HAW HAMBURG

BACHELORTHESIS
Lucie aus der Wieschen

Serverless – The Good, the Bad and the Ugly

FAKULTÄT TECHNIK UND INFORMATIK
Department Informatik

Faculty of Computer Science and Engineering
Department Computer Science

HOCHSCHULE FÜR ANGEWANDTE WISSENSCHAFTEN HAMBURG
Hamburg University of Applied Sciences

Lucie aus der Wieschen

Serverless – The Good, the Bad and the Ugly

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung
im Studiengang *Bachelor of Science Angewandte Informatik*
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer: Prof. Dr. Stefan Sarstedt
Zweitgutachter: Prof. Dr. Klaus-Peter Kossakowski

Eingereicht am: 31. Oktober 2021

Lucie aus der Wieschen

Thema der Arbeit

Serverless – The Good, the Bad and the Ugly

Stichworte

Serverlos, Entwicklung, Architektur, Cloud, FaaS, BaaS, DevOps, AWS Lambda

Kurzzusammenfassung

Serverless ist eine Evolutionsstufe des Cloud Computing, welche auf FaaS und BaaS basiert. Es ermöglicht einen verstärkten Fokus auf die Anwendungslogik, während ein Cloudanbieter die Skalierung und andere infrastrukturelle Aufgaben nach dem pay-as-you-go Prinzip übernimmt. Diese Arbeit versucht eine realistische Einschätzung des Hypes um Serverless vorzunehmen, sowohl in der Theorie als auch in der Praxis. Die wichtigste Erkenntnis war, dass Serverless in zwar einigen Punkten überbewertet ist, aber wenige unüberwindbare Grenzen hat.

Lucie aus der Wieschen

Title of Thesis

Serverless – The Good, the Bad and the Ugly

Keywords

Serverless, Development, Architecture, Cloud, FaaS, BaaS, DevOps, AWS Lambda

Abstract

Serverless is an evolutionary stage of Cloud Computing, which is based on FaaS and BaaS. It allows greater focus on the application logic, while the cloud provider is taking care of scaling and most other operational tasks, following a pay-as-you-go principal. This thesis tries to realistically asses the hype around Serverless, both in theory and practice. The main outcome was that Serverless is overhyped in certain aspects but only has a few hard limitations which cannot be overcome.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1 Einleitung

Seit den Anfängen des Cloud Computing hat sich ein Trend entwickelt, die operativen Tätigkeiten abzugeben und den Fokus mehr auf die Anwendungslogik zu legen. Serverless Computing ist eine neue Art Cloud Computing zu nutzen, welcher aktuell viel Aufmerksamkeit gewidmet wird. Es verspricht neben weiter erhöhtem Fokus auf die Anwendungslogik, eine enorme Skalierungs- und Kosteneffizienz. Serverless basiert auf Function-as-a-Service und Backend-as-a-Service und ist die nächste Evolutionsstufe des X-as-a-Service Paradigma der Cloud.
Im Jahr 2014 wurde die Idee von der Amazon AWS Plattform konkretisiert und seitdem stetig weiterentwickelt, sowie von anderen Cloudanbietern adaptiert. Die Einsatzmöglichkeiten der Serverless Architektur sind vielfältig und bieten eine hohe Flexibilität. In der Theorie ist Serverless damit in fast jedem Bereich anwendbar und könnte eine enorme Disruption der aktuellen IT-Landschaft auslösen.
Auf der anderen Seite ist es eine relativ neue Technologie, die gerade einen Hype erlebt. Diese Arbeit soll daher eine realistische Einschätzung der Architektur vornehmen, die versucht den Hype zu umgehen und Serverless aus dem Produktivitäts-Blickwinkel zu betrachten. Jede aufstrebende Technologie bringt neben neuen Möglichkeiten und Vorteilen auch Probleme und Einschränkungen mit sich. Diese Arbeit soll zu den Problemen hinarbeiten und dabei anschaulich vom Positiven über das Problematische bis hin zu den Grenzen gehen. Dabei sollen sowohl theoretisch als auch praktisch, mögliche Optimierungen und Limitierungen erkundet werden.
Es stellt sich die Frage, ob wirklich eine „Server-Revolution“ ausgelöst wurde oder ob Serverless nur ein weiteres Tool für speziell passende Anwendungen ist. Können die hohen Erwartungen erfüllt werden oder bestehen letztendlich doch gravierende Einschränkungen?

