lightweight alternative, the newer Function-as-a-Service (FaaS) model, reduces the management overhead. This thesis determines whether FaaS is more suitable than CaaS for developing REST APIs. The criteria cost and performance are used to compare the models. The following research question is defined: 'How does FaaS affect the performance and cost structure of a REST API compared to CaaS?' A CaaS and a FaaS API were implemented to answer the research question. Response Time (RT), the time that elapses between sending an API request and receiving a response from the server, was chosen as a metric to measure performance. The RT was measured using Artillery scripts to simulate simultaneous use by 1000 virtual users and to cover different customer journeys. The RT data were then used to calculate costs for different user numbers. The performance measurements show that the RT of FaaS is consistently higher than that of CaaS. Therefore, FaaS cannot be recommended for time-critical applications. Regarding costs, the picture is mixed. Especially with few API calls, FaaS hardly incurs any costs compared to CaaS. However, the higher the number of API calls, the more this reverses. At twelve million daily API requests, the FaaS solution is almost three times as expensive as CaaS. FaaS is ideal for smaller applications striking a compromise between performance and costs.

INHALTSVERZEICHNIS

1 EINLEITUNG

Der Einsatz von Cloud Computing Technologien ist ein Markttrend, welcher in den letzten Jahren stetig im Vormarsch war. Das Versprechen einer skalierbaren sowie kosteneffizienteren Infrastruktur war hierfür eines der Hauptargumente. Instanzbasierte Modelle wie Platform-as-a-Service (PaaS) wurden in diesem Bereich immer wichtiger. Trotz der individuellen Verwaltungsmöglichkeiten wird am PaaS Modell Kritik aufgrund der ineffizienten Nutzung von Ressourcen geübt (Albuquerque Jr. et al., 2018). Eine Alternative stellt Containerisierung mit Container-as-a-Service (CaaS) dar. CaaS bietet eine Möglichkeit, eine Anwendung innerhalb einer eigenen Umgebung zu isolieren. Da bei CaaS einzelne Anwendungen virtualisiert werden und kein eigenständiges Betriebssystem benötigt wird, ist es eine leichtgewichtigere Alternative zur vollwertigen Virtualisierung. Die Container müssen trotz allem, jedoch immer noch deployt und verwaltet werden, sei es auf der internen Infrastruktur oder in der Cloud. Es müssen also immer noch Entwicklungsressourcen für Konfiguration und Verwaltung der Container eingeplant werden (Sbarski, 2017).

Das neuere Function-as-a-Service (FaaS) Modell soll hier eine Alternative darstellen, welche eine noch effizientere Nutzung der Ressourcen ermöglicht und zeitgleich geringere Kosten anfallen (Albuquerque Jr. et al., 2018).

Als FaaS werden einzelne Funktionen bezeichnet, welche die Logik beinhalten, um auf ein genau vordefiniertes Event zu reagieren. Es handelt sich also um sehr kleine Codestücke. Der Code dieser Funktionen kann anschließend in eine Cloud Umgebung geladen werden, so bietet der Cloudprovider Amazon Web Services (AWS) hierfür z.B. den Service AWS Lambda an. Dadurch werden die Anforderungen an das Infrastrukturmanagement verringert, da man sich keine Gedanken mehr über Betriebssystem, oder Container Management machen muss und ein vergleichsweise minimaler Konfigurationsaufwand anfällt (Cloud Native Computing Foundation, 2018).

Der Verfasser der Arbeit ist beruflich als Softwareentwickler bei der Firma ebcont tätig. Daraus resultiert das Interesse, sich mit dem Themengebiet FaaS zu beschäftigen. Innerhalb des Unternehmens wird sehr stark auf Virtualisierung mit Amazon Elastic Container Services (CaaS) gesetzt. Hier kommt es beim Deployment immer wieder zu Problemen bzw. werden dafür DevOps Spezialist*innen benötigt. Durch die Erkenntnisse der Masterarbeit soll herausgefunden werden, ob FaaS hier ein geeigneter Ersatz wäre, da dadurch die Komplexität beim Deployment reduziert werden könnte und Entwickler*innen sich stärker auf die Implementierung der Businesslogik konzentrieren könnten.

Ziel dieser Masterarbeit ist, herauszufinden inwieweit FaaS eine sinnvolle Alternative gegenüber CaaS für die Entwicklung von REST APIs ist. Im theoretischen Teil werden die unterschiedlichen Architekturansätze der Modelle erklärt. Beim praktischen Forschungsteil wird einmal eine API mit FaaS umgesetzt werden und dann der ältere Ansatz mit CaaS gewählt. Diese beiden APIs werden anschließend hinsichtlich Performance verglichen. Bei FaaS wird im Gegensatz zu CaaS nicht für fixe Ressourcen, sondern pro Aufruf der Funktion gezahlt (Stojanovic & Simovic, 2019).

Daher sollen auch die Kosten zwischen den APIs verglichen werden und es wird die Frage beantwortet, ob eine FaaS Architektur auch für eine große Anzahl an Benutzer*innen noch geeignet ist.

Somit ergibt sich folgende Forschungsfrage:

- Wie wirkt sich FaaS gegenüber CaaS auf Performance und Kostenentwicklung einer REST API aus?

Diese implementierten APIs sollen grundlegende Backend-Anforderungen wie Einfügen, Aktualisieren und Löschen von Datensätzen in eine Datenbank erfüllen. Aus Vergleichbarkeitsgründen wird hier mit derselben Datenbank gearbeitet. Anschließend wird die Performance bei einzelnen Aufgaben gemessen und gegenübergestellt. Für beide Lösungen wird AWS als Cloudanbieter gewählt.

Da die Kosten bei FaaS von der Laufzeit der Funktionen abhängen, werden diese mit den Performancedaten der Implementierung geschätzt und den Kosten von CaaS gegenübergestellt.

Als grundlegende Arbeitshypothese für die Masterarbeit kann formuliert werden:

- H1: Die Performance der FaaS Implementierung ist besser als die der CaaS Implementierung
- H2: FaaS verursacht gegenüber CaaS geringere Kosten

2 GRUNDLAGEN

In diesem Kapitel werden grundlegende Begriffe erläutert werden, welche nötig sind, um ein besseres Verständnis für die Arbeit zu schaffen.

2.1 Serverless

Sowohl CaaS als auch FaaS folgen dem Serverless Paradigma. Um Serverless und die Gründe dahinter zu verstehen, sollte man sich bewusst machen, welche Tätigkeiten für Entwickler*innen bei der Softwareentwicklung früher typischerweise angefallen sind. Neben der Entwicklung des Programmes selbst musste die entsprechende Hardware aufgesetzt und konfiguriert, Speicher bereitgestellt und das Netzwerk entsprechend eingerichtet werden. Durch all diese Tätigkeiten werden Entwicklungsressourcen gebunden (Gupta, 2018). Für die meisten Unternehmen war die Beschaffung von Hardware eine mühsame Nebenaufgabe, ging es doch hauptsächlich darum einen Business Value durch das Softwareprodukt zu schaffen. Betroffen waren hiervon auch sehr stark kleinere Unternehmen und Startups. Wurde ihr Produkt plötzlich populär, war es ihnen nicht möglich ihre Hardware sofort zu skalieren und sie mussten erst auf das Eintreffen neuer physischer Hardware warten. Zur Lösung dieser Probleme wurde nach einer Möglichkeit gesucht, die Hardware zu abstrahieren, um besser skalieren und auf Fehler reagieren zu können. Aus all diesen Gründen wurden Cloud Technologien entworfen (Patterson, 2019). Doch auch mit dem Aufkommen von Cloud Technologien in Verbindung mit virtuellen Maschinen (Infrastructure as a Service) wurde dieses Problem nicht gelöst, da hierbei immer noch Server installiert, konfiguriert und gewartet werden müssen (Gupta, 2018). Eine weitere Herausforderung für Entwickler*innen war hierbei auch die richtige Skalierung der Server. Die meisten Systemdesigner*innen haben sich bei der Zuteilung der Ressourcen für Server in der Cloud an der höchsten Auslastung der Applikation orientiert. Dadurch sollte auch zu Spitzen Nutzungszeiten eine hohe Performance gewährleistet werden. Studien über die gemeldete Nutzung von Cloud-Ressourcen in Rechenzentren zeigen jedoch eine erhebliche Lücke zwischen den Ressourcen, die Cloud-Kunde*innen zuweisen und bezahlen und der tatsächlich benötigten Ressourcennutzung. Somit fallen in vielen Fällen durch die Cloud unnötige Kosten an, da Ressourcen nicht optimal genutzt werden bzw. zu viele Ressourcen zugeordnet sind, welche Großteils nicht benötigt werden (Castro et al., 2019).

Einen anderen Ansatz verfolgt das neuere sogenannten Serverless Paradigma. Hierbei werden Bereitstellung und andere operative Tätigkeiten auf Serverebene vor Nutzer*innen verborgen. Obwohl der Name Serverless lautet, werden im Hintergrund natürlich immer noch Server verwendet und es wird nur der Managementaufwand reduziert (Gupta, 2018). Eine häufig genutzte Definition des Begriffes Serverless beschreibt Serverless Computing als eine Plattform welche Server Konfigurationen vor den Nutzer*innen verbirgt, Code-on-demand ausführt, automatisch skaliert und nur für die Zeit Kosten anfallen, in welcher die Applikation läuft. Diese Definition hebt auch gleich die zwei Schlüsseleigenschaften von Serverless Computing hervor:

- Man bezahlt nur für den tatsächlichen Ressourcenverbrauch (pay-as-you-go). Da Server und ihre Nutzung nicht direkt Teil des Serverless Computing Modells sind (Nutzer*innen haben keinen direkten Serverzugriff) bezahlt man nur die Zeit, in welcher eigene Applikationen gerade laufen. Dadurch fallen keine Kosten bei Leerlaufzeiten an. Natürlich ist es jedoch trotzdem möglich Obergrenzen für Speicher und CPU anzugeben.
- Skalierung von null bis unendlich. Da die Entwickler*innen weder die Kontrolle über die Server haben, auf denen ihr Code läuft, noch die Anzahl der Server kennen, auf denen ihr Code läuft, werden Entscheidungen über die Skalierung den Cloud-Anbietern überlassen. Entwickler*innen müssen dabei keinerlei Richtlinien für die automatische Skalierung physischer Komponenten wie CPU und Speicher festlegen. Stattdessen verlassen sie sich darauf, dass der Cloud-Anbieter automatisch parallele Ausführungen startet, wenn eine größere Nachfrage besteht. Außerdem können die Entwickler*innen auch davon ausgehen, dass der Cloud-Anbieter sich um die Wartung, Sicherheitsaktualisierung, Verfügbarkeit und Überwachung der Server kümmert (Castro et al., 2019).

Es gibt zwei verschiedene Möglichkeiten Applikationen in die Cloud zu bringen ohne sich um die Server Infrastruktur dahinter kümmern zu müssen. Da im praktischen Teil der Arbeit mit AWS gearbeitet wird, wird hierbei immer auf die jeweilige Umsetzung durch AWS Bezug genommen. Die erste Option besteht darin, Docker Container direkt in der Cloud laufen zu lassen. Dies wird als CaaS bezeichnet. Bei AWS wird CaaS durch eine Kombination des Elastic Container Service und AWS Fargate umgesetzt (Ifrah, 2019). Bei der zweiten Möglichkeit wird der Code durch Entwickler*innen direkt in die Cloud hochgeladen. Da der Code hierfür in einzelne kleinere Funktionen aufgeteilt werden muss, entstand der Begriff FaaS (Kanikathottu, 2019). Die erwähnten Begriffe werden in den Kapiteln zu CaaS und FaaS näher erläutert.

2.2 REST APIs

REST steht für Representational State Transfer und ist eine Architektur Richtlinie für die Kommunikation zwischen verteilten Systemen. Laut Roy Fielding dem Entwickler des REST Paradigmas gibt es sechs wichtige Eigenschaften, welche für eine REST Architektur erfüllt sein müssen. Es wird jedoch nicht vorgegeben, wie diese implementiert werden müssen. Die Eigenschaften sind:

1. Client - Server
2. Zustandslosigkeit
3. Caching
4. Einheitliche Schnittstelle
5. Mehrschichtige Systeme
6. Code-on-demand (Doglio, 2018)

Client-Programme verwenden APIs, um mit Webservices zu kommunizieren. APIs gibt es beinahe so lange wie Computer Code selbst. Im Allgemeinen stellt eine API eine Reihe von Daten und Funktionen zur Verfügung, um die Interaktion zwischen Computerprogrammen zu erleichtern und ihnen den Austausch von Informationen zu ermöglichen (Massé, 2021). Eine Klasse, die bestimmte öffentliche Methoden bereitstellt, die ein anderer Code aufrufen kann, hat eine API. Ein Skript, das bestimmte Arten von Eingaben akzeptiert, hat eine API. Ein Treiber auf einem Computer, der von Programmen auf eine bestimmte Weise aufgerufen werden muss, hat eine API. Mit der Entwicklung des Internets wurde der Begriff API jedoch immer enger gefasst. Wenn jetzt von einer API die Rede ist, meint man damit fast immer eine Web-API. Eine Web-API greift das Konzept einer Schnittstelle zwischen zwei Komponenten auf setzt auf das Prinzip der Client/Server-Beziehung (Westerveld, 2021). Dieses Prinzip auf welchen Web-APIs basieren, wird in Kapitel 2.2.1 näher erläutert. Der REST-Architekturstil ist mittlerweile der Standard für die Entwicklung von APIs moderner Webservices. Eine Web-API, die dem REST-Architekturstil entspricht, wird dabei als REST-API bezeichnet (Massé, 2021). Obwohl REST hauptsächlich mit dem Hypertext Transfer Protocol (HTTP) genutzt wird, ist dies keine Vorgabe des Architekturstils, sondern es könnte auch jedes andere Protokoll verwendet werden (Sharma, 2021). Auch in dieser Arbeit wird REST mit HTTP verwendet.

2.2.1 Client - Server

Ein Server bietet eine Reihe von Diensten an und wartet auf Requests bezüglich dieser Dienste. Die Requests wiederum werden von einem Client System, welches diese Dienste benötigt, gestellt. Dadurch sollen Zuständigkeiten getrennt werden. Es soll der Front-End-Code, wie Darstellung und UI-bezogene Verarbeitung der Informationen vom serverseitigen Code, der sich um die Speicherung und serverseitige Verarbeitung der Daten kümmert, getrennt werden. Dadurch ist die unabhängige Entwicklung beider Komponenten möglich. So kann die Client-Anwendungen verbessert werden, ohne den Server-Code zu beeinträchtigen und umgekehrt (Doglio, 2018).

2.2.2 Zustandslosigkeit

Die Kommunikation zwischen Client und Server muss zustandslos sein, was bedeutet, dass jede vom Client gestellte Anfrage alle Informationen enthalten muss, die der Server benötigt, um sie zu verstehen, ohne auf gespeicherte Daten zurückzugreifen.

Diese Restriktion bringt mehrere Verbesserungen für die zugrunde liegende Architektur mit sich:

- Sichtbarkeit: Die Überwachung des Systems ist einfach, da alle erforderlichen Informationen in der Anfrage enthalten sind und keine vorherigen Anfragen betrachtet werden müssen.
- Skalierbarkeit: Da keine Daten zwischen den Anfragen gespeichert werden müssen, kann der Server Ressourcen schneller freigeben.
- Verlässlichkeit: Ein zustandsloses System kann sich viel leichter von einem Ausfall erholen als ein nicht zustandsloses, da nur die Anwendung selbst, aber keine Anfragen zwischen Systemen wiederhergestellt werden müssen.
- Leichtere Implementierung: Das Schreiben von Code, der keine Zustandsdaten über mehrere Server hinweg verwalten muss, ist viel einfacher zu implementieren, wodurch die Systeme weniger komplex sind.

Obwohl diese Einschränkung viele Vorteile mit sich bringt, gibt es doch einen Nachteil. Dadurch, dass jede Anfrage alle relevanten Informationen enthalten muss, entsteht ein Overhead da, Statusinformationen wiederholt gesendet werden müssen (Doglio, 2018).

2.2.3 Caching

Jede Antwort auf eine Anfrage soll Serverseitig im Cache zwischengespeichert werden. Auf Serverseite muss daher sollte zweimal nacheinander dieselbe Anfrage gestellt werden nicht mehr auf die Datenbank zugegriffen werden, sondern die Daten werden aus dem Cache geliefert. Dies führt zu einer Leistungsverbesserung, wodurch der Client, der die zweite Anfrage stellt, eine deutlich schnellere Antwort erhält. Der Nachteil besteht darin, dass der Client veraltete Daten aus

dem Cache erhalten kann, sofern nicht entsprechende Methoden implementiert wurden, um dies zu verhindern (Doglio, 2018).

2.2.4 Einheitliche Schnittstelle

Eines der wichtigsten Merkmale und Pluspunkte von REST im Vergleich zu Alternativen ist der Zwang zu einheitlichen Schnittstellen. Durch die Definition einer standardisierten Schnittstelle für die Dienste, welcher der Server anbietet, wird die Implementierung unabhängiger Clients vereinfacht (Doglio, 2018). Laut Fielding müssen folgend Bedingungen erfüllt sein, um dem Standard einer einheitlichen Schnittstelle zu entsprechen:

Identifizierung der Ressourcen

Bei Rest werden jegliche Inhalte als Ressource bezeichnet z.B. Text Dateien, HTML Seiten, Bilder oder Videos. Jede Ressource benötigt einen URI (Uniform Ressource Identifier) über welchen darauf zugegriffen werden kann (Massé, 2021). So könnte z.B. eine API über den URI https://api.com/users eine Liste aller User zurückgeben. Über den URI https://api.com/users/{id} könnte auf Informationen zu einem einzelnen User mit einer bestimmten ID zugegriffen werden. URIs sollen immer Nomen enthalten und so selbstbeschreibend wie möglich sein.

Manipulation von Ressourcen durch Darstellungen

Für unterschiedliche Clients kann dieselbe Ressource unterschiedlich dargestellt werden. Ein Dokument kann zum Beispiel für einen Webbrowser als HTML dargestellt werden, für ein automatisiertes Programm dagegen als JSON. Das passiert bei REST APIs mit HTTP über einen vom Client mitgesendeten Header, in welchem angegeben ist, welchen Response Type er erwartet. Dies wird als Content Negotiation bezeichnet (Massé, 2021).

Selbstbeschreibende Nachrichten

Der Client schreibt den gewünschten Zustand der Ressource in seine Request Message. Der Server schickt daraufhin eine Antwortnachricht mit dem aktuellen Zustand der Ressource. Zum Beispiel kann Client eine Request Message verwenden, um eine vom Server verwaltete Web-Seite zu aktualisieren. Es ist jedoch die Entscheidung des Servers, den Request des Clients anzunehmen oder abzulehnen, der Client erhält jedoch immer eine Response Message. Die ausgetauschten Nachrichten enthalten Metadaten über den Zustand der Ressource, Darstellungsformat etc. und werden daher als selbstbeschreibend bezeichnet (Massé, 2021).

Hypermedia

Die Darstellung einer Ressource enthält Links zu verwandten Ressourcen. Links ermöglichen Benutzer*innen, Anwendungen auf sinnvolle und gezielte Weise zu durchqueren. Das Vorhandensein oder Fehlen eines Links auf einer Seite ist ein wichtiger Bestandteil des aktuellen Zustands der Ressource (Massé, 2021).