1.1 Motivation

Häufig entsteht um neue Technologien ein Hype, welcher zu einer Überbenutzung und einer Fehleinschätzung der Einsatzzwecke führen kann. Ein ähnliches Beispiel für diesen Sachverhalt sind Microservices, welche bereits vermehrt auf diese Problematik untersucht wurden. Seit einigen Jahren erlebt nun Serverless Computing einen Hype und verspricht viele Vorteile, für die es sich lohnt, die Technologie genauer zu untersuchen. Abb. 1.1 zeigt Gartners Hype Cycle aus dem Jahr 2017, in welchem Serverless 2 - 5 Jahre vor dem Höhepunkt des Hypes eingeordnet wurde. Es ist also im Jahr 2021 besonders spannend herauszufinden, wo Serverless sich gerade befindet und wie das Plateau der Produktivität aussehen könnte.

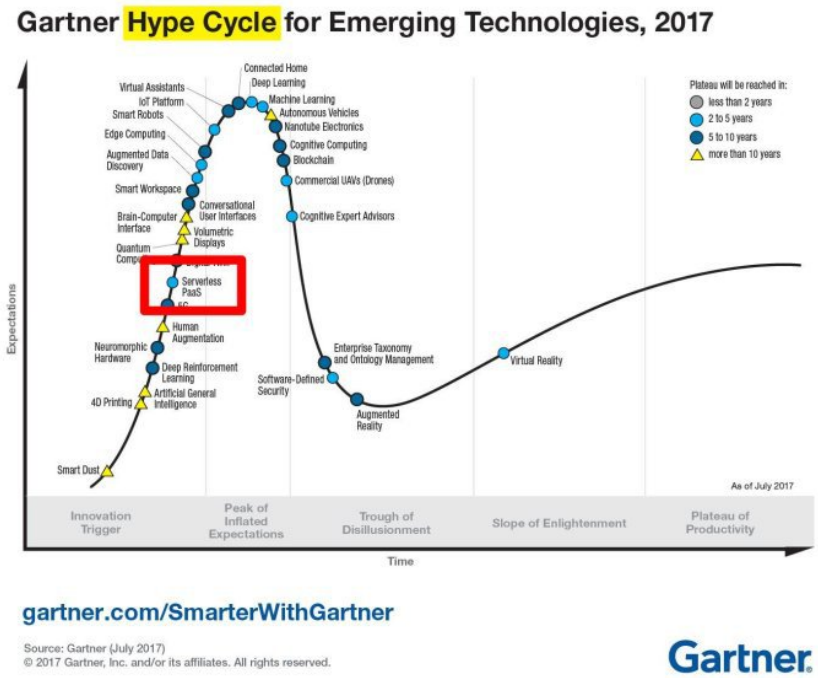


Abbildung 1.1: Hype Cycle für aufkommende Technologien, Gartner 2017

Die Idee sich mit Serverless zu beschäftigen, wurzelt in dem spannenden Ansatz, die Entwicklung möglichst effizient von den operativen Tätigkeiten zu trennen. Diese haben weniger mit der eigentlichen fachlichen Logik und damit dem Software Engineering zu tun, weshalb viele Entwickler:innen vermeiden möchten operative Tätigkeiten auszuführen, während gleichzeitig die DevOps Kultur gestärkt wird. Das Ziel ist es, dafür viele

aktuell relevante Technologien und Paradigmen kennenzulernen und auszuprobieren, um einen möglichst umfassenden Blick auf Serverless zu bekommen.

1.2 Kapitelüberblick

Die Arbeit teilt sich auf in Grundlagen, Fallbeispielbeschreibung, einen dreistufigen Hauptteil, welcher mit praktischer Umsetzung untermauert wird. Am Ende stehen die Diskussion sowie das Fazit. Mit der Einleitung umfasst die Arbeit also acht Kapitel.
Kapitel 1 enthält die Einleitung in das Thema Serverless und die dahinterstehende Motivation und Fragestellung.
Kapitel 2 beschäftigt sich in der ersten Hälfte mit den Grundlagen, welche für das Verständnis von Serverless und dessen Entstehung benötigt werden. Die zweite Hälfte umfasst eine Definition sowie Patterns und Best Practices.
Kapitel 3, Fallbeispiel, entwirft die später entwickelte Anwendung und geht dabei sowohl auf die Anforderungen als auch auf die gewählten Technologien ein.
Kapitel 4, 5 und 6 sind der Hauptteil der Arbeit. Sie entsprechen damit dem Titel The Good, The Bad, and The Ugly. Das vierte Kapitel beschreibt damit die Vorteile und sinnvollen Einsatzgebiete von Serverless. Kapitel fünf und sechs beschreiben die Herausforderungen und Probleme, welche sich unter The Ugly zu Grenzen verschärfen. In jedem dieser drei Kapitel steht am Ende die praktische Umsetzung, welches Teile der jeweils behandelten Themen untersucht und Ergebnisse darstellt.
In **Kapitel 7** wird eine abschließende Diskussion der vorherigen Kapitel gehalten, in welcher die Frage behandelt wird, ob sich Serverless lohnt und abschließend die praktische Umsetzung evaluiert wird.
Kapitel 8 schließt die Arbeit ab, fasst Ergebnisse und Erkenntnisse zusammen und gibt einen kurzen Ausblick.

2 Grundlagen

Einführend in das Thema Serverless Computing sollen einige Grundlagen erläutert werden. Diese Informationen werden benötigt, um in den nachfolgenden Kapiteln die aufgeführten theoretischen und praktischen Erkundungen besser verstehen zu können.
Serverless Computing basiert auf der Cloud Infrastruktur, weshalb zuerst auf die Geschichte der Cloud eingegangen wird, sowie das Everything-as-a-Service Paradigma, welches aktuell in Serverless resultiert. Danach soll Serverless selbst abgegrenzt, und die grundsätzlichen Prinzipien und Patterns vorgestellt werden.

2.1 Geschichte des Cloud Computing

In den letzten Jahrzehnten haben sich Begriffe wie die Cloud oder das Cloud Computing in der Informatik etabliert. Die gesamte Softwareentwicklung wurde auf ein neues Level gehoben und verspricht, zahlreiche Probleme besser lösen zu können. Viele der weltweit größten Firmen bieten mittlerweile Cloud Plattformen an, darunter Amazon, Google und Microsoft.
Das amerikanische National Institute of Standards and Technology (NIST) bietet eine vielzitierte Definition [S. Reinheimer, 2018]. NIST beschreibt Cloud Computing als ein Modell, welches uneingeschränkten, komfortablen on-demand Zugriff auf geteilte Ressourcen (Netzwerke, Speicherplatz, Rechenleistung etc.) bietet. Diese Dienste können schnell und mit möglichst wenig Management und Provider-Interaktion genutzt werden [P. Mell, 2011]. Des Weiteren werden dort folgende fünf essentielle Charakteristika genannt:

1. Bereitgestellte Dienste können jederzeit (on-demand) selbst gesteuert werden (z.B. Speicherplatz oder Serverzeit).

2. Bereitgestellten Dienste können über das Netzwerk erreicht werden und entsprechen den von verschiedenen, gängigen Plattformen benötigten Standards.

3. Ressourcen werden vom Provider in einer Multi-Tenancy Architektur, also mehrere Kunden auf einer Plattform, ohne deren Einfluss aufgeteilt.

4. Dienste skalieren elastisch und suggerieren quasi unendliche Ressourcen, welche zu jeder Zeit angefordert werden können.

5. Cloud Systeme helfen die Nutzung zu optimieren, z.B. über angepasste Zahlungsmodelle und Transparenz (Monitoring, Reporting etc.).

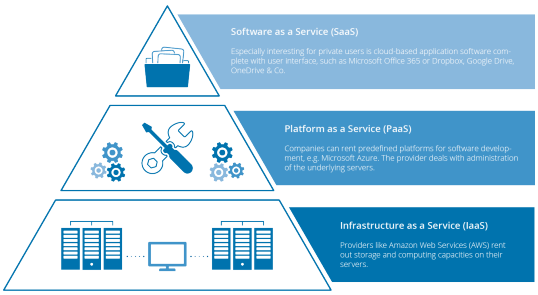
[P. Mell, 2011]

Ein weiterer Punkt der NIST Definition ist die Organisation von Clouds. Es gibt private, öffentliche, hybride und sog. Community Clouds. Diese haben diverse Vor- und Nachteile sowie Einsatzmöglichkeiten. Für Serverless sind vor allem die Public Clouds relevant. Dabei teilen sich mehrere Anwender dieselbe Infrastruktur, welche von einem unternehmensunabhängigen Provider gegen Bezahlung zur Verfügung gestellt wird [S. Reinheimer, 2018]. Serverless Computing nutzt die Dienste der öffentlichen Clouds, wie AWS, GCP oder Azure, kann aber auch mit einer privaten Cloud genutzt werden, wobei die sonst angebotenen Dienste selbst implementiert und verwaltet werden müssen (on-premise).