2.2.5 Mehrschichtige Systeme

REST wurde bereits im Hinblick auf das Internet entwickelt. Eine Architektur, die den REST Richtlinien folgt, muss daher dazu in der Lage sein auch mit massivem Datenverkehr einwandfrei zu funktionieren. Um dies zu erreichen, wird eine Gliederung der Komponenten in hierarchische Schichten vorgenommen. Jede Schicht soll nur die darunter liegende verwenden und gibt ihre Informationen an die höherliegende weiter. So kann die höchste Schicht, in welcher die Businesslogik implementiert wird, nicht direkt auf jene zugreifen, auf welcher die physische Speicherung der Daten passiert. Dies führt zu einer Entkoppelung und die Gesamtkomplexität des Systems wird verringert. Als Nachteil ist zu nennen, dass aufgrund der Interaktionen zwischen den verschiedenen Schichten unerwünschte Latenzzeiten entstehen können (Doglio, 2018).

2.2.6 Code-on-demand

Im Web wird häufig Code-on-demand eingesetzt, dadurch wird es Webservern temporär ermöglicht ausführbare Programme wie Skripte oder Plug-ins an Clients zu übertragen. Code-on-demand führt zu einer technologischen Kopplung zwischen Server und Client, da der Client in der Lage sein muss, den vom Server heruntergeladenen Code zu verstehen und auszuführen. Aus diesem Grund ist Code-on-demand die einzige der sechs Architektur Eigenschaften von REST, welche optional ist. Im Webbrowser gehostete Technologien wie Java-Applets, JavaScript und Flash sind ein Beispiel für die Code-on-demand (Massé, 2021).

2.2.7 REST mit HTTP

Wird REST mit HTTP verwendet können HTTP Implementierungen für REST genutzt werden. Hat man beispielsweise den URI https://api/v1/persons können HTTP Methoden genutzt werden, um verschiedene Aktionen auf eine REST Ressource auszuführen (Sharma, 2021). Die häufigsten Methoden sind dabei:

- GET: Daten von einer bestimmten Ressource anfordern (Allen, 2017). Bei obengenannter URI würde man dadurch alle Personen zurückerhalten.
- POST: Wird verwendet, um Daten an einen Server zu senden und eine Ressource zu erstellen (Allen, 2017). Wird POST auf den URI https://api/v1/persons angewandt kann eine neue Person erstellt werden
- PUT: Mit PUT werden Daten an einen Server gesendet um eine Ressource zu aktualisieren (Allen, 2017). Mit PUT könnte bei unser URI eine bestehende Person aktualisiert werden.
- DELETE: Löscht die angegebene Ressource (Allen, 2017). DELETE ist selbsterklärend und löscht eine bestehende Person.

Um dem Client mitzuteilen, ob sein Request erfolgreich war, werden HTTP Response Codes verwendet (Sharma, 2021). Diese lassen sich in fünf Kategorien einteilen, abhängig davon, ob es

sich um einen Client oder Servererror handelt, der Request erfolgreich war oder lediglich eine Information an den Client geschickt werden soll. FÜR REST Applikationen häufig genutzte HTTP Status Codes sind in der Tabelle 1 ersichtlich.

Code	**Grund**	**Beschreibung**
200	OK	Request konnte korrekt verarbeitet werden
400	Bad Request	Der Server konnte den Request nicht verstehen. Es wurden vermutlich falsche Daten vom Client gesendet.
403	Forbidden	Der Server hat den Zugriff auf die Ressource nicht erlaubt. Der Client ist nicht eingeloggt oder nicht berechtigt.
404	Not Found	Der Server konnte die angeforderte Ressource nicht finden.
500	Internal Server Error	Bei der Verarbeitung des Requests ist ein Fehler am Server aufgetreten. Meist liegt dies an fehlerhafter Programmierung
503	Service Unavailable	Der Server wird den Request aktuell nicht bearbeiten. Dies könnte z.B. bedeuten, dass der Server überlastet ist.

Tabelle 1: HTTP Response Codes (Allen, 2017)

2.3 JSON

JavaScript Object Notation (JSON) ist heute das beliebteste Format für API Kommunikation und wird von den meisten APIs unterstützt (Yudin, 2020). Wie HTTP als Transportprotokoll, ist JSON keine Vorgabe als Format für REST APIs, ist aber de facto der Standard. Ein valides JSON Objekt besteht aus Schlüssel-/ Wert-Paaren. Dabei können Schlüssel einzelne Werte oder Arrays mit mehreren Werten enthalten (Marrs, 2017). In Abbildung 1 ist die Syntax eines JSON Dokuments ersichtlich, welches verwendet werden könnte, um mithilfe eines POST Requests gegen den Endpunkt https://api/v1/persons eine neue Person anzulegen. Dabei werden für das Alter und den Namen jeweils einzelne Werte und für die Adressen ein Array mit mehreren Adressen übermittelt.

```
1  {
2      "age": 28,
3      "name": "Lukas",
4      "addresses":["address 1", "address 2"]
5  }
```

Abbildung 1: JSON Dokument

JSON bietet für Entwickler*innen viele Vorteile. Es kann gut von Menschen gelesen werden da nur Schlüssel-/ Wert-Paare und keine zusätzlichen Metainformationen in Maschinensprache enthalten sind. Durch den simplen Aufbau kann ein JSON Dokument auch ohne den Einsatz von speziellen Tools leicht selbst geschrieben und geparst werden. Um JSON mit HTTP für REST APIs zu nutzen, müssen die Daten im JSON Format im HTTP Body mitgesendet werden und als Content-Type application/json angegeben werden (Smith Ben, 2015).

Auch im praktischen Teil dieser Arbeit wird das JSON Format verwendet, um mit der REST API zu kommunizieren.

3 CAAS

Ziel des Kapitels ist es, die theoretischen Grundlagen hinter CaaS näher zu erläutern und hierbei auch den Begriff des dafür benötigten Containers sowie Docker zu erklären.

3.1 Container vs. virtuelle Maschine

Um den Begriff Container-as-a-Service erklären zu können ist es wichtig ein Verständnis für den Begriff Virtualisierung zu schaffen und wie sich hierbei ein Container von der schon länger genutzten Technologie der Virtuellen Maschine (VM) abgrenzt.

Unter Virtualisierung versteht man die Abstraktion der Hardware von der Software. Eines der größten Anwendungsgebiete ist hierbei die Virtualisierung des Betriebssystems mit virtuellen Maschinen (Kappel et al., 2009). Diese sind seit einigen Jahren der Standard, wenn es darum geht, einen Server zur Ausführung einer Anwendung einzurichten (Buchanan et al., 2020). VMs lösen das Problem, dass typischerweise auf Servern aus Sicherheits- oder Performance Gründen nur eine einzige Applikation installiert werden darf und somit eine 1:1:1 Relation zwischen Hardware, Betriebssystem und Applikation besteht. Diese Art der Architektur ist nicht flexibel und ineffizient, da viele Anwendungen nur einen kleinen Prozentsatz der physischen Ressourcen nutzen, was dazu führt, dass die physischen Server bei weitem nicht ausgelastet sind. Da die Hardware immer besser wird, wird die Kluft zwischen den vorhandenen Ressourcen und den oft geringen Anforderungen immer größer (Fitzhugh, 2014).

Die virtuelle Maschine ist hierbei eine Software, die ihr eigenes Betriebssystem und Anwendungen so ausführen kann, als wäre sie ein physischer Computer. Sie verhält sich dabei exakt gleich wie ein physischer Computer und besitzt eine eigene virtuelle CPU, RAM, Festplatte und Netzwerkkarte. Betriebssysteme und andere Geräte können den Unterschied zwischen einem virtuellen und einem physischen Computer nicht erkennen. Dennoch besteht die virtuelle Maschine vollständig aus Software und hat keinerlei Hardwarekomponenten (Kappel et al., 2009).

Jeder virtuellen Maschine wird ein Teil der physischen Hardware des Host Geräts zugewiesen. Dabei ist jede dieser VMs von den anderen isoliert und interagiert mit der zugrunde liegenden Hardware über eine dünne Softwareschicht, die als Hypervisor bezeichnet wird. Dies unterscheidet sich grundlegend von einer physischen Architektur, bei der das installierte Betriebssystem direkt mit der Hardware interagiert (Fitzhugh, 2014).

Bei Container handelt es sich um die Kapselung einer Anwendung mit ihren Abhängigkeiten. Auf den ersten Blick scheinen sie nur eine abgewandelte Form der virtuellen Maschine zu sein. Auch ein Container ist eine vollständig von anderen Containern isolierte Instanz (Mouat, 2016). Im Gegensatz zu VMs sind Container jedoch leichtgewichtig, da sie nicht über ein eigenes Betriebssystem verfügen. Host und Container teilen sich einen Betriebssystemkern, sowie Hard- und Software Ressourcen. Anstelle des Hypervisors tritt eine Container Engine und die zusätzliche Schicht des Gast-Betriebssystems entfällt, dies ist in Abbildung 2 ersichtlich. Da der Container eine oder mehrere Applikationen mit allen Abhängigkeiten enthält, kann er ohne

großen Aufwand an andere Benutzer*innen weitergegeben oder in der Cloud bereitgestellt werden. Gleichzeitig wird der Verwaltungsaufwand minimiert, da im Gegensatz zur VM nur ein einziges Betriebssystem betreut werden muss (Yadav et al., 2019).

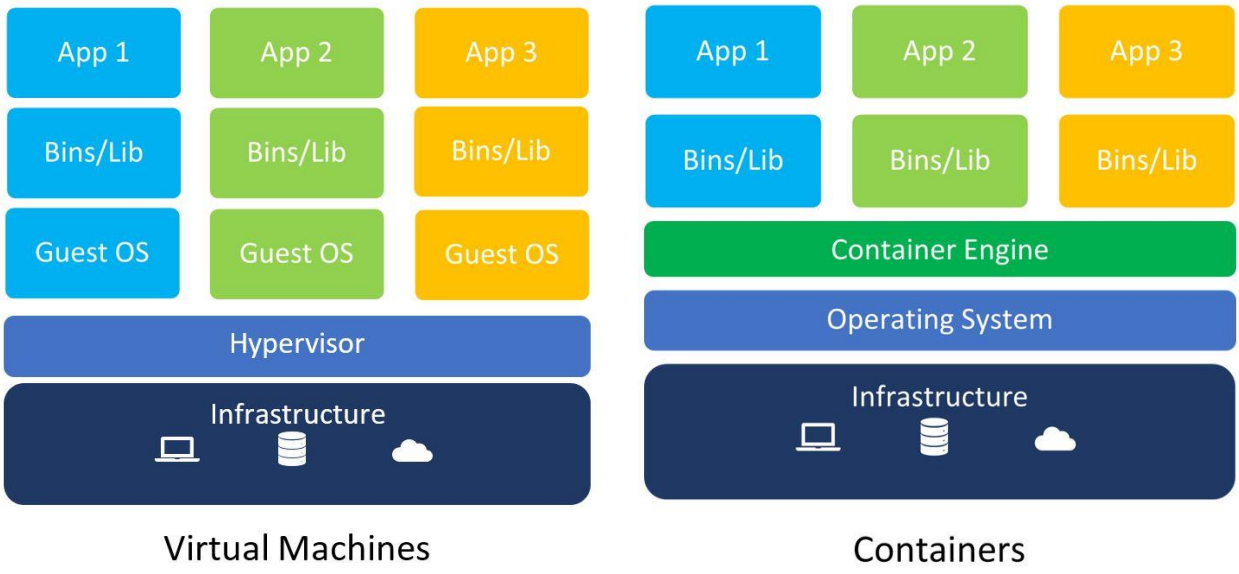

Abbildung 2: VM vs. Container

VMs und Container haben also einen unterschiedlichen Zweck. VMs sollen eine Umgebung, z.B. ein Betriebssystem vollständig emulieren, während ein Container zum Ziel hat, eine Anwendung portabel und in sich geschlossen zur Verfügung zu stellen (Mouat, 2016).

Container haben für die Paketierung von Anwendungen einige Vorteile:

- Geschwindigkeit: Docker-Container sind schneller als virtuellen Maschinen. Das Hochfahren eines Containers kann zwischen ein paar Millisekunden bis zu ein paar Sekunden dauern, während eine virtuelle Maschine in der Regel einige Minuten zum Hochfahren benötigt.
- Portabilität: Container können geteilt und gemeinsam genutzt werden. Die Anwendung verhält sich überall gleich, unabhängig davon wo der Container läuft. Dadurch werden menschliche Fehler und potenzielle Abhängigkeiten von der Umgebung minimiert (Buchanan et al., 2020).
- Dadurch, dass für Container nur wenige Ressourcen benötigt werden können viele Container gleichzeitig ausgeführt werden, um beispielsweise die Produktionsumgebung zu simulieren.
- Container haben auch Vorteile für Endbenutzer*innen da diese Anwendungen herunterladen können, ohne sich um Konfigurations- oder Installationsprobleme sorgen machen zu müssen (Mouat, 2016).

3.2 Docker

Docker ist eine Software, um Applikationen in Container zu verpacken und bereitzustellen. Obwohl es Alternativen gibt, ist Docker für diese Aufgabe die bekannteste Technologie in diesem Bereich (Davis, 2021).

Um Docker zu nutzen, werden sogenannte Docker Images benötigt. Ein Docker Image ist eine Paketierungseinheit, die alles enthält, was für die Ausführung einer Anwendung erforderlich ist. Es beinhaltet den Code der Anwendung sowie alle Abhängigkeiten. Besitzt man das Docker Image wird nur noch Docker selbst benötigt, um die Anwendung auszuführen (McKendrick, 2020). Images sind Architekturtechnisch ähnlich aufgebaut wie Linux. In Linux ist alles eine Datei. Das gesamte Betriebssystem ist im Grunde ein Dateisystem mit Dateien und Ordnern, welche auf der lokalen Festplatte gespeichert sind. Ein Image besteht im Grunde aus einer Gruppe von Dateien, die zu einer einzigen Datei zusammengefasst werden und einem Dateisystem (Schenker, 2020).

Um ein Docker Image zu erstellen, gibt es zwei Möglichkeiten. Die erste Möglichkeit ist das Erstellen über die Kommandozeile. Diese Methode kann nützlich sein, wenn es um die Erstellung von Prototypen oder Machbarkeitsstudien geht. Jedoch gibt es hierbei einen gravierenden Nachteil, da es sich um einen manuellen Prozess handelt und jedes neuen Images angepasst werden muss sind die Ergebnisse möglicherweise nicht wiederholbar und nicht skalierbar. Durch die manuelle Ausführung durch Entwickler*innen ist diese Methode auch fehleranfällig. Die zweite Möglichkeit zum Erstellen eines Images ist über ein Dockerfile. Ein Dockerfile ist eine Textdatei welche Anweisungen für die Erstellung eines Container Images enthält, es handelt sich dabei also um eine deklarative Art Images zu erstellen (Schenker, 2020). Wie ein Dockerfile aufgebaut sein kann, ist in Listing 1 ersichtlich.

```
FROM python:2.7
RUN mkdir -p /app
WORKDIR /app
COPY ./requirements.txt /app/
RUN pip install -r requirements.txt
CMD ["python", "main.py"]
```

Listing 1: Dockerfile

Dabei handelt es sich um Befehle, welche sequenziell abgearbeitet werden:

1. Mit welchem Image Template soll das Image erstellt werden -> python 2.7
2. Erstellt im Container ein neues Verzeichnis „app“
3. Setzt das aktuell verwendete Verzeichnis des Containers auf „app“
4. Kopiert die Datei requireents.txt vom Host in das Verzeichnis „app“ auf den Container
5. Installiert die in der Datei requirments.txt angegeben Packages
6. Führt das Python Skript „main.py“ aus

Durch das Dockerfile wird der Prozess zum Erstellen des Images automatisiert. Die Docker Engine führt die im Docker File enthaltenen Befehle automatisch aus und erstellt das Image (Charge, 2020).

Docker Container sind also nichts anderes als die laufende Instanz eines Docker Images. Während Images schreibgeschützte Dateien sind, können Benutzer*innen mit Containern interagieren und arbeiten und administrierende diese verwalten und anpassen (Charge, 2020).

Um Docker Images außerhalb des lokalen Gerätes zu verteilen und anderen Benutzer*innen Zugriff zu ermöglichen werden sogenannte Image Registries verwendet. Dabei handelt es sich um ein System zur zentralen Speicherung und Bereitstellung von Inhalten, das Docker Images enthält, die in verschiedenen Versionen zur Verfügung gestellt werden. Das bekannteste hierbei ist Docker Hub (Poulton, 2020). Im praktischen Teil dieser Arbeit wird das Elastic Container Registry (ECR) genutzt. Dabei handelt es sich um den Registry Dienst von AWS. Die Entscheidung gegen Docker Hub fußt darauf, dass im praktischen Teil nur AWS Services verwendet werden.

3.3 AWS Fargate

Nachdem ein Image für einen Container erstellt wurde, wird die entsprechende Infrastruktur benötigt, um den Container bereitzustellen. Eine Möglichkeit hierzu bietet CaaS. CaaS ist ein Geschäftsmodell, bei dem Anbieter von Cloud-Computing-Diensten Container-basierte Virtualisierung als skalierbaren Online-Dienst anbieten. Dies ermöglicht es den Nutzern*innen, Containerdienste zu nutzen, ohne über die erforderliche Infrastruktur zu verfügen. Es handelt sich um einen Marketingbegriff, um sich neben etablierten Cloud-Service-Modelle wie Infrastructure-as-a-Service (IaaS), PaaS und Software-as-a-Service (SaaS) zu positionieren (McKendrick, 2020).

Für CaaS gibt es viele verschiedene Anbieter, im praktischen Teil wird jedoch der in der AWS Cloud enthaltene Service genutzt, welcher sich AWS Fargate nennt. AWS Fargate ist ein CaaS Angebot von Amazon welcher im Elastic Container Service (ECS) enthalten ist. ECS bietet zwei verschiedene Modi zur Bereitstellung von Containern an. Einmal die „alte“ kompliziertere Methode, bei welcher eine virtuelle Maschine aufgesetzt werden muss und der Container in der virtuellen Maschine betrieben wird. Und, seit Ende 2017, der neue Service AWS Fargate. Dadurch wird es ermöglicht, Container in AWS auszuführen, ohne dass darunterliegende Infrastruktur konfiguriert werden muss, wodurch eine schnelle Bereitstellung der Container ermöglicht wird (McKendrick, 2020).

3.3.1 Task Definition

Benutzer*innen müssen Fargate ein Docker Image zur Verfügung stellen. Die passiert über die sogenannte Task Definition (Ifrah, 2019). Die wichtigsten Parameter, welche in der Task Definition spezifiziert werden können, sind:

- Anzahl der Container und welches Docker Image diese verwenden
- CPU und RAM, welche für den Task zur Verfügung stehen und wie diese auf einzelne Container aufgeteilt werden sollen
- Logging Konfigurationen
- Wie sich der Task verhalten soll, wenn ein Container abstürzt
- Welche Befehle der Container ausführen soll, wenn er gestartet wird (Amazon ECS task definitions - Amazon Elastic Container Service, o. D.)