Technisch definiert geht es beim Cloud Computing um die Virtualisierung von Hardware und Rechenzentren. Dadurch werden drei Service-Ebenen ermöglicht: Software-as-a-Service, Platform-as-a-Service und Infrastructure-as-a-Service [S. Reinheimer, 2018]. Diese sollen im nächsten Abschnitt als Grundbausteine auf dem Weg zu Serverless betrachtet werden.

2.1.1 Everything-as-a-Service Paradigma

Die Informatik beinhaltet aktuell zahlreiche Abstraktionen von Cloud Services, welche als Serviceebenen bezeichnet werden. Nach dem Buch “What is Serverless” von M. Roberts und J.Chapin lassen sich deren Anfänge am ehesten auf die Veröffentlichung der AWS Elastic Compute Cloud (EC2) im Jahr 2006 datieren. EC2 gilt als eines der ersten Infrastructure as a Service Produkte (IaaS) [Roberts and Chapin, 2017]. IaaS ist die unterste der drei grundlegenden Abstraktionsebenen, welche in Abbildung 2.1 dargestellt sind, und im Folgenden näher erläutert werden sollen.



Quelle: https://de.cleanpng.com/png-ieh7n5/download-png.html

Abbildung 2.1: IaaS, PaaS und SaaS im Verhältnis

Infrastructure as a Service (IaaS)

IaaS entspricht der physikalischen Infrastruktur. Ihre Hauptaufgabe ist die dynamische Zuweisung von Ressourcen (wie Speicher- o. Prozessorkapazitäten), um dem jeweiligen Nutzer auf Abruf zur Verfügung zu stehen. IaaS kommt hauptsächlich zum Einsatz, wenn die Anwendungslandschaft für klassische Hardware zu komplex ist. So kann die Infrastruktur immer möglichst optimal an den Use Case angepasst werden und die Betreiber sparen sich Erwerb und komplexe Verwaltung eigener Rechenzentren [S. Reinheimer, 2018]. Die gesammelten Vorteile von IaaS dienen als Grundlage für die Vorteile der anderen Ebenen und werden an dieser Stelle nach dem Buch “What is Serverless?” dargestellt.

Reduzierte Arbeitskosten Unternehmen müssen sich nicht mehr selbst um das Management physikalischer, eigener Server kümmern, für welches zuvor spezielle Teams in den Datencentern benötigt wurden. Die Verwaltung liegt nun komplett bei den jeweiligen Anbietern.

Verringertes Risiko Das Risiko verringert sich vor allem im Hinblick auf die Hardware. Dadurch, dass die Server nicht mehr die eigenen sind, müssen die Folgen von kritischen Vorfällen oder Ausfällen nicht mehr selbst getragen werden. Ziel ist es die Auseinandersetzung mit diesem Problem so stark zu verringern, dass z.B. lediglich eine neue Maschine

angefragt werden muss oder Ausfälle durch die enorme Rechenleistung der Provider kaum auffallen.

Reduzierte Infrastrukturkosten Cloud Provider bieten Zahlungsmodelle angepasst an die Nutzung an. Dadurch kann in den meisten Fällen gegenüber der Anschaffung von eigenen Servern gespart werden, vor allem wenn einige Instanzen nicht konstant (mehrere Monate oder Jahre) gebraucht werden, sondern eher für wenige Tage bis Wochen benötigt werden, weil z.B. bei hohen Auslastungen hochskaliert werden muss. Dadurch werden die eigentlich nötigen initialen Investitionsaufwendungen zu Betriebskosten, was eine größere finanzielle Flexibilität ermöglicht.

Skalierung Die Skalierung ist direkt an die reduzierten Infrastrukturkosten geknüpft. Vorherige Planung von benötigten Ressourcen und damit die Anschaffung von anfangs eventuell wenig oder gar nicht genutzten Servern fällt weg. Es kann mit einer günstigen, schwachen Instanz begonnen, und nach Bedarf hochskaliert werden.

Reduzierte Vorlaufzeit Die Vorlaufzeit beschreibt die Zeit von der Idee zu dem ersten lauffähigen Code. Ohne die Anschaffung und Aufsetzung von eigenen Servern verringert sich diese Zeit von bis zu mehreren Monaten auf einige Minuten. Dies unterstützt das Experimentieren und verkürzt Produktentwicklungszyklen.

(Vgl. [Roberts and Chapin, 2017])

Die Nutzung von IaaS wird auch als 'Infrastructural Outsourcing' bezeichnet, welches die Vorteile von IaaS spiegelt, aber noch näher beschreibt was unter Infrastruktur fällt. Dazu gehört demnach all das, was nicht spezifisch für das Projekt ist. Also physikalische Ressourcen wie Strom bis hin zu abstrahierbaren Funktionen wie Authentifikation [Roberts and Chapin, 2017].

Platform as a Service (PaaS)

Die nächste Evolutionsstufe der öffentlichen Clouds ist Platform as a Service, welches direkt über IaaS liegt und zusätzlich das Betriebssystem liefert [Roberts and Chapin,

2017]. Dazu werden sogenannte Programming- und Execution Environments zur Verfügung gestellt, in welchen dann Software in der jeweiligen Programmiersprache entwickelt werden kann [S. Reinheimer, 2018]. Entwickler:innen brauchen dadurch nur noch ihre Anwendungen fachlich zu implementieren und bereitzustellen ohne sich mit Betriebssystem Installation, systemlevel Monitoring oder ähnlichem beschäftigen zu müssen [Roberts and Chapin, 2017].
PaaS kann entweder privat oder öffentlich gehostet werden. Alternativ zu PaaS werden heute häufig Container eingesetzt (Software wie Docker), da beide auf unterliegenden virtuellen Maschinen basieren und sich damit gut substituieren lassen. Da es auch für Container, Cloud basierte Lösungen gibt, existiert zusätzlich die Bezeichnung Containers as a Service (CaaS) [Roberts and Chapin, 2017].
Öffentliches PaaS oder CaaS sind weitere Abstraktionen von Infrastructural Outsourcing. Die Vorteile bleiben also grundsätzlich die von IaaS. Zusammengefasst sind diese Services generische Umgebungen in denen fallspezifische Software laufen kann (= Compute as a Service) [Roberts and Chapin, 2017].

Software as a Service (SaaS)

SaaS stellt die höchste Abstraktionsebene dar und liefert damit vollständige, standardisierte Anwendungen. Diese Bereitstellung richtet sich damit an die Endnutzer:innen und nicht zwangsläufig die Entwickler:innen als Kontrast zu PaaS. Technisch läuft dies über einen Anbieter, der die Software online bereitstellt, wodurch keine lokale Installation notwendig ist [S. Reinheimer, 2018].
Die größte Einschränkung für Entwickler:innen entsteht durch die Multi-Tenant Architektur der Cloud, welche diese Anwendungen in den Anpassungs- und Integrationsmöglichkeiten einschränkt. Trotzdem gibt es viele standardisierte Businessprozesse, die in SaaS ausgelagert werden können, wie z.B. für technische Tools wie GitHub oder Prozesse im Sales Bereich [Roberts and Chapin, 2017].

2.2 Bestandteile von Serverless

Zwei weitere Serviceebenen sind Function as a Service und Backend as a Service, auf welchen Serverless Computing, im Folgenden einfach Serverless genannt, basiert. Serverless

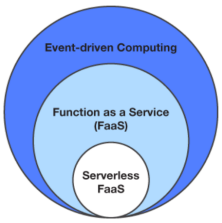
ist eine Weiterentwicklung des Cloud Computing und die nächste Evolutionsstufe des Infrastructual Outsourcing. Es erbt damit die fünf grundsätzlichen, unter IaaS erläuterten Vorteile [Roberts and Chapin, 2017].
Serverless ist ein noch relativ junger Ansatz, der vor allem durch den Fortschritt der Technologie geformt wurde. Die prägenden Grundideen der Architektur sind schon deutlich älter und wurden letztendlich im Jahr 2014 durch Amazon's AWS Lambda konkretisiert [AWS, 2014] und stetig weiterentwickelt.