Technisch gesehen kann man in ECS keine Container direkt laufen lassen, sondern nur Tasks, welche ihrerseits einen oder mehrere Container enthalten. Es ist dabei nicht zwingend notwendig in der Task Definition Speicher und Rechenleistung anzugeben. Werden diese Parameter nicht übergeben, findet eine automatische Skalierung statt und es werden immer so viele Ressourcen zugeteilt wie gerade benötigt werden. Task Definitions können im JSON Format bereitgestellt werden.

```
"family": "mywebsite",
"networkMode": "awsvpc",
"cpu": "256",
"memory": "512",
"requiresCompatibilities": ["EC2"],
"containerDefinitions":
    "name": "mywebsite-nginx",
    "image": "nginx:latest",
    "essential": true,
    "cpu": 128,
    "memory": 256,
    "memoryReservation": 128,
    "linuxParameters":
      "maxSwap": 512,
      "swappiness": 50
```

Listing 2: Task definition

In Listing 2 ist der Aufbau einer Task Definiton ersichtlich. Hierbei wird ein Task „mywebsite“ erstellt welchem 256 CPU Einheiten und 512 MB RAM zugeteilt werden. Innerhalb des Tasks wird ein Container „mywebsite-nginx“ definiert, welchem 128 CPU Einheiten und 256 MB RAM zugeteilt werden. Der Container basiert auf dem Docker Image „nginx:latest“ (How Amazon ECS Manages CPU and Memory Resources, 2020). Durch Fargate verringert sich der Aufwand der Bereitstellung etwa um das zehnfache gegenüber der alten Methode mit virtuellen Maschinen (Ifrah, 2019).

Neben den Task Definitions gibt es noch die sogenannten ECS Cluster. Ein ECS Cluster ist eine logische Gruppierung von Tasks und Services. Tasks und Services werden im Hintergrund auf einer Infrastruktur ausgeführt, welche für einen bestimmten Cluster registriert ist. Diese Cluster sind zwingend erforderlich. Typischerweise gibt es bei Softwareprojekten eine Development-, Test- und Produktivumgebung. Dies kann bei ECS durch das Erstellen von drei verschiedenen Clustern abgebildet werden (Amazon ECS clusters - Amazon Elastic Container Service, o. D.).

Zusammengefasst muss mit Fargate zwar nicht mehr die Infrastruktur bereitgestellt und konfiguriert werden dennoch fällt ein Verwaltungsaufwand für Task Definitionen und Container an. Wird neuer Sourcecode geschrieben muss ein neues Docker Image erstellt werden und die Container in AWS Fargate aktualisiert werden. Obwohl dieser Vorgang Großteils automatisierbar ist, handelt es sich dennoch um einen potenziell fehleranfälligen Prozess. Außerdem muss immer noch ein ECS Cluster verwaltet werden, was vor allem für Entwickler*innen ohne Erfahrung in diesem Bereich kompliziert und aufwändig sein kann. Infolgedessen müssen also immer noch viele Entwicklungsressourcen für Konfiguration und Verwaltung eingeplant werden. Einfacher wäre die Möglichkeit, Code direkt in der Cloud auszuführen (Sbarski, 2017). Diese Alternative wird im vierten Kapitel näher beschrieben.

3.3.2 Kosten

Die Preisgestaltung für Container, welche in AWS Fargate laufen ergibt sich anhand der verbrauchten Ressourcen und es fallen keine Vorabkosten an. Die Kosten, welche für Mitteleuropa (genannt Region „Europe (Frankfurt)“ anfallen sind in Tabelle 2 ersichtlich.

Einheit	**Architektur**	**Preis**
Pro vCPU pro Stunde	Linux/X86	€ 0,041
Pro GB RAM pro Stunde	Linux/X86	€ 0,0045
Pro vCPU pro Stunde	Linux/ARM	€ 0,033
Pro GB RAM pro Stunde	Linux/ARM	€ 0,0036

Tabelle 2: AWS Fargate Preise (Serverless Compute Engine–AWS Fargate Pricing–Amazon Web Services, o. D.)

Die Preise werden pro Sekunde berechnet mit einem Verrechnungsminimum von einer Minute. Die Verrechnung beginnt, sobald der Task zu laufen beginnt und endet mit dessen Terminierung und wird auf die nächste volle Sekunde aufgerundet (Serverless Compute Engine–AWS Fargate Pricing–Amazon Web Services, o. D.). Außerdem können zusätzliche Gebühren anfallen, wenn andere AWS Services wie etwa CloudWatch zum Monitoring der Applikation verwendet werden.

4 FAAS

Im folgenden Kapitel werden die theoretischen Grundlagen zum FaaS Modell näher erläutert.

4.1 Initialisierung

Mit FaaS stellen wir unseren Code als unabhängige Funktionen oder Operationen bereit und konfigurieren diese Funktionen so, dass sie aufgerufen oder ausgelöst werden, wenn ein bestimmtes Ereignis oder eine Anfrage innerhalb der FaaS-Plattform auftritt. Faas ist also im Gegensatz zu CaaS vollständig eventgesteuert und wird nur ausgeführt, wenn es aufgerufen wird, wodurch es keine Leerlaufzeiten gibt. Die Plattform selbst ruft unsere Funktionen durch die Instanziierung einer dedizierten Umgebung für jedes Ereignis auf - diese Umgebung besteht aus vollständig von der Plattform verwalteten, leichtgewichtigen virtuellen Maschine oder Container, der FaaS-Laufzeitumgebung und unserem Code. Resultierend daraus müssen wir uns keine Gedanken um die Laufzeitverwaltung unseres Codes machen. Diese virtuelle Maschine bzw. Container existiert nur so lange wie sie benötigt werden und werden nach dem Funktionsaufruf und einer Wartefrist von etwa 10 Minuten zerstört (Chapin & Roberts, 2020).

Ein Hindernis für die Adaption von FaaS stellt der sogenannte cold start dar. Nachdem bei FaaS nur für jene Zeit gezahlt werden muss, welche die Funktion zur Ausführung benötigt werden, werden am Ende der Ausführung alle der Funktion zugewiesenen Ressourcen freigegeben. Um jedoch auf künftige Anfragen zu reagieren, müssen der Funktion erneut Ressourcen zugewiesen werden und der Container im Hintergrund erstellt und gestartet werden. Die Dauer dieser Zeitspanne wird als cold start delay bezeichnet. Dies ist vor allem kritisch bei verzögerungsempfindlichen Anwendungen wie Smart Health Anwendungen im Internet of Things Bereich, welche in Echtzeit Informationen zum Zustand des Patienten liefern müssen (Vahidinia et al., 2020). Aus Performancegründen fahren FaaS Provider den Container nicht sofort herunter. Nachfolgende Requests profitieren also von bereits vorhandenen Containern und haben so eine kürzere Laufzeit, dies wird als warm start bezeichnet (Manner et al., 2018).

4.2 AWS Lambda

AWS Lambda ist die Bezeichnung der FaaS Plattform von Amazon welche 2014 eingeführt wurde und seitdem ständig um neue Funktionen erweitert wurde. Lambda bietet ein außergewöhnlich einfaches Programmier- und Deployment Modell. Um eine einzelne Funktion bereitzustellen, wird lediglich der Code als ZIP oder JAR Datei benötigt. Die Laufzeitumgebung wird dabei vollständig von Lambda verwaltet (Barid et al. ,2017) Lambda erfüllt dabei alle Kriterien an einen Serverless Service, welche in Kapitel 2.1 genannt wurden:

- Erfordert keine Verwaltung des Host Systems

  Mit Lambda sind wir vollständig von dem zugrunde liegenden Host abstrahiert, auf dem unser Code ausgeführt wird. Außerdem müssen wir keine langlebige Anwendung verwalten. Sobald unser Code die Verarbeitung ein bestimmtes Ereignis abgeschlossen hat, kann AWS die Laufzeitumgebung beenden.

- Automatische Skalierung abhängig von der Anzahl der Aufrufe

  Dies ist einer der Hauptvorteile von Lambda - Ressourcenmanagement und Skalierung sind völlig transparent. Sobald wir unseren Funktionscode hochgeladen haben, erstellt die Lambda-Plattform gerade genug Instanzen der Funktion, um die Last zu einem bestimmten Zeitpunkt zu bewältigen. Wenn nur eine Instanz benötigt wird, wird auch nur diese erstellt. Werden andererseits hunderte Umgebungen benötigt skaliert Lambda schnell und ohne Aufwand für Benutzer*innen

- Kosten richten sich je nach Verbrauch

  AWS berechnet für Lambda nur die Zeit, in der unser Code pro Umgebung ausgeführt wird, aufgerechnet auf die nächste Millisekunde. Wenn unsere Funktion beispielsweise alle 5 Minuten für 200 ms aktiv ist, werden uns nur 2400 ms Nutzung pro Stunde berechnet.

- Performance wird in anderen Größen als Anzahl an Hosts angegeben

  Da wir mit Lambda vollständig vom zugrunde liegenden Host abstrahiert sind, können wir keine Anzahl oder Art der zugrundeliegenden EC2-Instanzen angeben, die verwendet werden sollen. Stattdessen legen wir nur fest, wie viel RAM unsere Funktion benötigt (bis zu maximal 10240 MB).

- Hohe Verfügbarkeit

  Wenn der Server hinter der Lambda Funktion ausfällt, startet Lambda automatisch Instanzen auf einem anderen Server. Wenn ein bestimmtes Rechenzentrum in einer Region ausfällt, startet Lambda automatisch Instanzen mit Servern eines anderen Rechenzentrums derselben Region. Erst wenn es zu einem Regionsweiten Ausfall kommt, gibt es keinen Standardmäßigen Ausfallsplan mehr und es obliegt den Kund*innen darauf zu reagieren (Chapin & Roberts, 2020).

Die Kosten für AWS Lambda setzen sich aus zwei Teilen zusammen, nämlich der Anzahl und der Dauer der Requests. Bei der Anzahl der Requests wird Benutzer*innen ein Kontigent von einer Million gratis Requests zur Verfügung gestellt. Danach ergibt sich ein Preis von 18 Cent pro eine Million Requests. Die Dauer des Requests wird ab dem Zeitpunkt berechnet, an welcher der Code mit der Ausführung beginnt, bis er zurückgegeben wird oder anderweitig beendet wird und es wird auf die nächste Millisekunde aufgerundet. Der Preis hier ist einmal vom Arbeitsspeicher abhängig, welcher der Funktion zugewiesen wurde, hier ist ein Maximum von 10240 MB möglich. Wird der Arbeitsspeicher erhöht, werden im Hintergrund auch andere Ressourcen wie CPU ebenfalls proportional erhöht. In die Preisberechnung fließt ebenfalls die für die Lambda Funktion ausgewählte Prozessor Architektur mit ein. Hier ist eine Auswahl zwischen x86 und ARM Architekturen möglich, wobei von Amazon selbst die ARM Architektur empfohlen wird. Die Preise hierfür sind in Tabelle 3 ersichtlich (AWS Lambda - Preise, o. D.)

Arbeitsspeicher (MB)	**x86**	**ARM**
128	€ 0,0000000019	€ 0,0000000015
512	€ 0,0000000073	€ 0,0000000059
1024	€ 0,000000015	€ 0,000000012
2048	€ 0,000000029	€ 0,000000024
4096	€ 0,000000059	€ 0,000000047
8192	€ 0,00000012	€ 0,000000094
10240	€ 0,00000015	€ 0,00000012

Tabelle 3: AWS Lambda Preise (AWS Lambda - Preise, o. D.)

Die Kosten sind insgesamt schwerer vorherzusagen als bei CaaS, da nicht immer abschätzbar ist, wie lange Funktionen ausgeführt werden. Ein weiteres Problem ergibt sich durch Programmierfehler. Landen Funktionen in einer Endlosschleife, ist es möglich das durch jeden neuen Request eine neue Funktion instanziiert wird, die wieder in der Endlosschleife gefangen ist. Da für die Dauer der Funktion gezahlt wird, können so sehr hohe Kosten anfallen. Daher ist es notwendig, da man solche Fehler nicht immer ausschließen kann, auch ein entsprechendes Monitoring zu implementieren. Generell muss im Vergleich zu CaaS noch mehr Fokus auf Code Optimierung gelegt werden, da sich schon durch einzelne Funktionen, wenn ihre Laufzeit von z.B. 35 ms auf 2 ms gekürzt werden kann, große Preisunterschiede ergeben können, wenn die Funktion oft genug aufgerufen wird (Eivy & Weinman, 2017).

Außerdem fallen bei einer REST API noch Kosten für den Service API Gateway an, welcher im nächsten Kapitel beschrieben wird.

4.3 API Gateway

Ein API Gateway wird von allen gängigen Cloudanbietern angeboten und ist ein Service zum Erstellen, Veröffentlichen, Verwalten, Überwachen und Sichern von REST-, HTTP- und WebSocket-APIs. Der entsprechende AWS Service nennt sich Amazon API Gateway (What is Amazon API Gateway?, o. D.). Obwohl es möglich ist, eine REST API mit Lambda ohne das API Gateway zu entwickeln, ist eine Kombination aus beiden die häufigste Umsetzungsmethode. Viele Gründe sprechen dafür Benutzer*innen mit dem API Gateway, statt direkt mit der Lambda Funktion kommunizieren zu lassen. Der vielleicht wichtigste Grund ist, dass sicherheitsrelevante Themen wie Authentifizierung, Autorisierung und Zertifikatsverwaltung an API Gateway ausgelagert werden können. Hierbei sind bereits viele Sicherheitsanforderungen implementiert und Entwickler*innen können sich so auf die Geschäftslogik konzentrieren, anstatt diese neu zu entwickeln. Dem AWS API Gateway ist noch ein Service namens Amazon CloudFront vorgeschaltet. Dieser filtert Requests und funktioniert als ein Loadbalancer. Durch CloudFront ist auch ein Schutz gegen DDoS Attacken gegeben, da bei einem Angriff verdächtige Requests erkannt werden und einfach nicht mehr an das API Gateway weitergeleitet werden (Patterson, 2019). Abbildung 3 zeigt ein Architekturschaubild für Requests mit CloudFront, API Gateway und Lambda.

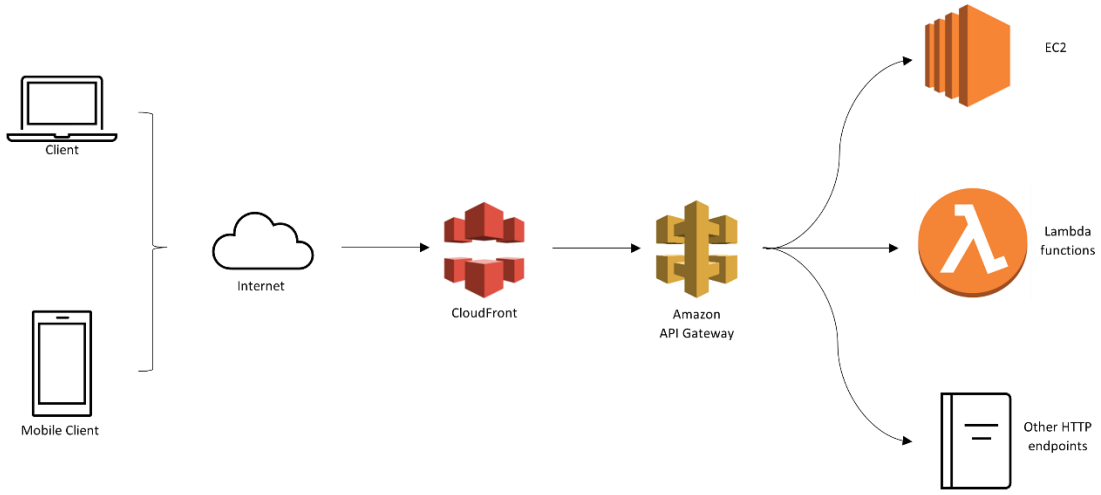

Abbildung 3: FaaS Beispielarchitektur mit API Gateway

Weitere Vorteile sind eine effiziente API-Entwicklung und vorkonfigurierte Überwachungsmöglichkeiten. Durch das API Gateway wird die Implementierung der Lambda Funktionen vom Client abstrahiert und entkoppelt. Da der Client nur das API Gateway nicht aber die Funktionen dahinter kennt, können Funktionen einfach ausgetauscht werden (Patterson, 2019). Es können außerdem unterschiedliche Versionen derselben API ausgeführt werden. Hiermit ist beispielsweise eine Aufteilung in Test und Produktivumgebung möglich und neue Versionen können schneller getestet und freigegeben werden. Zur Überwachung bietet das API Gateway Dashboard eine Vielzahl an Leistungsmetriken, sowie detaillierte Informationen zu API Aufrufen, Latenzzeiten und Fehlern (Amazon API Gateway, o. D.).

Auch beim API Gateway gibt es, wie bei AWS Lambda, ein gratis Kontingent von einer Million Requests. Der Preis wird pro Million Requests abgerechnet und abhängig von der Anzahl der Aufrufe wird der Preis stufenweise geringer. Tabelle 4 zeigt die Kosten für das API Gateway (Amazon API Gateway – Preise, o. D.).

Anzahl Requests (pro Monat)	**Preis (pro Million)**
Erste 333 Millionen Aufrufe	€ 3,27
Nächste 667 Millionen Aufrufe	€ 2,82
Nächste 19 Milliarden Aufrufe	€ 2,39
Alle Aufrufe über 20 Milliarden	€ 1,72

Tabelle 4: API Gateway Preise (Amazon API Gateway – Preise, o. D.)

Um eine bessere Leistung und schnellere API Ausführung zu erreichen empfiehlt sich gerade in Kombination mit Lambda die Bereitstellung eines Cache, um Daten aus dem Zwischenspeicher abrufen zu können. Hier erfolgt die Preisberechnung auf Stundenbasis (Amazon API Gateway – Preise, o. D.). Tabelle 5 zeigt hier den jeweiligen Stundensatz für die entsprechenden Zwischenspeichergrößen. Für den Zwischenspeicher kann eine Größe zwischen 0,5 GB und 237 GB gewählt werden.

Größe des Zwischenspeichers (GB)	**Stundensatz**
0,5	€ 0,018
1,6	€ 0,034
6,1	€ 0,18
13,5	€ 0,22
28,4	€ 0,44
58,2	€ 0,88
118	€ 1,68
237	€ 3,35

Tabelle 5: API Gateway Cache Preise (Amazon API Gateway – Preise, o. D.)

Generell sehen die Preise für AWS Services auf den ersten Blick sehr niedrig aus. Gerade für umfangreiche Projekte bei welchen durch einen Seitenaufruf viele verschiedene API Aufrufe ausgelöst werden können sich die Kosten jedoch schnell summieren.

Nutzt man mehrere AWS Services wie etwa Lambda, Gateway und eine AWS Datenbank kann eine Modellierung der Kosten im Vorfeld rasch komplex und verwirrend für Benutzer*innen werden. Grund hierfür ist, dass die Zeitangabe für Kosten für alle Services unterschiedlich sein kann, so sind etwa Abrechnungszeiträume von Millisekunden, Sekunden, Stunden oder Tagen möglich. Auch wird die Leistung für bestimmte Services in MB pro Sekunde konfiguriert, aber die Abrechnung erfolgt mittels GB pro Sekunde. Jeder neue zusätzliche Service erschwert somit die Kostenvorhersage. Allgemein ist es entscheidend den aktuellen und zukünftig geplanten Umfang des Projektes zu analysieren, um ein Verständnis der Kosten zu erlangen und wie diese sich mit zukünftigem Wachstum entwickeln (Eivy & Weinman, 2017).