2.2.1 Function as a Service (FaaS)

Die Serverless Architektur basiert in erster Line auf Funktionen und damit auf FaaS. Häufig werden die beiden Begriffe sogar synonym verwendet. Dies ist allerdings ein Fehlschluss, da Serverless nur ein Ausführungsmodell ist, um FaaS zu realisieren. Das momentan am meisten genutzte FaaS-Produkt ist AWS Lambda [Stalcup, 2021], welches im späteren Verlauf der Arbeit Anwendung findet.
FaaS hat den Fokus auf der Infrastruktur und beschreibt eine neue Art des Bauens und Bereitstellens der serverseitigen Software. Die Besonderheit dabei ist, dass nur noch einzelne Funktionen bereitgestellt werden, anstelle der ganzen Anwendung. Der eigentlich notwendige Container oder die VM, in welchen die Anwendung bzw. der Systemprozess bisher ausgeliefert wurde fällt bei Serverless FaaS weg. Lediglich die Logik der Anwendung, verpackt in einzelne Funktionen, wird auf die jeweilige FaaS-Plattform (z.B. AWS Lambda) hochgeladen.
Es besteht auch hier die Möglichkeit FaaS Angebote privat, auf der eigenen Hardware zu betreiben [Roberts and Chapin, 2017].
Aus FaaS ergibt sich eine weitere Besonderheit von Serverless. Nachdem die Funktionen hochgeladen wurden, verbrauchen sie keine weiteren Ressourcen. Zum Ansprechen der Funktionen haben diese Trigger, welche über Events aktiviert werden. Dadurch ist es mit Serverless möglich, sehr viel Kapazität und damit Geld zu sparen, was insbesondere der optimierten Nutzung, also der fünften Charakteristik vom Cloud Computing, zugutekommt.

Eventbasierte Programmierung

Die eventbasierte Programmierung ist die Basis der FaaS Ebene. Die eben erwähnten Trigger, werden auch Event Sources genannt und können sehr vielfältig sein. Amazon AWS bietet beispielsweise mittlerweile über 140 integrierbare Eventquellen an [AWS, 2021b]. Diese Quellen können synchron oder asynchron agieren. Ein gängiges Beispiel für synchrone Kommunikation ist eine HTTP-API, während ein Nachrichten Bus ein Beispiel für asynchrone ist.
In der Abb. 2.2 von Amazon selbst, wird verdeutlicht wie die eventbasierte Programmierung, FaaS und Serverless im Verhältnis stehen. Der größte Unterschied von Serverless FaaS zu normalem FaaS ist demnach, dass bei Serverless keine Container oder VMs mehr benötigt werden und der Anbieter für Zuverlässigkeit und Skalierbarkeit sorgt [AWS, d].



Quelle: siehe [AWS, d]

Abbildung 2.2: Zusammenhang von eventbasiert, FaaS und Serverless

Zustandslosigkeit

FaaS Funktionen werden als Zustandslos bezeichnet. Genauer gesagt existiert kein persistenter Zustand bezüglich der Daten im lokalen Speicher, in Variablen etc. Daraus folgt, dass es keine Garantie gibt, dass zwischen mehrfachen Aufrufen einer Funktion der vorherige Zustand erhalten bleibt. Wird ein Zustand für das Programm benötigt, muss dieser außerhalb der Funktionsinstanz persistiert werden [Roberts, 2018].

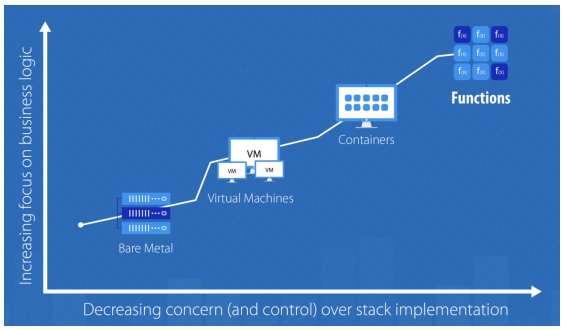
2.2.2 Backend as a Service (BaaS)

Ein weiterer Teil der Serverless Architektur ist BaaS, welches die Differenzierung von Serverless und FaaS unterstreicht.
BaaS soll serverseitige Komponenten, die normalerweise selbst geschrieben werden mit

vorgefertigten Services ergänzen bzw. ersetzen. Der Umfangsreichtum der Funktionen erinnert damit eher an SaaS als an die auf Infrastruktur basierenden Ebenen. Allerdings geht es hier nur darum einzelne Teile der eignen Anwendung mit externen Produkten zu realisieren. Dies funktioniert meist über eine API und kann dann z.B. für Datenbanken oder Authentisierung eingesetzt werden. Beispiele dafür sind Google Firebase oder Amazon's Cognito Service. Es gibt natürlich noch viele weitere Möglichkeiten BaaS einzusetzen. Gerade in der Mobil- oder Frontend Entwicklung sind die angebotenen Dienste beliebt, beschränken sich aber auf Grund der Diversität der angebotenen Services nicht mehr darauf.
(Vgl. [Roberts and Chapin, 2017])

2.3 How to Serverless

Serverless kann also als natürliche Evolution der Cloud Services beschrieben werden, ist eine eventbasierte Architektur, welche sich vor allem durch FaaS, aber auch durch BaaS definiert und ist nicht als Softwarearchitektur zu verstehen.
Der sich in den X-as-a-Service Paradigmen abzeichnende Trend geht dahin, immer mehr abstrahierbare Funktionalität und Infrastruktur auszulagern. Serverless ist ein weiteres Angebot in diese Richtung, wobei lediglich die fachspezifische Logik, gekapselt in einzelne Funktionen, übrig bleibt [Roberts and Chapin, 2017]. Abbildung 2.3 zeigt die Entwicklung grob bildlich.



Quelle: siehe [Shachar, 2018]

Abbildung 2.3: Steigender Fokus auf die Businesslogik

Die Weiterentwicklung einer Technologielandschaft setzt immer ein gewisses Umdenken voraus, welches bei Serverless ebenfalls in großem Maße gefordert ist. Bereits im Entwurf

werden neue Herangehensweisen benötigt. Andere Modelle, wie Microservices, setzen für sinnvollen Einsatz ebenfalls ein Umdenken voraus, allerdings ist die Veränderung bei Serverless noch drastischer. Die Logik steht allein und ist normalerweise in feingranulare Deploymenteinheiten verpackt, welche dementsprechend passend entworfen werden müssen [Roberts and Chapin, 2017].
Um diese Veränderung besser zu verstehen, soll im Folgenden auf die Prinzipien von Serverless und die Besonderheiten serverloser Anwendungen eingegangen werden. Danach soll deutlich gemacht werden, warum Serverless Funktionen nicht mit Microservices auf einer Ebene stehen, da dies eine häufige Fehleinschätzung ist. Im Anschluss werden dann mögliche Patterns und Antipatterns, sowie Best Practices vorgestellt.

2.3.1 Prinzipien der Serverless Architektur

Serverless basiert darauf, die Serververwaltung vor den Entwickler:innen zu verstecken und Code On-Demand und automatisch skaliert laufen zu lassen. Dabei ist das Zahlungsmodell so angepasst, dass nur für die Laufzeit abgerechnet wird. Zur weiteren Abgrenzung von vorherigen Ansätzen werden nun 5 Kernprinzipien für Serverless Anwendungen nach Sbarski dargestellt. Sie wurden gegenüber den alternativen 10 Prinzipen des Serverless Manifesto (siehe z.B. [Framework, a]) gewählt, da diese sich deutlicher auf Serverless und nicht nur auf FaaS beziehen.

1. **Verwende Compute Services nach Bedarf** (keine Server): Ganz nach dem Prinzip von FaaS, soll Rechenleistung nur dann verwendet werden wenn sie gebraucht wird. Daher zielt dieses Prinzip darauf ab, den Code in einzelne Funktionen zu kapseln und keine eigenen Server, Container oder VMs zu betreiben (mögl. Anbieter: AWS Lambda, Google Cloud Functions u. Azure Functions). Diese Funktionen können jede gängige Aufgabe übernehmen und dabei zusätzlich mit anderen Services, wie Datenbanken, oder anderen Funktionen, z.B. über Messaging Systeme interagieren.

2. **Schreibe zustandslose Funktionen, die nur einen Zweck erfüllen**: Funktionen sollten möglichst nach dem Single-Responsibility-Prinzip (SRP) entworfen werden. Die einzelnen Funktionen sind dann robuster, leichter zu Testen und gut verständlich. Die Zustandslosigkeit ist in erster Linie für die Skalierbarkeit wichtig. Funktionen sollten nicht abhängig von anderen Ressourcen oder Prozessen sein, sondern sich nur auf die aktuelle Session beziehen.

3. **Designe push-basierte, Event getriebene Lösungen**: Nach Sbarski sind eventbasierte Lösungen, die flexibelsten und stärksten Lösungen für serverlose Anwendungen. Dabei stoßen sich die Prozessschritte selbst an und Events müssen nicht eigenständig gepulled werden. Auch wenn solche Lösungen nicht immer möglich oder gewollt sind, sind sie trotzdem meist weniger komplex und kostengünstiger als die Alternativen.
   Die Gedanken von Sbarski folgen damit auch dem typischen Paradigma eventbasierter Programmierung, "Choreography over Orchestration". Funktionen sind nur lose durch Events gekoppelt und haben ihre eigene Funktionalität.