5 IMPLEMENTIERUNG

In diesem Kapitel finden sich sämtliche Details zur Implementierung der CaaS und FaaS APIs. Für beide Umsetzungen wurde ein AWS Account benötigt, welcher im Vorhinein erstellt wurde.

5.1 API

Die API soll die Grundfunktionalitäten für ein fiktives Unternehmen abbilden, welches eine Plattform zur Parkplatzsuche anbietet. Benutzer*innen können sich einloggen, anschließend nach Parkplätzen suchen und diese buchen. Bestehende Buchungen können von Benutzer*innen bei Bedarf wieder gelöscht werden. Außerdem können sie eigene Parkplätze zur Vermietung anlegen und diese bei Bedarf wieder löschen. Ziel war es, die API so simpel wie möglich zu gestalten, aber dennoch komplex genug, um verschiedene API Calls gleichzeitig ausführen zu können und so realistische Ergebnisse zu erhalten.

Die API wurde mit dem Model View Controller (MVC) Design Pattern implementiert. Durch das MVC Pattern soll eine Trennung zwischen den einzelnen Bereichen (Model, View und Controller) sichergestellt werden. Dadurch soll erreicht werden, dass die Darstellung von der eigentlichen Businesslogik entkoppelt ist. Folgt man diesem Ansatz, erleichtert dies das Warten und Testen der Applikation (Deinum & Cosmina, 2021). Die Aufgaben der Komponenten werden wie folgt unterteilt:

- Model: Ist dafür zuständig die Daten zu verwalten, welche von der View angezeigt werden. Es stellt auch die Schnittstelle zu einem persistenten Datenspeicher wie etwa einer Datenbank dar. Die Modellkomponente implementiert die Logik, die der Applikation das Erstellen, Lesen, Aktualisieren und Löschen von Daten ermöglicht.
- View: Darstellung des Modells für den Client. Diese kann entweder visuell, etwa als HTML, oder PDF erfolgen, es kann sich aber auch nur um einen Datenaustausch ohne grafische Darstellung, etwa im XML oder JSON Format handeln
- Controller: Der Controller ist für die Geschäftslogik der Software zuständig. Dazu gehört auch, dass der Controller mit View und Model kommuniziert. Er reagiert auf Aktionen von Benutzer*innen, etwa das Absenden eines Formulars oder das Anklicken eines Buttons und aktualisiert daraufhin Modell und View mit den neuen Daten. Im Wesentlichen enthält der Controller den Code, welcher die drei Komponenten zusammenhält (Voorhees, 2021).

Abbildung 4 visualisiert die eben beschriebenen Zusammenhänge zwischen den Komponenten.

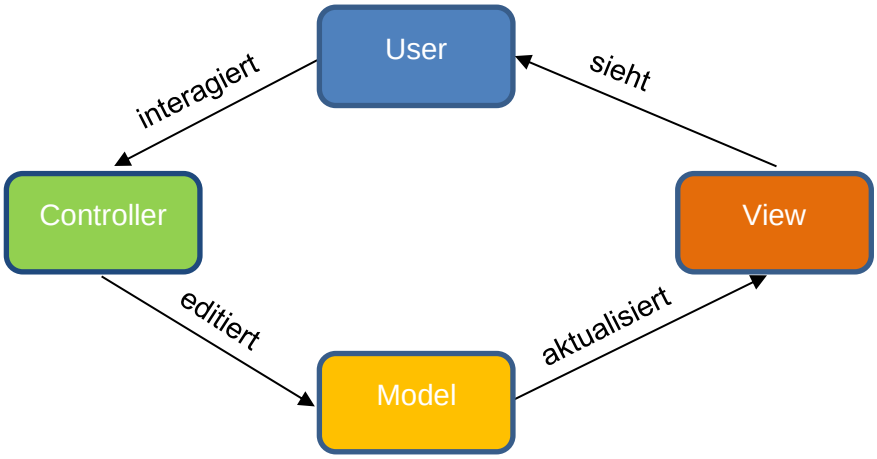

Abbildung 4: MVC Architektur

Im Fall unserer API stellen die Parkplätze und Buchungen das Modell dar. In der View erfolgt die Darstellung als JSON.

In Tabelle 6 folgt eine Auflistung aller implementierten API Endpunkte.

HTTP Methode	**Aufruf**	**Beschreibung**
GET	/parkinglots	Auflistung aller buchbaren Parkplätze.
POST	/parkinglots	Legt einen neuen Parkplatz in der Datenbank an.
PUT	/ parkinglots /id	Aktualisiert einen bestehenden Parkplatz mit den neuen Werten.
GET	/parkinglots/id	Gibt den Parkplatz mit der angegebenen ID zurück.
DELETE	/parkinglot/id	Löscht den Parkplatz mit der angegebenen ID.
GET	/bookings	Auflistung aller Buchungen der Benutzer*innen.
POST	/bookings	Legt eine neue Buchung an.
PUT	/ bookings /id	Aktualisiert eine bestehende Buchung mit den neuen Werten.
GET	/booking/id	Liefert Buchung mit der ID zurück.
DELETE	/booking/id	Löscht die angegebene Buchung.

Tabelle 6: API Endpunkte

Da durch die API nur Standard CRUD Funktionalitäten zur Verfügung gestellt werden, wird hier nicht näher auf die technische Umsetzung eingegangen.

5.2 Amazon Relational Database Service

Nachdem im Kapitel 5.1 die funktionalen Anforderungen und API Endpunkte spezifiziert wurden, musste im nächsten Schritt die Datenbank erstell und konfiguriert werden. Hier fiel die Entscheidung, die Datenbank ebenfalls in der Cloud laufen zu lassen, um die Performancemessungen von CaaS und FaaS nicht durch fehlende Ressourcen zu verzerren. Die Wahl hierfür fiel den Relational Database Service (RDS) welcher Teil von AWS ist. Mit diesem Service bietet Amazon folgende sechs relationalen Datenbanken in der Cloud an:

- Amazon Aurora
- MySQL
- Maria DB
- PostgreSQL
- Oracle
- Microsoft SQL Server

Auch hier können Ressourcen je nach Bedarf verringert, erhöht oder automatisch skaliert werden. Ein weiterer Vorteil ist das Datenbanken automatisch durch Amazon gepatcht werden und standardmäßig alle sieben Tage ein Backup erstellt wird. Sämtliche Skripts, Applikationen und Tools, welche bereits mit einer bestehenden Datenbank verwendet werden, sind auch mit Amazon RDS kompatibel (Mukherjee, 2019). In Tabelle 7 befinden sich die Einstellungen, welche zur Konfiguration der Datenbankinstanz gewählt wurden.

Parameter	**Wert**
Engine	PostgreSQL 13.3
Instanzenname	masterarbeit
vCPUs	2
RAM	1 GB
Speichertyp	Universell-SSD
Zugewiesener Speicher	20 GB

Tabelle 7: AWS RDS Einstellungen

Aus Kostengründen wurden niedrige Werte für vCPUs, RAM und Speicher gewählt, welche jedoch für die reine Entwicklung und zum Testen der Verbindung ausreichend sind. Die PostgreSQL Datenbank versucht automatisch durch den PostgreSQL Query Optimizer Datenbankabfragen zu beschleunigen. Dies geschieht, indem Teile der Datenbank in den Arbeitsspeicher geladen werden. Für große Datenbanken wird also mehr Arbeitsspeicher benötigt, um eine hohe Performance zu gewährleisten (Dombrovskaya et al., 2021). Für die Perfomance Messungen im nächsten Kapitel wurden die Werte in Relation zur Anzahl der eingefügten Datensätze erhöht.

Nach dem Erstellen der Datenbankinstanz erzeugt AWS automatisch eine URL, mit dieser und dem gewählten Benutzernamen und Passwort können Applikationen auf die Datenbank zugreifen. Um die Verbindung zur Datenbank herzustellen, wurde anschließend noch die Sicherheitsregel für den eingehenden Datenverkehr der virtuellen Netzwerkumgebung, wie in Abbildung 5 dargestellt, abgeändert. Als erlaubte Quelle wurde der IP Bereich 0.0.0.0/0 festgelegt, was bedeutet, dass jegliche IP-Adressen erlaubt sind. Dieses Setting wurde gewählt, um keine eignen Netzwerke und Sicherheitsrichtlinien für die CaaS und FaaS Applikationen abbilden zu müssen und sollte niemals für produktive oder anderweitig kommerziell genutzte Datenbanken gewählt werden. In der Praxis sollte der IP Bereich je nach Anwedungsszenario angepasst werden und so klein wie möglich gehalten werden.

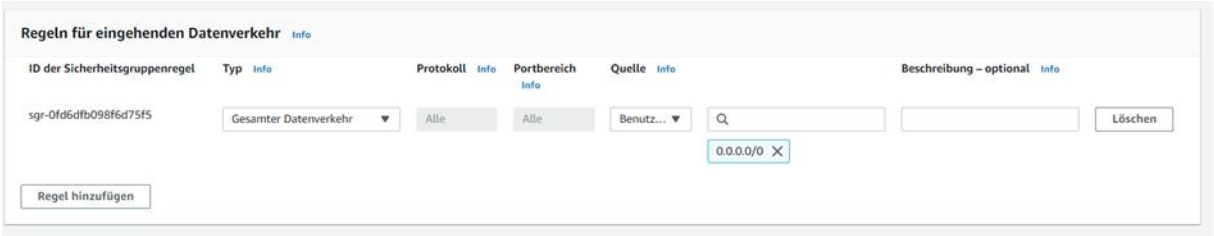

Abbildung 5: RDS Sicherheitsregel

5.3 CaaS

Im Rahmen der Anforderungsanalyse wurden, die in Tabelle 8 ersichtlichen Technologien und Services identifiziert, welche für die CaaS Umsetzung benötigt werden. Um die Performance der Umsetzungen vergleichen zu können, wurde wie bei der FaaS API JavaScript als Programmiersprache gewählt.

Parameter	**Wert**
Programmiersprache	JavaScript
Language runtime	Node.js 14.18
Datenbank	PostgreSQL 13.3
CaaS Plattform	AWS Fargate
Request Routing	Elastic Load Balancer API Gateway
Region	Europe (Frankfurt)

Tabelle 8: CaaS Technologien

Aufgrund der Wahl auf die in Tabelle 8 gelisteten Technologien ergibt sich die in Abbildung 6 gezeigte Architektur für die CaaS Umsetzung. Durch den User wird ein JSON Request an das API Gateway gesendet. Dieses ruft wiederum den Elastic Load Balancer auf, welcher auf den entsprechenden Container in Fargate weiterleitet. Fargate erhält das Container Image durch Zugriff auf ein Repository, welches in ECR angelegt wurde. Der Fargate Container greift beim Aufruf der API auf die in der Postgres Datenbank gespeicherten Daten zu. Schließlich wird die Response im JSON Format durch das API Gateway wieder an den User zurückgesendet. Da Lambda automatisch HTTPS nutzt, war es notwendig, um die Vergleichbarkeit zu gewährleisten, auch die CaaS Umsetzung mit HTTPS zu implementieren. Durch den Overhead des HTTPS Protokolls bei der Verschlüsselung kommt es zu Perfomance Einbußen. In der Praxis ist schwer zu sagen, wie groß der Unterschied tatsächlich ist, jedoch wird die HTTP Lösung bei ansonsten gleichen Bedingungen immer performanter sein (Naylor et al., 2014). Es ist dabei nicht möglich den Container in Fargate direkt mit HTTPS zu nutzen. Dies ist der Grund, warum ein Elastic Load Balancer, vom Typ Application Load Balancer (ALB) genutzt werden muss. Mit dem ALB ist es möglich einen HTTPS-Listener zu erstellen, welcher verschlüsselte Verbindungen erzwingt. Diese Funktion ermöglicht die Verschlüsselung des Datenverkehrs zwischen dem ALB und den Clients, die SSL- oder TLS-Sitzungen initiieren. Beim Erstellen des HTTPS-Listeners, muss ein SSL/TLS-Zertifikat bereitgestellt werden. Der ALB verwendet dieses Zertifikat, um ausgehenden Datenverkehr zu verschlüsseln und eingehenden Datenverkehr zu entschlüsseln. Die interne Datenübertragung zwischen ALB und der API im Fargate Container erfolgt dabei immer noch unverschlüsselt. Dies ist für die meisten Applikationen, sofern es keine externen Regulierungen oder sonstiges gibt, ausreichend (Mallu & Alvarez-Parmar, 2020).

Im Rahmen der Implementierung kam es bei ALB jedoch zu Problemen mit einem selbst signierten SSL Zertifikat. Um keine Domäne kaufen zu müssen (wodurch man ein echtes Zertifikat

erhalten könnte) wurde das API Gateway in die Architektur mitaufgenommen. Alle API Gateway Endpunkte sind automatisch mit HTTPS verschlüsselt (Amazon API Gateway – Häufig gestellte Fragen, o. D.). Der ALB wurde trotzdem beibehalten, da er im Gegensatz zu den Containern eine fixe IP-Adresse hat, welche beim API Gateway angegeben werden kann.

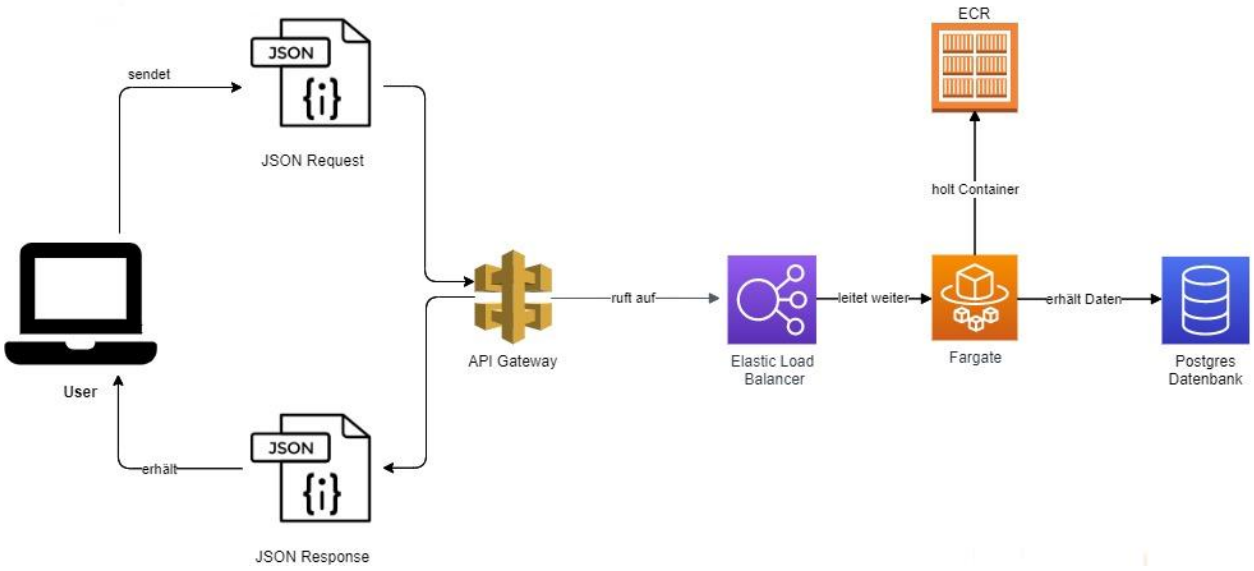

Abbildung 6: CaaS Architektur

Nachdem die grundsätzlichen Funktionalitäten implementiert wurden, musste im nächsten Schritt die API mit Fargate zur Verfügung gestellt werden. Um dies mit minimaler Konfiguration, also ohne spezielle Sicherheitsgruppen oder Subnetzen zu erreichen, mussten folgende Schritte abgearbeitet werden:

1. Docker Image erstellen
2. User und Berechtigungen im Identity and Access Management (IAM) konfigurieren
3. ECR Repository anlegen und Image hochladen
4. Fargate Cluster erstellen
5. ECS Task anlegen
6. ALB erstellen und mit ECS Task verknüpfen
7. API mit API Gateway verknüpfen

Im ersten Schritt musste zunächst ein Docker File angelegt werden, welches in Listing 3 gezeigt wird. In der ersten Zeile ist ersichtlich, dass die Node.js Version 14.18 genutzt wird. Als working directory innerhalb des Docker Containers wurde für die Applikation das Verzeichnis /usr/src/app gewählt. Im nächsten Schritt wird die Datei package.json kopiert, welche alle Abhängigkeiten der API wie zusätzliche Bibliotheken enthält. Diese werden anschließend über npm install automatisch heruntergeladen und installiert. Durch expose 8080 wird festgelegt, welcher Port der Applikation zugeordnet ist. Im letzten Schritt wird angegeben, welche Befehle nach Start des

Docker Containers ausgeführt werden sollen. Durch node server.js wird das JavaScript File server.js ausgeführt, welches die Programmlogik enthält, damit die API gestartet wird.

```
FROM node:14.18
WORKDIR /usr/src/app
COPY package*.json ./
RUN npm install
COPY ..
EXPOSE 8080
CMD [“node”, “server.js”]
```

Listing 3: Dockerfile CaaS Implementierung

Anschließend kann das Docker Image durch den Befehl in Listing 4 erstellt werden. Der -t Tag erlaubt es einen Namen zu spezifizieren, der Docker Container kann also über die Bezeichnung caas-ma angesprochen werden.

```
docker build -t caas-ma.
```

Listing 4: Dockerimage erstellen

Im nächsten Schritt musste ein IAM Benutzer und entsprechende Berechtigungen konfiguriert werden. Jeder neue Benutzer bzw. jede Applikation welche AWS Services verwendet, sollte hier als neuer Benutzer abgebildet werden. Dies hat Sicherheitsgründe, der Stammuser oder root User (jener User, welcher bei der Registrierung mit AWS erstellt wurde) sollte niemals für alltägliche Aufgaben wie programmatischen Zugriff auf das AWS Konto oder administrative Aufgaben verwendet werden. Stattdessen sollte für jede neue Applikation ein eigener User mit klar definierten Berechtigungen erstellt werden (AWS Stammbenutzer des -Kontos, o. D.). In Abbildung 7 wird der benötigte Benutzer mit der Bezeichnung caas_user angelegt. Durch Erstellen eines Zugriffsschlüssels, wird der programmatische Zugriff auf AWS Services, also direkt aus dem Programmcode einer API oder der Command Line ermöglicht. Beim Anlegen des Users muss noch festgelegt werden welche Berechtigungen dieser hat. Hier wurde Vollzugriff auf die Services ECS und ECR gewährt, unter realen Bedingungen müsste hier noch evaluiert werden, ob tatsächlich Vollzugriff benötigt wird oder auch weniger Rechte ausreichend sind.

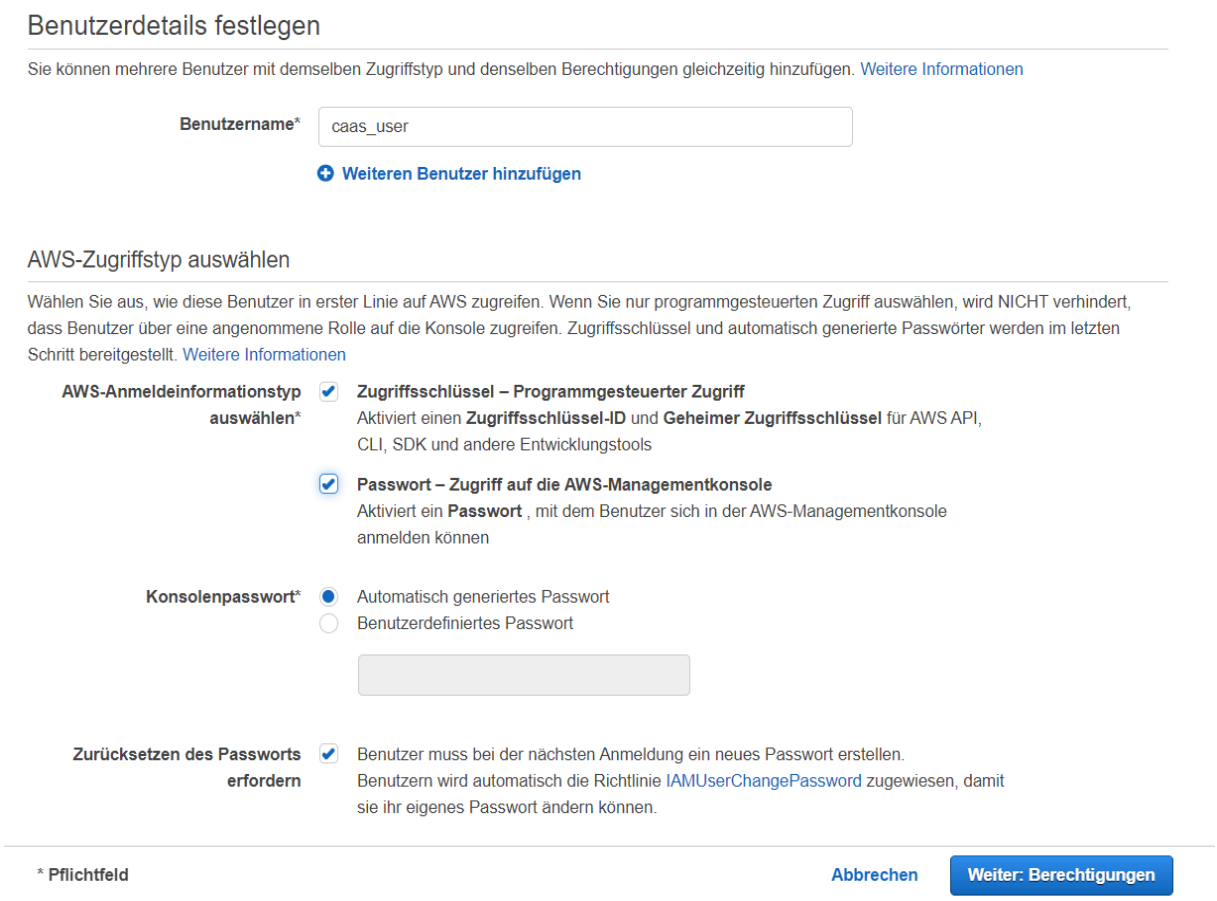

Abbildung 7: IAM Benutzer anlegen

Im dritten Schritt wurde ein ECR Repository angelegt. ECR ist ein Speicherdienst, um Docker Images an einem sicheren Ort abzuspeichern. Images können mit einem einzigen Befehl hochgeladen werden, wodurch dieser Prozess unkompliziert und schnell abgeschlossen werden kann und Applikationen schneller veröffentlicht werden können. Außerdem können alte Images archiviert werden, wodurch automatisch ein System zur Versionierung gegeben ist (Ifrah, 2019). Nach erstellen des ECR Repositories erhält man eine URL, um darauf zugreifen zu können. Anschließend erfolgt das Hochladen des zuvor erstellten Docker Images durch die Befehle in Listing 5. Durch AWS Configure meldet man sich bei AWS an. Hierfür sind der User sowie der Zugriffsschlüssel aus Schritt zwei notwendig. Zeile zwei erweitert unseren zuvor gewählten Tag caas-ma um den URL für das ECR Repository. Durch docker push wird schließlich das Image hochgeladen.

```
aws configure
docker tag caas-ma [ECR - URI]
docker push [ECR - URI]
```

Listing 5: Dockerimage in ECR hochladen

Als nächstes musste mit ECS ein Fargate Cluster erstellt werden. Hierfür wurde ein Cluster mit dem Namen caas-ma-cluster erstellt. Beim Anlegen des Clusters muss außerdem ein Template ausgewählt werden. Um eine Bereitstellung der Container durch Fargate zu erreichen, muss hier

das Template Networking only gewählt werden, dadurch wird im Hintergrund automatisch ein Fargate Provider angelegt (AWS Creating a cluster using the classic console, o. D.).

Im fünften Schritt musste schließlich noch der entsprechende ECS Task angelegt werden. Am Ende dieses Schrittes ist es möglich, die API ohne HTTPS zu nutzen. Abbildung 8 zeigt die Konfiguration eines ECS Task. Für CaaS muss Fargate als Launch Type gewählt werden. Als Name wurde caas-ma bestimmt.

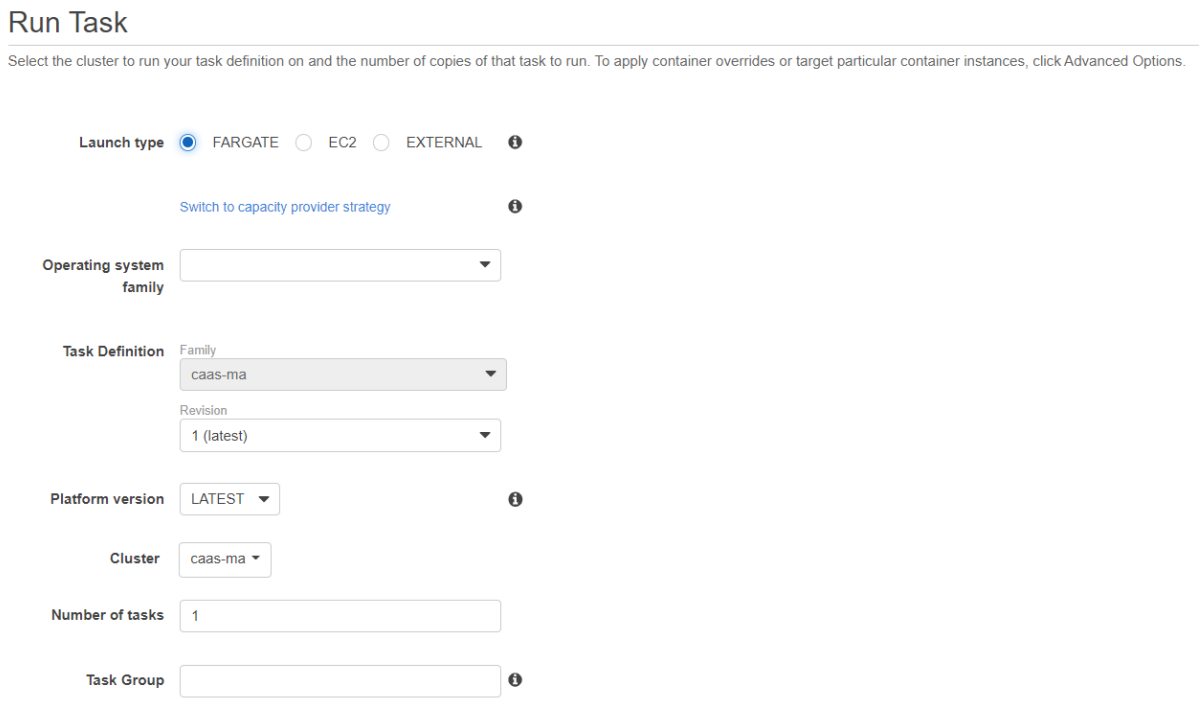

Abbildung 8: ECS Task Konfiguration

Außerdem wurden dem Task 0.5 GB Task Memory und 0.25 vCPU zugewiesen. Abschließend mussten einige Sicherheitseinstellungen getroffen werden. Hier war es wichtig dem Container eine öffentliche IP-Adresse zuzuweisen und die entsprechenden Ports, in unserem Fall 80 und 8080 freizugeben, da ansonsten die Applikation nicht von außen erreicht werden kann.

Für die Umsetzung von Schritt sechs musste zuerst ein ECS Load Balancer vom Typ Application Load Balancer erstellt werden. Die Netzwerkeinstellungen des ECS Tasks mussten anschließend angepasst werden, damit dieser den ALB verwendet. Im Zuge dessen ist es in AWS möglich, automatisch eine Load Balancer Zielgruppe zu erstellen. Danach mussten die Listener Rules am ALB konfiguriert werden. Dies ist notwendig, um den ALB mitzuteilen, wohin er Requests weiterleiten soll. So weiß der ALB, dass er die Pfade aus Abbildung 9. z.B. /api/bookings alle an die Zielgruppe caas-load-balancer-https-target weiterleiten muss, in welcher sich der ECS Task befindet.

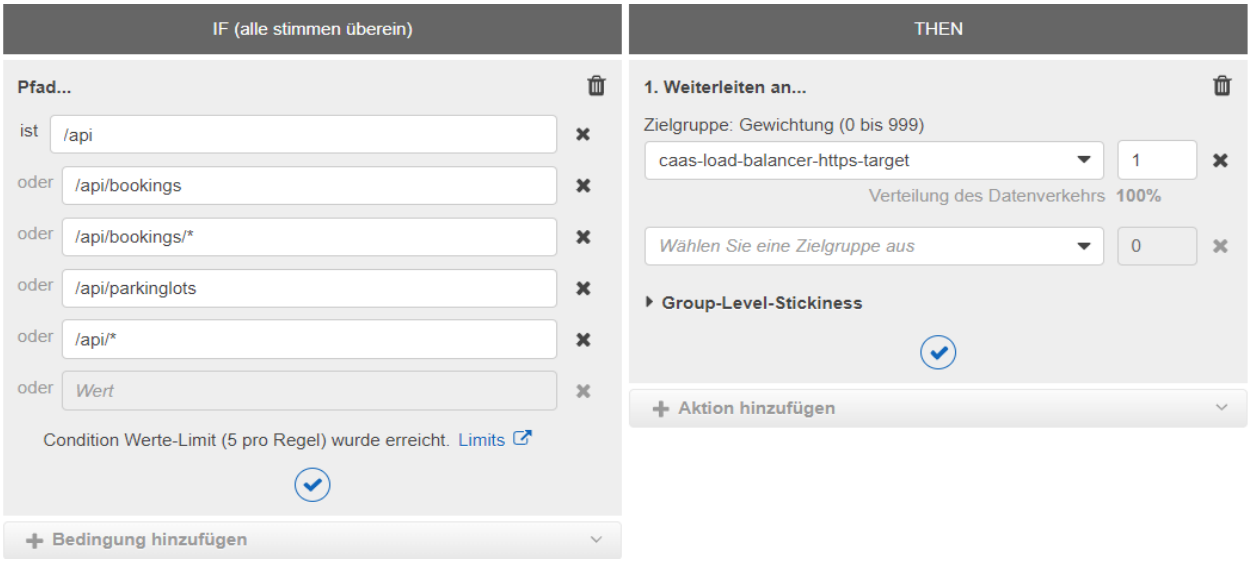

Abbildung 9: ALB Listener Rules

Im siebten und letzten Schritt wurde die API noch im API Gateway zur Verfügung gestellt. Für den ALB wurde automatisch ein DNS Eintrag mit der Adresse http://caas-load-balancer-UUID.eu-central-1.elb.amazonaws.com:8080/ erstellt. Für jede HTTP Methode aus Tabelle 6 musste noch einmal die entsprechende Methode im API Gateway erstellt werden. Dazu musste der Methodentyp und der HTTP URL für die Weiterleitung angegeben werden. Z.B. GET und http://caas-load-balancer-UUID.eu-central-1.elb.amazonaws.com:8080/api/bookings, um alle Buchungen aufzurufen. Nach dem Bereitstellen der API erhält man einen HTTPS Endpunkt, mit welchem man die API aufrufen kann. Im Hintergrund leitet das API Gateway hier auf den ALB weiter, welcher schließlich die Applikation im Container aufruft.

5.4 FaaS

AWS Lambda unterstützt die Programmiersprachen Java, Go, Python, Ruby und JavaScript in Verbindung mit Node.js (*Lambda runtimes*, o. D.). Für die praktische Umsetzung wurde JavaScript gewählt. JavaScript eignet sich perfekt für kleine, zustandslose Funktionen und wird außerdem auch von allen anderen großen FaaS Providern wie Azure und Google Cloud unterstützt. Dadurch sind die Ergebnisse auch eher auf andere Anbieter übertragbar als mit einer Programmiersprache, welche nicht von allen Anbietern unterstützt wird (Manner et al., 2018). In Tabelle 9 werden die verwendeten Technologien zusammenfassend aufgelistet. Als JavaScript Runtime wurde Node.js gewählt. JavaScript Code kann direkt im Browser ausgeführt werden und wurde früher hauptsächlich zur dynamischen Manipulation von Websites als Reaktion auf Client Interaktionen genutzt. Mittlerweile wird JavaScript jedoch auf für die Entwicklung von REST APIs eingesetzt. Um JavaScript Code Serverseitig auszuführen, wird eine Runtime Umgebung benötigt. Die meist genutzte ist hierbei Node.js. Node.js nutzt im Hintergrund die V8 Engine von Google Chrome, das bedeutet Node.js. verwendet also dieselbe Engine wie der Browser Google Chrome, um JavaScript ausführen zu können (Scherer, 2020). Als Webserver Framework wurde Express.js eingesetzt. Dies ist ein Framework für Node.js, um APIs zu entwickeln, ohne sich um die Implementierung von ständig gleichbleibenden Grundfunktionalitäten zu kümmern, wodurch die Entwicklung der API beschleunigt wird (Biswas, 2021). Zum Entstehungszeitpunkt der Arbeit war die Version 14.18 die höchste durch AWS Lambda unterstütze Node.js Version, wodurch erklärt wird, warum diese und keine neuere Version gewählt wurde (Lambda runtimes, o. D.).

Parameter	**Wert**
Programmiersprache	JavaScript
Language runtime	Node.js 14.18
Webserver Framework	Express.js
Datenbank	PostgreSQL 13.3
FaaS Plattform	AWS Lambda
Request Routing	API Gateway
Region	Europe (Frankfurt)

Tabelle 9: FaaS Technologien

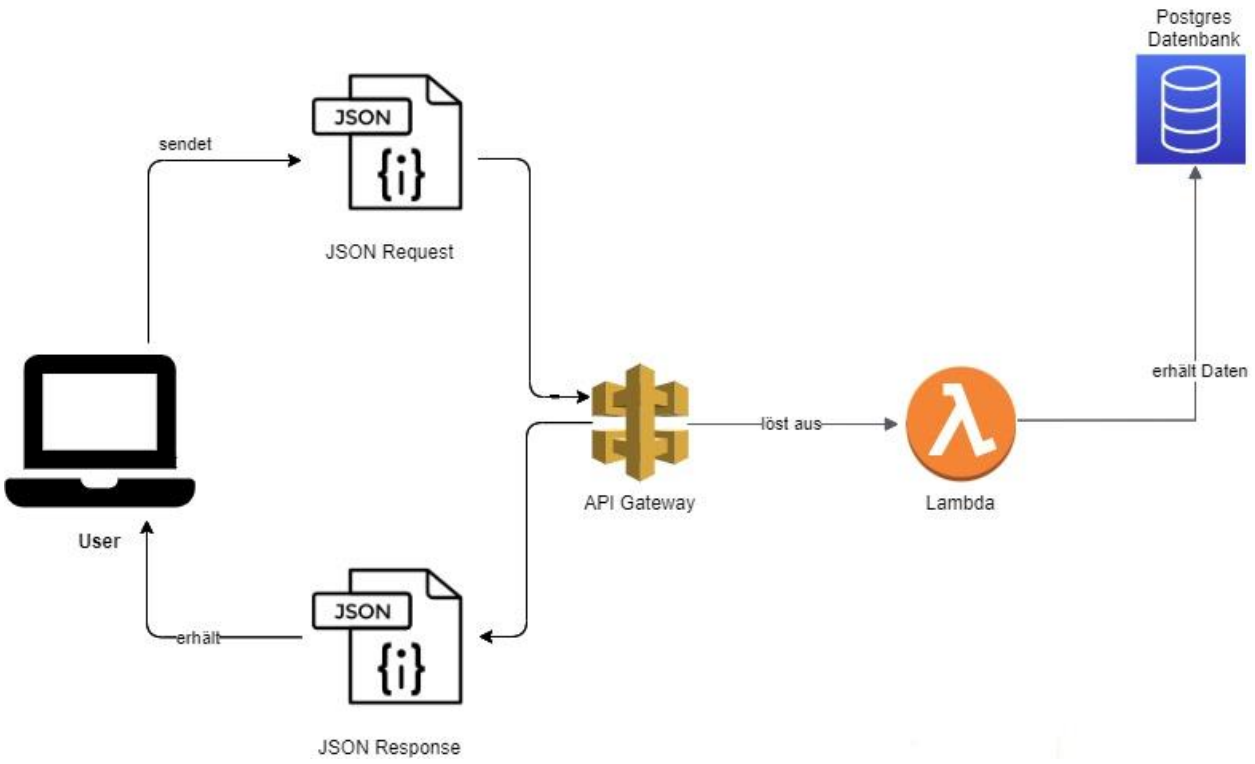

Abbildung 10: FaaS Architektur

Für die FaaS Umsetzung in Abbildung 10 wurde nur das API Gateway, Lambda und eine Postgres Datenbank benötigt. Im Vergleich zur CaaS Umsetzung gab es hier bei der Implementierung nur zwei Schritte:

1. Lambda Funktion erstellen
2. API mit API Gateway verknüpfen

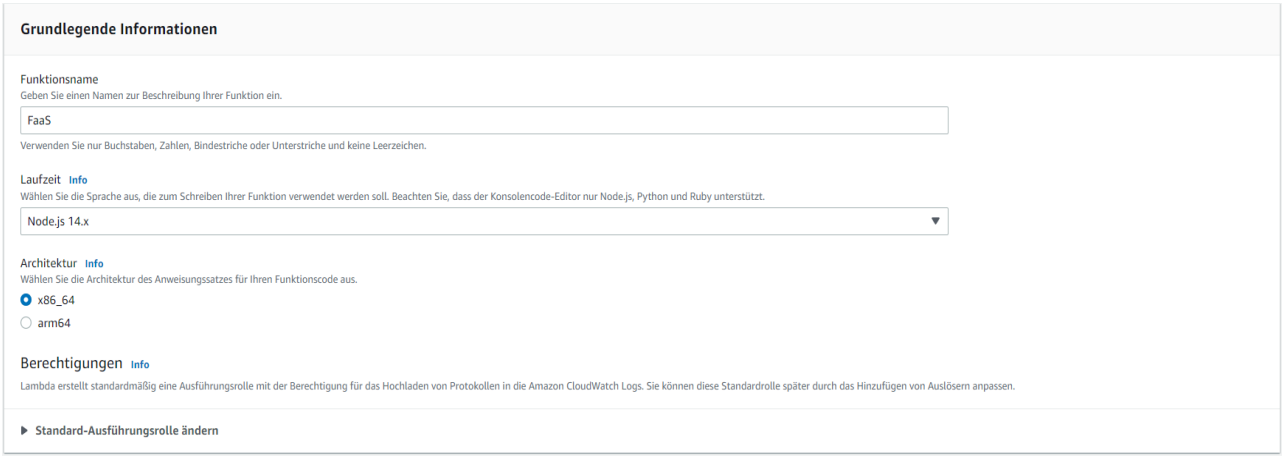

Abbildung 11: FaaS Konfiguration

Für die initiale Konfiguration muss wie in Abbildung 11 nur ein Funktionsname, eine Laufzeit (Node.js 14.18) und eine Architektur gewählt werden.

Anschließend muss der Code für die Funktion hochgeladen werden. Hier gibt es zwei verschiedene Ansätze. Entweder wählt man eine Microservice Architektur und erstellt für jeden unterschiedlichen API-Endpunkt eine eigene Lambda-Funktion, oder man wählt den monolithischen Ansatz und lädt den gesamten Applikationscode in eine einzige Funktion (Peralta, 2022). Der erste Ansatz bietet in der Praxis Vorteile, da jedoch auch die CaaS Umsetzung monolithisch ist, wurde die gesamte API mit nur einer Lambda-Funktion umgesetzt. Außerdem muss spezifiziert werden, wie die Lambda-Funktion ausgelöst werden soll, hier wurde das API Gateway gewählt.

Anders als bei der CaaS Umsetzung musste für FaaS nur eine einzige Methode im API Gateway erstellt werden. Diese leitet alle eingehenden API Aufrufe an Lambda weiter, welches diese weiterverarbeitet. Auch wenn es sich hier jeweils nur um eine minimale Konfiguration handelt und in der Praxis noch viele weitere Einstellungen konfiguriert werden können bzw. müssen sieht man beim Vergleich dieses Kapitels mit Kapitel 5.3 bereits den geringeren Konfigurationsaufwand bei der FaaS Umsetzung.

5.5 Security

Sowohl CaaS als auch FaaS fungieren in Bezug auf Security als eine Art Blackbox, ein Vergleich der beiden Implementierungen ist daher nicht möglich. AWS ist für den Schutz der Infrastruktur verantwortlich und kümmert sich auch um allfällige Aufgaben wie das Einspielen von Sicherheitsupdates (Security in AWS Lambda, o. D.). CaaS und FaaS haben jeweils nur schreibgeschützt Zugriff auf das Dateisystem des unterliegenden Systems. Außerdem ist es nicht möglich aus einem Container oder einer Lambda-Funktion heraus root Zugriff zu erhalten. Somit ist ausgeschlossen, dass Hacker*innen direkten Zugriff auf das System bekommen (Gilbert & Caudill, 2019). Dennoch gibt es einige Angriffsmethoden auf CaaS und FaaS Applikationen, welche häufig genutzt werden, zwei davon werden in den folgenden Unterkapiteln beschrieben.

5.5.1 Kompromittierung des Code Repository

In der Praxis werden Applikationen oft durch Build Pipelines automatisch bereitgestellt. Entwickler*innen laden ihren Code in ein zentrales Repository und durch Tools wie etwa einen Jenkins Server wird dieser Code sofort gebaut und die Applikation auf einer entsprechenden Umgebung bereitgestellt. Haben Angreifer*innen Zugriff auf das Code Repository, können sie schadhaften Code einschleusen, welcher durch die Build Pipeline sofort automatisch deployt wird, im schlimmsten Fall auf die Produktivumgebung. Selbst wenn der Vorfall innerhalb von Minuten erkannt wird, kann es bereits zu spät sein und es können etwa Nutzerdaten gestohlen worden sein oder Services kompromittiert worden sein (Combe et al., 2016).

5.5.2 Distributed Denial of Service Attacks

Ein DDoS-Angriff ist ein massiver, verteilter, absichtlicher und koordinierter Angriff durch mehrere Rechner, um einen Online-Dienst oder einen Server zu überwältigen. Die Angreifer versuchen, die Verfügbarkeit des Dienstes anzugreifen, indem sie umfangreiche Dummy-Daten senden, um die Ressourcen des Zielrechners zu verringern. Endziel ist es, dass der Server nicht mehr auf Anfragen reagieren kann und somit der Service komplett ausfällt (Gupta & Dahiya, 2021).

Hierauf kann bei CaaS und FaaS mit automatischer Skalierung reagiert werden. Dies bremst DDoS Attacken aus, da hier unlimitiert Ressourcen zur Verfügung stehen (theoretisch durch die Anzahl der Ressourcen der AWS Rechenzentren begrenzt, in der Praxis jedoch unrealistisch einen Angriff durchzuführen, welcher mehr Ressourcen verbraucht als AWS zur Verfügung stellen kann). Der große Nachteil hierbei ist jedoch, dass durch DDoS Attacken, wenn eine automatische Skalierung aktiviert ist, hohe Kosten entstehen, da der gesamte Ressourcenverbrauch bezahlt werden muss. AWS bietet hierfür eine Lösung namens GuardDuty an. GuardDuty ist ein Service zur Erkennung von Bedrohungen, der Tasks/Container kontinuierlich auf bösartige Aktivitäten und unbefugtes Verhalten überwacht. Der Service nutzt maschinelles Lernen, um Bedrohungen zu identifizieren und zu priorisieren. So können Anfragen von Geräten mit bestimmten IP-Adressen oder immer gleichbleibende Anfragen automatisch blockiert werden.GuardDuty kann mit dem Logging Service CloudWatch kombiniert werden, um

Events an Administrierende zu verschicken und sie auf die Bedrohung aufmerksam zu machen (Cook, 2018)

5.6 Tools

Um die API Implementierung testen zu können, sowie die Performancemessungen durchführen zu können, mussten entsprechende Tools gefunden werden. Dieser werden in den folgenden Unterkapiteln vorgestellt.

5.6.1 Postman

Im Rahmen der Implementierung wurde für beide Applikationen darauf verzichtet eine grafische Oberfläche zum Aufruf der API umzusetzen, da dadurch kein Mehrwert für den praktischen Teil der Masterarbeit erzielt worden wäre. Stattdessen musste ein Weg gefunden werden, die API nur über HTTP Requests anzusprechen. Hierbei fiel die Wahl auf Postman. Postman ist eine Applikation, um APIs zu testen und zu dokumentieren. Die Kernfunktionalität bildet ein HTTP Client mit grafischer Oberfläche, durch welchen HTTP Endpunkte mit den in Kapitel 2.2.7 erwähnten HTTP Methoden aufgerufen werden können. Dieser ermöglicht es Requests an verschiedene Arten von APIs, wie REST oder SOAP APIs, mit verschiedenen Datenformaten zu senden. Postman bietet hier die Möglichkeit Requests, welche einmal getätigt wurden abzuspeichern, um diese erneut auszuführen (Westerveld, 2021). Hier ging es nur darum, die Endpunkt Aufrufe der Applikationen zu testen und zu verifizieren, dass diese die Requests korrekt empfangen und verarbeiten können. Für eine reale, kommerziell genutzte API würde es sich anbieten, die API komplett mit Postman zu dokumentieren, hierauf wurde im praktischen Teil dieser Arbeit jedoch verzichtet.

Abbildung 12 zeigt, wie ein solcher Request mit Postman aussehen kann. Zuerst muss eine HTTP Methode ausgewählt werden. In diesem Fall handelt es sich um einen POST Request, es wird ein Request Body mitgesendet und die darin enthaltenen Daten sollen weiterverarbeitet werden. Der Request ruft den Endpunkt /api/bookings auf, es soll also eine neue Buchung in der Datenbank erstellt werden. Zur Kommunikation mit der API wird das in Kapitel 2.3 erklärte JSON Format genutzt. Die JSON Daten müssen dabei im Request Body mitgesendet werden. Dieser enthält die Uhrzeit, die Parkplatz ID und den User für die Buchung. Die rechte untere Hälfte der Abbildung zeigt schließlich, wie die Antwort des Servers auf unsere Anfrage aussieht. Wir haben eine Response mit HTTP Statuscode 200 erhalten, der Request konnte also korrekt verarbeitet werden und eine neue Buchung wurde in der Datenbank angelegt.

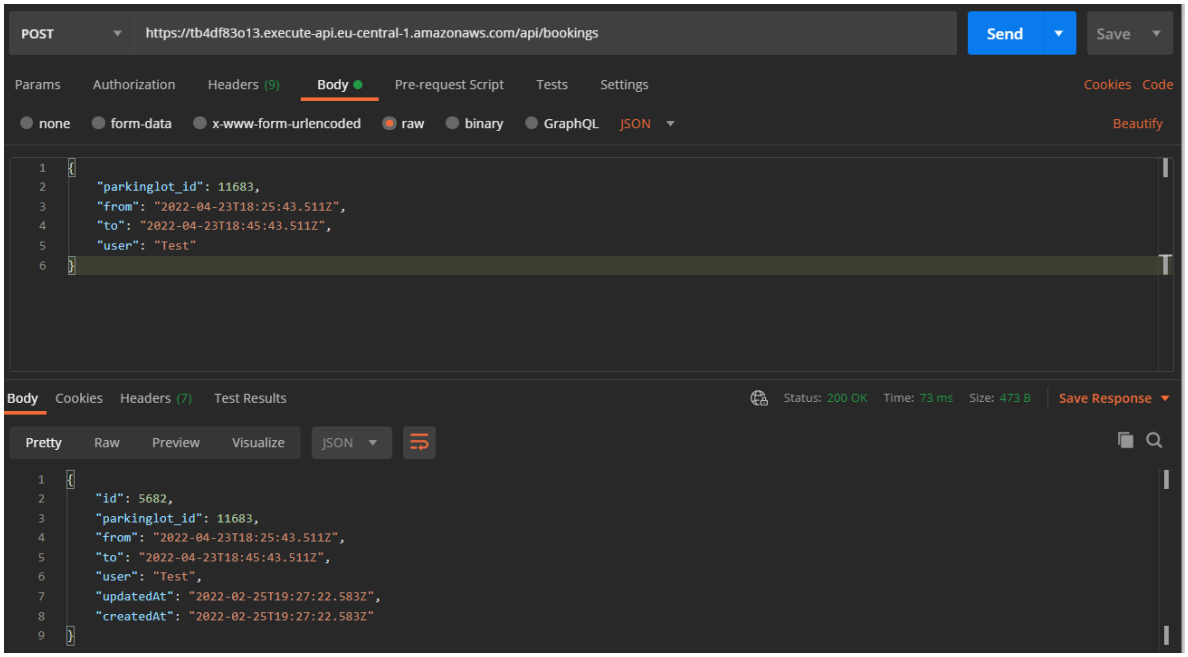

Abbildung 12: Postman Request

5.6.2 Artillery

Abschließend musste noch ein Tool gefunden werden, um die Performance der beiden APIs zu messen. Hierbei fiel die Wahl auf Artillery. Artillery ist ein Toolkit für Lasttests, das eine Reihe von Möglichkeiten zum Testen von Diensten bietet. Es unterstützt HTTP bzw. HTTPS und es gibt vorkonfigurierte Schnelltests. Bei diesen können APIs über ihre URL mit spezifischen Parametern getestet werden. Zum Beispiel ist es möglich zu konfigurieren, wie viele Anfragen gesendet werden und wie viele virtuelle Benutzer diese Anfragen senden. Komplexere skriptgesteuerte Tests, um spezifischere Testfälle abzudecken sind ebenfalls möglich. Die Definition für solche Tests muss dabei in ein YAML File geschrieben werden. Diese Art von Test ist ersichtlich in Listing 6. Als target muss der URL der API angegeben werden. Bei phases werden einerseits die Dauer des Tests, hier 300 Sekunden und andererseits die Anzahl der Aufrufe pro Sekunde, hier 50, definiert.

Der Punkt scenarios gibt an welche Aktionen die virtuellen User auf der API ausführen. In diesem Fall würde jeder virtuelle User einen Post Request gegen den Endpunkt /parkinglots senden, um einen neuen Parkplatz anzulegen. Hier kann noch definiert werden welche Daten im JSON Format, der gesendete Request enthält, zum Anlegen eines Parkplatzes werden beispielsweise die Felder „street“ und „city“ benötigt (Artillery Docs, o. D.).

```
config:
    target:”https://lambda-endpoint”
    phases:
          duration: 300
          arrivalRate: 50
    scenarios:
          flow:
               post:
                    url:”/parkinglots”
                    json:
                         ”street”: ”Körblergasse 126”,
                         ”city”: ”Graz”
```

Listing 6: Artillery Skript

6 EVALUIERUNG

Nach der Implementierung der CaaS und FaaS Umgebung folgt in diesem Kapitel der Vergleich zwischen den beiden Modellen, wodurch in weiterer Folge die Forschungsfrage beantwortet wird.

6.1 Hypothesen und Ziel

Ziel des praktischen Teils der Arbeit ist es die Forschungsfrage:

- Wie wirkt sich FaaS gegenüber CaaS auf Performance und Kostenentwicklung einer REST API aus?

zu beantworten. Dabei werden folgende Hypothesen aufgestellt, um die Forschungsfrage beantworten zu können:

- H1: Die Performance der FaaS Implementierung ist besser als die der CaaS Implementierung
- H2: FaaS verursacht gegenüber CaaS geringere Kosten

Um die Performance messen zu können, musste hier eine geeignete Kennzahl gefunden werden. Da die APIs in AWS laufen sind klassische Kennzahlen wie Verfügbarkeit (darum kümmert sich AWS) oder Ressourcennutzung (da man Ressourcen einfach skalieren kann) nicht relevant. Daher wurde die Performance mit der Client Response Time(RT) gemessen. Diese gibt die Zeit in Millisekunden an, vom Zeitpunkt, an dem der Client einen Request an den Server sendet, bis zu dem Zeitpunkt, an dem er eine Antwort erhält. Schnell reagierende Services spielen eine entscheidende Rolle für die Zufriedenheit der Benutzer*innen. Kund*innen welche regelmäßig unter langen Verzögerungen bei Webservices leiden, werden schnell frustriert sein und sich nach möglichen Alternativen umsehen. Die RT kann also über Erfolg und Scheitern von Webapplikationen entscheiden (Wei & Xu, 2011). Wie in Kapitel 5.1 beschrieben, werden durch die APIs Funktionalitäten für ein fiktives Unternehmen mit Buchungsplattform zur Verfügung gestellt. Daher ist die RT hier die ideale Kennzahl.

Der in Kapitel 4.1 erwähnte cold start wurde hierbei nicht untersucht. Durch das Senden von vielen Requests, können nachfolgende bereits initialisierte Lambdafunktionen nutzen (Manner et al., 2018). Die Einzelfälle, in denen ein cold start auftritt, beeinflussen die Messungen daher nicht signifikant, da der Median der RT Werte verwendet wird und cold starts somit als Ausreißer gelten.

6.2 Experimente

Um die RT zu messen, wurden zwei verschiedene Szenarien erstellt. In Szenario 1 wurde die RT jedes einzelnen Endpunkts gemessen. In Szenario 2 wurde ebenfalls die RT gemessen, dabei wurden aber durch jeden virtuellen User mehrere Endpunkte aufgerufen. Szenario 2 ähnelt also mehr einer customer journey und ist daher näher an der Realität. Diese Ergebnisse für die Performance wurden anschließend in Kapitel 6.2.3 genutzt, um Kosten für verschiedene Nutzungszahlen vorherzusagen. Jedes Szenario wurde mit 1.000 virtuellen Usern durchgeführt. Dies bedeutet, dass jeder Test 1.000 Mal durchgeführt wurde und gewährleistet, dass die Ergebnisse aussagekräftig sind, da sich zwischen einzelnen Ausführungen immer Unterschiede ergeben werden. Die Anzahl der virtuellen User wurde nicht höher als 1.000 gewählt, um zu verhindern, dass zu hohe reale Kosten anfallen.

Folgende Metriken wurden für die RT gemessen:

- **Minimum**: Schnellste RT über alle 1.000 Requests
- **Maximum**: Langsamste RT über alle 1.000 Requests
- **95. Perzentil**: 95 % der RT Messungen liegen unter diesem Wert. Diese Kennzahl kann zusätzlich zum Median als Wert herangezogen, um einzuschätzen, wie lange Benutzer*innen auf eine Antwort vom Server warten müssen.
- **Median**

Für die Performancemessungen wurde die in RDS genutzte Datenbank auf das Modell db.t4g.xlarge umgestellt. Dadurch besitzt die Datenbank 4 vCPUs und 16 GB Arbeitsspeicher (Amazon RDS Instance Types, o. D.)

6.2.1 Szenario 1

Im ersten Szenario wurden alle Endpunkte separat getestet. Dadurch sollte herausgefunden werden, ob es bei verschiedenen Endpunkten bzw. HTTP Methoden größere Unterschiede gibt. Es wäre z.B. möglich, dass das Finden eines Parkplatzes über die Id, bei CaaS und FaaS ähnlich performant ist, CaaS aber beim Aktualisieren eines Parkplatzes eine signifikant bessere RT hat. Wäre nur Szenario 2 durchgeführt worden, bei welchem mehrere API Endpunkte nacheinander aufgerufen werden, wäre es nicht möglich gewesen Performanceunterschiede zwischen einzelnen Endpunkten zu erhalten, da in Szenario 2 nur eine aggregierte RT für alle Schritte erhalten wurde.

Die Endpunkte wurden in folgender Reihenfolge getestet:

- POST /parkinglots
- POST /bookings
- PUT /parkinglots/id
- PUT /bookings/id

- GET /parkinglots
- GET /bookings
- GET /parkinglots/id
- GET /bookings/id

Es wurden also erst jeweils 1.000 Parkplätze und Buchungen erstellt, bei welchen anschließend der Datensatz aktualisiert wurde. Danach wurden alle Parkplätze und Buchungen sowie jeweils ein bestimmter Parkplatz und eine Buchung per Id aufgerufen. Leider war es nicht möglich, die Endpunkte zum Löschen von Datensätzen zu testen. Da nur eine fixe Id für den zu löschenden Datensatz im Skript hinterlegt werden kann und Daten natürlich nicht zwei Mal gelöscht werden können. Daher wäre es notwendig gewesen nach dem Löschen, den Datensatz wieder neu mit der alten Id einzufügen. Dadurch wäre es aber nicht möglich gewesen nur die Zeit für den Löschvorgang herauszufinden. Man kann beim Löschen von Daten jedoch davon ausgehen, dass die Performance sich ähnlich wie beim Aktualisieren verhält. Beide Methoden wurden in den APIs ähnlich implementiert und haben jeweils zwei Datenbankaufrufe. Damit die CaaS und FaaS APIs mit denselben Datenmengen arbeiten, wurde die Datenbank nachdem die CaaS Endpunkte getestet wurden, vollständig bereinigt bevor der Test der FaaS Endpunkte gestartet wurde.

Tabelle 10 enthält die Ergebnisse der Peformancemessungen der einzelnen Endpunkte. Wie zu erwarten, verhielten sich Endpunkte mit ähnlicher Logik und derselben Implementierung beinahe deckungsgleich. So ergab sich beim Einfügen eines Datensatzes in die Datenbank bei CaaS für ein POST auf /parkinglots eine RT von 40ms und auf /bookings 41ms. Eine Schwankungsbreite einigen ms ist dabei grundsätzlich normal, da sich Netzwerkbedingungen ständig ändern. Die einzige Ausnahme bildet der Aufruf aller Datensätze über GET /parkinglots bzw. /bookings. Sowohl bei CaaS als auch FaaS war die RT beim Aufruf der Buchungen signifikant höher. Dies liegt daran, dass Buchungen einen Fremdschlüssel besitzen, welcher mit den Parkplätzen verknüpft ist. Beim Aufruf der Buchungen müssen also auch Informationen über die Parkplätze geladen werden, wodurch sich der große Unterschied bei der RT erklärt.

Implementierung	**Endpunkt**	**Minimum**	**Maximum**	**Median**	**95. Perzentil**
CaaS	POST /parkinglots	40ms	265 ms	54 ms	74 ms
FaaS		53ms	352 ms	83 ms	149 ms
CaaS	POST /bookings	41ms	269 ms	56 ms	80 ms
FaaS		56ms	545 ms	87 ms	153 ms
CaaS	PUT /parkinglots/id	43ms	265 ms	55 ms	104 ms
FaaS		52ms	986 ms	85 ms	144 ms
CaaS	PUT /bookings/id	40ms	280 ms	55 ms	77 ms
FaaS		56ms	380 ms	87 ms	153 ms

Implementierung	Endpunkt	Minimum	Maximum	Median	95. Perzentil
CaaS	GET /parkinglots	92 ms	1702 ms	308 ms	871 ms
FaaS		239 ms	1495 ms	347 ms	658 ms
CaaS	GET /bookings	139 ms	5500 ms	596 ms	3190 ms
FaaS		345 ms	2100 ms	487 ms	854 ms
CaaS	GET /parkinglots/id	40 ms	258 ms	51 ms	71 ms
FaaS		48 ms	327 ms	71 ms	122 ms
CaaS	GET /bookings/id	39 ms	257 ms	53 ms	74 ms
FaaS		53 ms	119 ms	73 ms	115 ms

Tabelle 10: RT per Endpunkt

Um schließlich einzelne Funktionalitäten wie Einfügen, Aktualisieren und Datenaufrufe vergleichen zu können, wurden Medianwerte für Endpunkte mit gleicher Funktionalität, z.B. Einfügen eines Datensatzes in die Tabelle Parkplätze oder Buchungen, über POST /parkinglots bzw /bookings, zusammengezählt. Da es zwei Tabellen in der Datenbank gibt, gibt es also jeweils zwei Endpunkte mit gleicher Funktionalität. Mit dem zusammenaddierten Medianwert wurde dann der Durchschnitt gebildet. Mit diesen Werten wurde in Tabelle 11 der RT Unterschied in Millisekunden und in Prozent berechnet. CaaS schlägt FaaS hier bei allen vier Aufgaben. Beim Einfügen und Aktualisieren eines Datensatzes ergibt sich ein extremer Performanceunterschied von 54 % bzw. 56 %

Aufgabe	CaaS	FaaS	RT Unterschied in ms	Performanceunterschied RT in %
Datensatz einfügen	55 ms	85 ms	30 ms	54,54 %
Datensatz aktualisieren	55 ms	86 ms	31 ms	56,36 %
Aufrufen aller Datensätze	452 ms	417 ms	35 ms	8,39 %
Aufrufen eines Datensatzes	52 ms	72 ms	20 ms	38,46 %

Tabelle 11: Szenario 1 CaaS vs. FaaS RT

Hier muss jedoch beachtet werden, dass Szenario 1 eher theoretischer Natur ist, da Benutzer*innen in den seltensten Fällen nur einen einzigen API Aufruf tätigen/auslösen. Die Performancevorteile für CaaS wirken auf den ersten Blick gewaltig, könnten sich unter realistischeren Bedingungen jedoch noch als weniger gravierend herausstellen.

6.2.2 Szenario 2

In Szenario 2 wurden nicht mehr einzelne isolierte API Aufrufe getestet, sondern es wurden durch jeden virtuellen User mehrere API Aufrufe nacheinander ausgeführt. Ziel war es, durch Verknüpfen mehrerer API Aufrufe, reale Anwendungsfälle abzubilden. Listing 7 zeigt einen Auszug des Artillery Skripts, welches hierfür erstellt wurde. Hier wurden insgesamt, wie in Szenario 1, 1.000 API Aufrufe durchgeführt. Dafür wurden vier verschiedene Anwendungsfälle erstellt und mit einer Gewichtung versehen. Durch die Gewichtung kann festgelegt, wie wahrscheinlich es ist, dass ein virtueller User diesen Fall auswählt. Eine Gewichtung von 600 für den ersten Fall sagt etwa aus, dass bei 1.000 virtuellen Usern, 600 davon die Schritte für diesen Fall ausführen. Dieses Verhalten ist realistischer, da die meisten Benutzer*innen die Plattform nur besuchen werden, ohne einen Parkplatz zu buchen oder anzulegen. Folgende vier Anwendungsfälle wurden ausgewählt:

- Besucher*in: Eine Person, welche die Website nur besucht, ohne etwas zu buchen. Sieht sich mehrmals alle Parkplätze sowie Details einzelner Parkplätze an. Insgesamt 6 API Aufrufe.
- Mieter*in: Sieht sich ebenfalls alle Parkplätze sowie mehrere Detailseiten an. Tätigt anschließend eine Buchung und ruf die Detail Seite der Buchung auf. Ändert abschließend noch die bereits getätigte Buchung. Insgesamt sieben API Aufrufe.
- Vermieter*in: Legt einen Parkplatz an und ruft die Detailseite auf. Da erst Parkplatzinformationen vergessen wurden, werden die Daten anschließend aktualisiert. Ruft danach eine Buchung für einen zuvor erstellten Parkplatz auf. Insgesamt vier API Aufrufe.
- Administrator*in: Ruft alle Parkplätze und Buchungen auf, die im System hinterlegt sind. Insgesamt zwei API Aufrufe.

Jeder Anwendungsfall löst unterschiedlich viele API Aufrufe aus. Insgesamt ergeben sich durch alle virtuellen User 6080 API Aufrufe.

```
config:
    target:”https://lambda-endpoint”
    phases:
         duration: 100
         arrivalRate: 10
         maxVUsers: 1000
    scenarios:
       name: "A user user collecting information"
       weight: 600
          url: "GET /parkinglots" x3
          url: "GET /parkinglots/id" x3
       name: "A user who books a parking lot"
       weight: 300
          url: "GET /parkinglots" x2
          url: "GET /parkinglots/id" x2
          url: "POST /bookings " x1
          url: "GET /bookings/id" x1
          url: "PUT /bookings/id" x1
       name: "A user who offers his parking lot"
       weight: 90
          url: "GET /parkinglots" x2
          url: "GET /parkinglots/id" x2
          url: "POST /bookings " x1
          url: "GET /bookings/id" x1
       name: "An administrator looking data entries”
       weight: 10
          url: "GET /parkinglots" x1
          url: "GET /bookings" x1
```

Listing 7: Artillery Skript Szenario 2

Für Fälle, in denen durch virtuelle User mehrere Endpunkte getestet werden, ist es mit Artillery nicht möglich Werte für Minimum, Maximum und das 95. Perzentil zu erhalten, daher enthält Tabelle 12 nur den Median für die RT der einzelnen Anwendungsfälle. Zwar war CaaS hier immer noch die performantere Lösung für jeden der vier Anwendungsfälle, allerdings nicht mehr mit einem Performanceunterschied von teilweise über 50 %, wie in Kapitel 6.2.1 in Tabelle 11 gezeigt. Ein Performanceunterschied von 16 % wie beim Anwendungsfall Vermieter*in, kann für viele Applikationen, die Daten schnell in Echtzeit bearbeiten müssen jedoch immer noch einen enormen Unterschied ausmachen.

Implementierung	Anwendungsfall	Median	Performance-unterschied in ms	Performance-unterschied RT in %
CaaS	Besucher*in	1137 ms	36ms	3,16 %
FaaS		1173 ms		
CaaS	Mieter*in	1132 ms	83ms	7,33 %
FaaS		1215 ms		
CaaS	Vermieter*in	460 ms	75ms	16,30 %
FaaS		535 ms		
CaaS	Administrator*in	702 ms	43ms	6,13 %
FaaS		745 ms		

Tabelle 12: Szenario 2 CaaS vs. FaaS RT

6.2.3 Kostenberechnung

Bei den Performancemessungen war CaaS zumindest mit den in dieser Arbeit verwendeten Parametern performanter als FaaS. Doch wie sieht es mit den Kosten aus? Es könnte immer noch sinnvoll sein, einen Kompromiss bei der Performance einzugehen, wenn dadurch Kosten gespart werden können. Für die Lambdafunktionen wurde die Kalkulation der Kosten auf Basis einer ARM-Architektur und 512 MB Arbeitsspeicher durchgeführt, da dies für die meisten Aufgaben ausreichend ist. Nicht berücksichtigt wurden Kosten für die Datenbank und das API-Gateway da diese Services sowohl von Caas als auch FaaS verwendet werden. Da die Architektur von CaaS sich grundlegend von jener von FaaS unterscheidet, war es hier nötig, die Anzahl an Ressourcen für unterschiedliche Nutzerzahlen zu erhöhen. Bei Lambda stehen jeder Nutzer*in 512 MB an Ressourcen für die Funktion zur Verfügung. Bei CaaS würde man jedoch einer Applikation mit 1.000 täglichen Benutzer*innen nicht dieselben Ressourcen zuweisen wie einer Applikation mit 100.000 täglichen Benutzer*innen, da die Ressourcen über alle Benutzer*innen geteilt werden. Die hierfür gewählten Parameter sind in Tabelle 13 ersichtlich.

Tägliche Benutzer*innen	VCpus	RAM
1.000	0,25	2
10.000	0,5	4
100.000	2	8
500.000	4	16

Tägliche Benutzer*innen	VCpus	RAM
700.000	4	16
1.000.000	4	22
2.000.000	4	30

Tabelle 13: CaaS Ressourcen

Auf Basis dieser Werte für die CaaS Ressourcen, wurde der Kostenvergleich in Tabelle 14 durchgeführt. Da die Kosten für FaaS abhängig von der Ausführungsdauer der Lambdafunktion ist, wurden hierfür die RT Werte aus Tabelle 12 verwendet. Die Kosten wurden mithilfe der Preise für CaaS und FaaS aus Tabelle 2 und Tabelle 3 berechnet. Zusätzlich wurden bei FaaS noch die In Kapitel 4.2 erwähnten Zusatzkosten von 18 Cent pro einer Million API Aufrufe hinzugerechnet. Ausgegangen wurde hier ebenfalls von der in Szenario 2 ausgewählten Verteilung der Anwendungsfälle wodurch sich 6080 Requests pro 1.000 Benutzer*innen ergeben. Im Durchschnitt also etwa sechs Requests pro Benutzer*in. In der Spalte Kostensteigerung pro Jahr wird dabei Steigerung der Kosten von der günstigeren auf die teurere Implementierung in absoluten Zahlen und Prozent angegeben. Hier ist ersichtlich, dass gerade bei einer geringen Anzahl an API Aufrufen FaaS deutlich günstiger ist. Bei 6.080 API Aufrufen fallen bei FaaS nur 3,29 € pro Tag an, bei CaaS jedoch 149,65 €, was einer Kostensteigerung im Jahr von 4448,63 % entspricht. Bei den betrachten Nutzungszahlen fallen erst bei 700.000 Benutzer*innen höhere Kosten für FaaS an. Für sehr viele Benutzer*innen bzw. API Aufrufe ist CaaS also besser geeignet.

Implementierung	Tägliche Benutzer* innen	API Aufrufe pro Tag	Preis pro Tag	Preis pro Jahr	Kostensteigerung pro Jahr
CaaS	1.000	6.080	€ 0,41	€ 149,65	€ 146,36
FaaS			€ 0,0090	€ 3,29	4448.63 %
CaaS	10.000	60.800	€ 0,84	€ 306,06	€ 274,66
FaaS			€ 0,086	€ 31,40	874,41 %
CaaS	100.000	608.000	€ 2,57	€ 939,24	€ 625,34
FaaS			€ 0,86	€ 313,90	199,21 %
CaaS	500.000	3.040.000	€ 5,14	€ 1.878,49	€ 305,34
FaaS			€ 4,31	€ 1.573,15	19,40 %
CaaS	700.000	4.256.00	€ 5,14	€ 1.878,49	€ 300,56
FaaS			€ 5,97	€ 2.179,05	16,00 %

Implementierung	Tägliche Benutzer*innen	API Aufrufe pro Tag	Preis pro Tag	Preis pro Jahr	Kostensteigerung pro Jahr
CaaS	1.000.000	6.080.000	€ 5,73	€ 2.093,65	€ 1.112,19
FaaS			€ 8,81	€ 3.215,65	53,60 %
CaaS	2.000.000	12.160.000	€ 6,52	€ 2.380,09	€ 3.905,21
FaaS			€ 17,22	€ 6.285,30	164,07 %

Tabelle 14: CaaS vs. FaaS Kostenvergleich

Abbildung 13 zeigt, die visualisierten Ergebnisse aus Tabelle 14 für die Kosten pro Jahr. Erst bei 1.554.440.000 jährlichen API Requests übersteigen die Kosten von FaaS jene von CaaS. Ersichtlich ist auch das Abflachen des Kostenanstiegs bei CaaS ab 1.109.600.000 API Aufrufen. Die jährlichen Kosten betragen hier für CaaS 1.878,49 €. Erhöhen sich die API Aufrufe von 1.109.600.000 auf 22.192.000.000 Aufrufe entsteht hier eine Kostensteigerung von 26 % auf 2.380,09 €. Bei FaaS steigern sich die Kosten für dieselbe Erhöhung der API Aufrufe von 1.573,15 € auf 6.285,30 € pro Jahr, dies entspricht einer Steigerung von 300 %. Während es sich in absoluten Zahlen bei CaaS um eine Erhöhung von 501,60 € handelt steigen die Kosten bei FaaS um 4.712,15 €. Die Kostensteigerung bei FaaS beträgt hier also etwas mehr als das Neunfache der Steigerung bei CaaS.

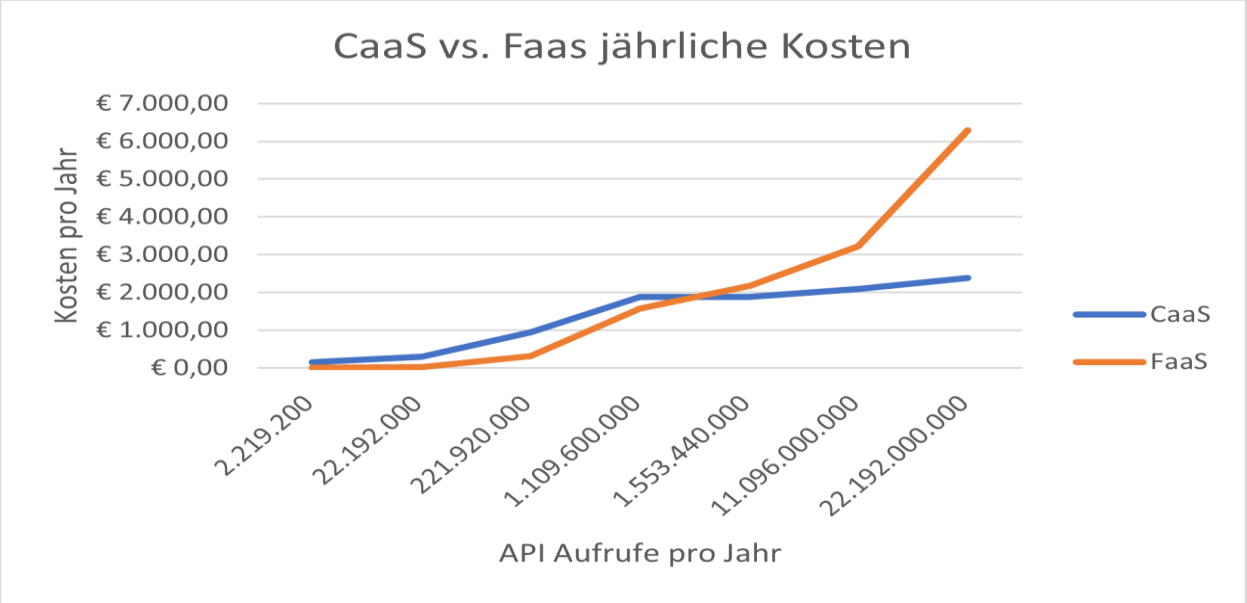

Abbildung 13: CaaS vs. FaaS jährliche Kosten

7 DISKUSSION

Mithilfe der Ergebnisse aus Kapitel 6 werden in diesem Kapitel die Hypothesen beurteilt sowie Risiken für die Validität der Arbeit erörtert.

7.1 Performance & Kosten

Die in Kapitel 6.1 aufgestellten Hypothesen konnten durch die Erkenntnisse der Experimente in Kapitel 6.2 beantwortet werden.

H1: Die Performance der FaaS Implementierung ist besser als die der CaaS Implementierung

Dies konnte mit den Parametern dieser Arbeit nicht gezeigt werden. FaaS hatte bei allen Performancetests eine höhere RT als CaaS. Bei Szenario 1 ergaben sich teils sogar Performanceunterschiede von über 50 %. Bei Szenario 2 bei welchem die Performance mehrerer API Aufrufe getestet wurde, waren die Unterschiede nicht mehr so extrem, es ergab sich jedoch immer noch ein Unterschied zwischen 3 % und 16 %. Bei FaaS entsteht, wie in Kapitel 4.1 beschrieben, bei der Initialisierung der Funktionen ein größerer Overhead als bei CaaS, die Ergebnisse spiegeln dies durch die schlechtere Performance von FaaS wider.

H2: FaaS verursacht gegenüber CaaS geringere Kosten

Auch diese Hypothese trifft nicht immer zu und konnte daher nicht bestätigt werden. Zwar konnte gezeigt werden, dass FaaS vor allem wenn es nur einige hunderttausend API Aufrufe gibt, um einiges günstiger ist, jedoch nimmt dieser Kostenvorteil mit einer höheren Anzahl an API Aufrufen immer mehr ab. Bei etwa vier Millionen täglichen API Aufrufen ist ein Punkt erreicht, an dem für CaaS und FaaS in etwa dieselben Kosten anfallen, bei 12 Millionen täglichen Aufrufen ist die FaaS Lösung bereits zweieinhalb Mal so teuer wie CaaS. Für APIs, welche täglich mehrere Millionen API Aufrufe haben, ist CaaS also besser geeignet. FaaS ist hier vor allem für APIs mit „wenigen“ Aufrufen geeignet bzw. für APIs, die nur sporadisch aufgerufen werden, da hier bei FaaS nur für die Zeit der Ausführung gezahlt werden würde, bei CaaS der Container aber immer laufen würde und dadurch höhere Kosten entstehen. Für APIs im Enterprise Bereich oder Webservices welche API Aufrufe im zweistelligen Millionenbereich erreichen ist FaaS jedoch im Hinblick auf anfallende Kosten nicht geeignet.

7.2 Validität der Ergebnisse

Die wissenschaftliche oder industrielle Bedeutung von empirischen Studien hängt von ihrer Validität ab, d. h. davon, inwieweit man den Ergebnissen einer empirischen Studie vertrauen kann. Shadish definiert mehrere Arten von Bedrohungen der Validität, welche beachtet werden sollten, interne Validität, externe Validität und Konstruktvalidität. Dieser Ansatz findet auch in der empirischen Forschung der Softwareentwicklung Verwendung und wird daher auch in dieser Arbeit angewandt (Felderer & Travassos, 2020). Um Ergebnisse außerdem transparent und reproduzierbar zu machen, wird sämtlicher Code für die Implementierung sowie Artillery Skripte auf https://github.com/barcly16/caas bzw. https://github.com/barcly16/faas zur Verfügung gestellt.

7.2.1 Interne Validität

Unter interne Validität fallen alle Vorgänge, welche mit dem Sammeln von Messdaten zu tun haben (Felderer & Travassos, 2020).

Eine Bedrohung der internen Validität dieser Arbeit stellen die Netzwerkbedingungen zum Zeitpunkt der Performancemessungen dar. Zwar wurden alle Messungen öfters durchgeführt, um Ausreißer zu erkennen, jedoch kann nicht ganz ausgeschlossen werden, dass eine Implementierung von etwas besseren Netzwerkbedingungen zum Zeitpunkt der Messung profitiert hat.

Weiteres wurde die Zahl von 1.000 virtuellen Benutzer*innen pro Szenario willkürlich gewählt. Diese Zahl wurde vom Autor als ausreichend groß empfunden, um aussagekräftige Ergebnisse zu erzielen und nicht höher gewählt, um nicht zu hohe Kosten zu verursachen. Im Sinne der internen Validität wären zehntausend oder gar einige hunderttausend virtuelle Benutzer*innen noch wesentlich besser, jedoch standen im Rahmen dieser Arbeit weder die monetären noch die technischen Ressourcen hierfür zur Verfügung.

7.2.2 Externe Validität

Externe Validität bezieht sich darauf, ob Ergebnisse generalisiert werden können, also auch unter anderen Bedingungen erzielt werden können (Felderer & Travassos, 2020). Durch die Szenarien wurde versucht, reale Anwendungsfälle nachzustellen. Jedoch wurde hierbei die Verteilung der virtuellen Benutzer*innen, in die einzelnen Gruppen, nur vom Autor aufgrund eigener Erfahrungen im Bereich von Buchungsplattformen gewählt, es gibt hierzu keine genauen generalisierbaren Statistiken. Weiters ist der gewählte Cloudprovider AWS nur einer von vielen Anbietern für CaaS bzw. FaaS Plattformen. Es ist daher nicht möglich zu sagen, ob die Ergebnisse dieser Arbeit auf andere Anbieter wie Azure oder Google Cloud übertragbar sind. Außerdem werden sowohl Lambda als auch Fargate laufend weiterentwickelt, zukünftige neue Entwicklungen könnten eine Bedrohung der externen Validität dieser Arbeit darstellen.

7.2.3 Konstruktvalidität

Konstruktvalidität bezeichnet die Validität des Studiendesigns (Felderer & Travassos, 2020). Da die FaaS und CaaS Implementierungen vom Autor durchgeführt wurden, handelt es sich dabei im Vergleich zu Projekten an denen Softwareteams in der Praxis arbeiten, um sehr simple APIs. Dies könnten die Ergebnisse auf zwei Arten beeinflussen. Erstens wäre es möglich, dass in der Praxis, der Code durch entsprechende Analyse und Anwenden spezieller Software Design Patterns, welche dem Autor zum Zeitpunkt der Implementierung nicht bekannt waren, noch optimiert werden könnte. Je nachdem welche Implementierung hiervon betroffen wäre, würde sich die Performance von CaaS oder FaaS noch verbessern. Da die APIs aber keine komplexen Funktionalitäten zur Verfügung stellen, wird dieses Risiko als eher gering eingeschätzt. Zweitens anschließend an den vorigen Punkt, ist die Logik hinter API Aufrufen je nach Anwendung in der Realität häufig deutlich komplexer als in dieser Arbeit. Dadurch könnten sich für FaaS längere Laufzeiten ergeben, wodurch sich die in dieser Arbeit abgeleiteten Kosten verschieben würden. Bei der Komplexität der Umsetzung ist es jedoch schwierig einen Kompromiss zu finden. Simple Umsetzungen sind besser vergleichbar, stellen in der Realität jedoch nicht den Großteil der Anwendungsfälle dar. Zu komplexe Umsetzungen unterscheiden sich jedoch in der Art der Umsetzung vielleicht schon so stark, dass kein sinnvoller Vergleich mehr gegeben ist. Daher wurde im Hinblick auf die Vergleichbarkeit, mit dem Einfügen, Lesen, Aktualisieren und Löschen von Werten aus der Datenbank, eine simple Implementierung gewählt, welche in der Praxis dennoch relevant ist.

8 FAZIT

Ziel der Masterarbeit war es, die Forschungsfrage:

Wie wirkt sich FaaS gegenüber CaaS auf Performance und Kostenentwicklung einer REST API aus?

zu beantworten.

Die Forschungsfrage kann so beantwortet werden, dass die Performance von FaaS sicher nicht für alle Applikationen geeignet ist. Für Applikationen, bei welchen Performance im Vordergrund steht und eine um 100 ms schlechtere RT schon einen signifikanten Unterschied bedeutet, wird sich FaaS negativ auf die Erfahrung von Benutzer*innen auswirken. FaaS kann also nicht bedingungslos für alle API Anwendungsfälle empfohlen werden.

Anders sieht es bei den Kosten aus. Hier erscheint FaaS vor allem für kleine Applikationen mit geringem Datenverkehr oder für Prototypen relevant. Für diese Applikationen könnte die Entscheidung zugunsten deutlich geringerer Kosten in Kombination mit weniger Konfigurationsaufwand, auf einige Prozent an Performance zu verzichten durchaus sinnvoll sein. Bei sehr vielen API Aufrufen geht jedoch auch der Kostenvorteil verloren und es können durch eine FaaS Architektur sogar hohe Mehrkosten entstehen.

Abschließend sollen einige Möglichkeiten aufgezeigt werden, wie dieses Thema in Zukunft weiter betrachtet werden könnte, speziell in Hinblick der schlechteren Performance der FaaS Implementierung:

- Eine in dieser Arbeit nicht quantifizierte Variable ist der Aufwand für Konfiguration und Wartung der jeweiligen Implementierungen. Gerade wenn man den deutlich geringeren Aufwand der FaaS Implementierung in Kapitel 5.4 gegenüber CaaS in Kapitel 5.3 betrachtet, könnten dadurch Entwicklungsressourcen/Kosten eingespart werden.
- Die offensichtlichste Möglichkeit, um die Performance zu verbessern, wäre den Arbeitsspeicher, der den Lambdafunktionen zugewiesen ist zu erhöhen. In dieser Arbeit wurden 512 MB verwendet, das Maximum sind hier 10 GB (AWS Lambda - Preise, o. D.). Hier ist nicht klar, ob eine Erhöhung automatisch eine Verbesserung der Performance bedeuten würde, da die Funktionalitäten der APIs nicht sonderlich komplex sind und 512 MB hier ausreichen sollten. Auf jeden Fall ginge eine Erhöhung des Arbeitsspeichers mit höheren Kosten für FaaS einher.
- Zwar wurde der cold start nicht explizit in dieser Arbeit betrachtet und es waren auch keine großen Verzerrungen durch cold starts festzustellen, dennoch ist es möglich diesen abzuschwächen bzw. zu verhindern, um noch an Performance zu gewinnen.
- Eine letzte Möglichkeit, die betrachtet werden könnte, ist ein Caching System für Lambda zu nutzen. Dadurch könnten Datenaufrufe mit den HTTP GET Methoden bei mehreren identen Aufrufen mit Daten aus dem Cache beantwortet werden und es wäre kein Zugriff auf die Datenbank notwendig. So könnten Daten schneller zurückgeliefert werden, da sie bereits im Cache gespeichert wären (Wood & Prasath, 2021).

ABKÜRZUNGSVERZEICHNIS

ALB ..... *Application Load Balancer*
API ..... *Application Programming Interface*
AWS ..... *Amazon Web Services*
CaaS ..... *Container-as-a-Service*
ECR ..... *Elastic Container Registry*
ECS ..... *Elastic Container Service*
FaaS ..... *Function-as-a-Service*
HTTP ..... *Hypertext Transfer Protocol*
IaaS ..... *Infrastructure-as-a-Service*
IAM ..... Identity and Access Management
JSON ..... *JavaScript Object Notation*
MVC ..... *Model View Controller*
PaaS ..... *Platform-as-a-Service*
RDS ..... *Relational Database Service*
RT ..... *Response Time*
SaaS ..... *Software-as-a-Service*
VM ..... *Virtuelle Maschine*

ABBILDUNGSVERZEICHNIS

TABELLENVERZEICHNIS

LISTINGS

LITERATURVERZEICHNIS

Albuquerque Jr., L. F., Ferraz, F., S., Oliveira, F., A., Galdino, S. L. (2017). Function-as-a-Service X Platform-as-a-Service: Towards a Comparative Study on FaaS and PaaS. In. L. Lavazza et al., Twelfth International Conference on Software Engineering Advances (S. 206-212). International Academy, Research, and Industry Association.

Allen, S. K. (2017). What Every Web Developer Should Know About HTTP (3. Aufl.). OdeToCode LLC.

Amazon API Gateway (o. D.). Documentation AWS. Abgerufen am 12. 12. 2021 von https://aws.amazon.com/de/api-gateway/faqs/

Amazon API Gateway – Häufig gestellte Fragen (o. D.) Documentation AWS. Abgerufen am 07.01.2022 von https://aws.amazon.com/de/api-gateway/pricing/

Amazon API Gateway – Preise (o. D.). Documentation AWS. Abgerufen am 07.01.2022 von https://aws.amazon.com/de/api-gateway/pricing/

Amazon ECS clusters - Amazon Elastic Container Service. (o. D.). Documetation AWS. Abgerufen am 24. 10. 2021 von https://docs.aws.amazon.com/AmazonECS/latest/developerguide/clusters.html

Amazon ECS task definitions - Amazon Elastic Container Service (o. D.). Documentation AWS. Abgerufen am 23. 10. 2021 von https://docs.aws.amazon.com/AmazonECS/latest/developerguide/task_definitions.html

Amazon RDS Instance Types. Documentation AWS. Abgerufen am 11. 02. 2022 von https://aws.amazon.com/rds/instance-types/?nc1=h_ls

Artillery Docs (o. D.). Abgerufen am 06.02.2022 von https://www.artillery.io/docs/

AWS Lambda - Preise. Documentation AWS (o. D.). Abgerufen am 06.01.2022 von https://aws.amazon.com/de/lambda/pricing

AWS Stammbenutzer des -Kontos (o. D.) Abgerufen am 23.01.2022 von https://docs.aws.amazon.com/de_de/IAM/latest/UserGuide/id_root-user.html

AWS Creating a cluster using the classic console (o. D.) Abgerufen am 23.01.2022 von https://docs.aws.amazon.com/AmazonECS/latest/developerguide/create_cluster.html

Barid, A., Huang, G., Munns, C., Weinstein, O. (2017). Serverless Architectures with AWS Lambda - Overview and Best Practices [White paper]. AWS. https://d1.awsstatic.com/whitepapers/serverless-architectures-with-aws-lambda.pdf

Biswas, N. (2021). MERN Projects for Beginners: Create Five Social Web Apps Using MongoDB, Express.js, React, and Node (1.Aufl.). Apress.

Buchanan, S., Rangama, J. & Bellavance, N. (2020). Introducing Azure Kubernetes Service. (1. Aufl.). Apress.

Castro, P., Ishakian, V., Muthusamy, V. & Slominski, A. (2019). The rise of serverless computing. Communications of the ACM, 62(12), 44–54.

Chapin, J. & Roberts, M. (2020). Programming AWS Lambda. (1.Aufl.). Van Duuren Media.

Charge, M. (2020). Docker Easy: The Complete Guide on Docker World for Beginners. Independently published.

Cloud Native Computing Foundation (2018, 27. August). CNCF. 07.4.2021, https://github.com/cncf/wg-serverless/raw/master/whitepapers/serverless-overview/cncf_serverless_whitepaper_v1.0.pdf

Cook, B. (2018). Formal Reasoning About the Security of Amazon Web Services. In: Chockler, H., Weissenbacher, G.: Computer Aided Verification, (1. Aufl.). Springer Publishing, S. 38-47.

Combe, T., Martin, A. & di Pietro, R. (2016). To Docker or Not to Docker: A Security Perspective. IEEE Cloud Computing, 3(5), 54–62.

Davis, A. (2021). Bootstrapping Microservices with Docker, Kubernetes, and Terraform. A project-based guide (1. Aufl.). Manning Publications.

Deinum, M. & Cosmina, I. (2021): Pro Spring MVC with WebFlux. Berkeley, CA: Apress.

Doglio, F. (2018). REST API Development with Node.js (1. Aufl.). Apress.

Dombrovskaya, H., Novikov, B. & Bailliekova, A. (2021). PostgreSQL Query Optimization: The Ultimate Guide to Building Efficient Queries (1.Aufl.). Apress.

Eivy, A. & Weinman, J. (2017). Be Wary of the Economics of Serverless Cloud Computing. IEEE Cloud Computing, 4(2), S. 6–12.

Fitzhugh, R. (2014). vSphere virtual machine management (1. Aufl.) Packt Publishing.

Felderer, M. & Travassos, G. H. (2020). Contemporary Empirical Methods in Software Engineering. Springer Publishing.

Gilbert, K., & Caudill, B. (2019). Hands-On AWS Penetration Testing with Kali Linux: Set up a virtual lab and pentest major AWS services, including EC2, S3, Lambda, and CloudFormation. Packt Publishing.

Gupta, M. (2018). Serverless Architectures with AWS (1. Aufl.). Packt Publishing.

Gupta, B. & Dahiya, A. (2021). Distributed Denial of Service Attacks (1. Aufl.). Taylor & Francis.

How Amazon ECS manages CPU and memory resources (2020, 11. Dezember). Amazon Web Services. Abgerufen am 24. 10. 2021 von https://aws.amazon.com/blogs/containers/how-amazon-ecs-manages-cpu-and-memory-resources/

Ifrah, S. (2019). Storing, Managing, and Deploying Docker Container Images with Amazon ECR (1. Aufl.). Springer

Kanikathottu, H. (2019). Serverless Programming Cookbook: Practical Solutions to Building Serverless Applications Using Java and AWS (1.Aufl.). McGraw-Hill Osborne Media.

Kappel, J., Velte, T., Velte, J. (2009). Microsoft virtualization with Hyper-V (1. Aufl.). McGraw Hill

Lambda runtimes (o. D.). Documetation AWS. Abgerufen am 28. 01. 2022 von https://docs.aws.amazon.com/AmazonECS/latest/developerguide/clusters.html

Mallu, S. & Alvarez-Parmar, R. (2020, Mai 29). Maintaining Transport Layer Security all the way to your container: using the Application Load Balancer with Amazon ECS and Envoy. AWS App Mesh, Containers. https://aws.amazon.com/de/blogs/containers/maintaining-transport-layer-security-all-the-way-to-your-container-using-the-application-load-balancer-with-amazon-ecs-and-envoy/

Manner, J., Endreß, M., Heckel, T. & Wirtz, G. (2018). Cold Start Influencing Factors in Function as a Service. In 2018 IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC Companion) (S. 181–188). IEEE.

Marrs, T. (2017). JSON at Work. Practical Data Integration for the Web (1. Aufl.). O'Reilly.

Massé, M. (2021). REST API Design Rulebook (1. Aufl.). O'Reilly.

McKendrick, R. (2020). Mastering Docker. Enhance your containerization and DevOps skills to deliver production-ready applications (4. Aufl.). Packt Publishing.

Mouat, A. (2016). Using Docker (1. Aufl.). O'Reilly.

Mukherjee, S. (2019). Benefits of AWS in Modern Cloud. SSRN Electronic Journal.

Naylor, D., Finamore, A., Leontiadis, I., Grunenberger, Y., Mellia, M., Munafò, M., Papagiannaki, K., & Steenkiste, P. (2014). The Cost of the “S” in HTTPS. Proceedings of the 10th ACM International on Conference on Emerging Networking Experiments and Technologies.

Patterson, Scott (2019). Learn AWS Serverless Computing: A beginner's guide to using AWS Lambda, Amazon API Gateway, and services from Amazon Web Services (1. Aufl.). Packt Publishing.

Peralta, J. H. (2022). Microservice APIs in Python. Manning.

Poulton, N. (2020). Docker Deep Dive. Van Haren Publishing.

Sbarski, P. (2017). Serverless architectures on AWS. With examples using AWS Lambda (1.Aufl.). Manning Publications Co.

Schenker, G. (2020). Learn Docker – Fundamentals of Docker 19.x. Build, Test, Ship, and Run Containers with Docker and Kubernetes (2. Aufl.). Packt Publishing.

Security in AWS Lambda (o. D.). Documentation AWS. Abgerufen am 06.02.2022 von https://docs.aws.amazon.com/lambda/latest/dg/lambda-security.html

Serverless Compute Engine–AWS Fargate Pricing–Amazon Web Services. (o. D.). Amazon Web Services, Inc. Abgerufen am 24. 10. 2021 von https://aws.amazon.com/fargate/pricing/
Sharma, S. (2021). Modern API Development with Spring and Spring Boot (1. Aufl.). Packt Publishing.

Smith, B. (2015). Beginning JSON. LEARN THE PREFERRED DATA FORMAT OF THE WEB. (1. Aufl.). Apress.

Stojanovic, S. & Simovic, A. (2019). Serverless Applications with Node.js (1. Aufl.). Manning Publications Co.

Vahidinia, P., Farahani, B. & Aliee, F. S. (2020). Cold Start in Serverless Computing: Current Trends and Mitigation Strategies. 2020 International Conference on Omni-layer Intelligent Systems (COINS).

Voorhees, D. P. (2021). Guide to Efficient Software Design: An MVC Approach to Concepts, Structures, and Models (Texts in Computer Science) (1. Aufl.) Springer.

What is Amazon API Gateway? (o. D.). Documentation AWS. Abgerufen am 12. 12. 2021 von https://docs.aws.amazon.com/apigateway/latest/developerguide/welcome.html

Wei, J. & Xu, C. Z. (2011). Measuring Client-Perceived Pageview Response Time of Internet Services. IEEE Transactions on Parallel and Distributed Systems, 22(5), 773–785.

Westerveld, D. (2021). API Testing and Development with Postman: A practical guide to creating, testing, and managing APIs for automated software testing. Packt Publishing.

Wood, J. & Prasath, H. (2021). Caching data and configuration settings with AWS Lambda extensions. AWS Compute Blog. https://aws.amazon.com/de/blogs/compute/caching-data-and-configuration-settings-with-aws-lambda-extensions/

Yadav, K., Garg, M. L., Ritika (2019). Docker Containers Versus Virtual Machine-Based Virtualization. In: Ajith, A., Paramartha, D., Jyotsna, M., Abhishek, B., Soumi, D. (Hrsg.): Emerging Technologies in Data Mining and Information Security, Bd. 814. Singapore: Springer Singapore , S. 141–150.

Yudin, Art (2020). Building Versatile Mobile Apps with Python and REST (1.Aufl.). Apress.