4. **Entwickle umfangreiche Frontends** (Thick Clients): Wie oben erwähnt werden Funktionen nach Laufzeit und/oder benötigter Rechenleistung abgerechnet, wodurch sich mit mehr Last bei Client Kosten einsparen lassen. Durch die Abgabe von Aufgaben verbessert sich zudem die User Experience, da Latenzen verringert werden. Durch die heutigen technischen Möglichkeiten können Frontends selbst mit Drittanbieterservices kommunizieren und Logik ausführen. Natürlich hat diese Umschichtung auch Grenzen, da es einige Funktionen gibt, die aus Sicherheits- oder Datenschutzgründen nicht clientseitig ausgeführt werden sollten.

5. **Nutze Services von Drittanbietern**: "Don't build for the sake of building, [...] stand on the shoulders of giants". Wenn es gute Angebote der Cloud Provider oder von Drittanbietern gibt und diese sinnvoll bezüglich Kosten und Nutzen, sowie Risiken abgewogen wurden, sollten diese eingesetzt werden.

(Vgl. [Sbarski, 2017])

2.3.2 Serverless vs. Microservices

Sowohl Microservices als auch Serverless stammen von serviceorientierten Architekturen (SOA) ab, weshalb sie sich viele Prinzipien mit ihnen teilen. Dazu gehören z.B. Wiederverwendbarkeit, Autonomie und die Granularität der Services. Sie versuchen dabei aber beide, die eigentliche Komplexität von SOAs zu verringern [Sbarski, 2017]. Microservices wurden dahingehend bereits ausführlich untersucht und zeigen, dass die Idee von reduzierter Komplexität häufig eine Fehleinschätzung ist [1].
Den großen Unterschied bei Serverless macht die Flexibilität. Die Funktionen können

[1] Nähere Informationen dazu sind z.B. in den Blockeinträgen von Uwe Friedrichsen über die Microservice Fallacies zu finden

genauso gestaltet werden wie Microservices, müssen sie aber nicht [Sbarski, 2017]. Das bedeutet, dass Serverless die negativen Aspekte mit Microservices zwar eigentlich gemein hat, aber diese auch zu einem gewissen Grad umgangen werden können. Trotzdem hat das Zerteilen des Codes in Funktionen natürlich Nachteile, die sich mit denen von Microservices überlappen. Serverless ist keine strikte Anwendungsarchitektur, die wie Microservices relativ genau vorgibt wie die Anwendung strukturiert sein sollte. Um diesen Fakt zu untermauern, folgen nun verschiedene mögliche Patterns für Serverless Anwendungen, die zeigen, dass verschiedenste Modelle möglich sind und die Vor- und auch Nachteile sich damit mannigfaltiger darstellen.

2.3.3 Patterns und Best Practices

Patterns können Muster für verschiedenste Bereiche der Entwicklung vorgeben. Serverless ist relativ jung und hat aus diesem Grund keine lang etablierten Muster und Empfehlungen für die Entwicklung. Allerdings sind in den letzten Jahren einige Ansätze entstanden. Serverless Funktionen sind in der Gestaltung theoretisch sehr frei und können dann jeweils andere Vor- oder Nachteile mit sich bringen. Daher ist es neben Mustern sinnvoll, Best Practices für die Entwicklung aufzustellen. Paul Johnston, einer der Mitbegründer der Konferenz ServerlessDays, hat in einem Artikel einen Leitfaden für gute serverlose Anwendungen beschrieben. Die Empfehlungen beziehen sich in erster Linie auf den Erhalt der Skalierbarkeit, welche eine der Hauptgründe für den Einsatz von Serverless ist. Typischerweise sind Funktionen sehr feingranular gehalten. Oft ist jede Funktionalität bzw. jeder Job in einer separaten Funktion isoliert. Die Funktionen sollten also möglichst so gestaltet sein, dass sie nur eine Aufgabe erfüllen. Dazu gehört auch, dass eine Funktion möglichst nur eine HTTP Route bedienen sollte. Nur so kann die Skalierung eines spezifischen Teils funktionieren und die Funktionen selbst bleiben leicht verständlich [Johnston, 2018]. Wird dieses Muster verfolgt, können sie in Produktion sehr sicher und agil sein. Natürlich ergibt sich aus dieser Aufteilung eine sehr große Anzahl an einzelnen Funktionen die verwaltet werden müssen [Hefnawy, 2017]. Neben dieser recht typischen Einteilung sind auf Grund der großen Flexibilität noch viele weitere möglich. Beispielsweise können die Funktionen auch als einzelne Microservices entworfen werden, was dann die meisten Vor- und Nachteile dieser mit sich bringt, bis auf die nochmal verstärkte Abgabe von infrastrukturellen Aufgaben an den Provider.
Ein weiteres Entwurfsmuster könnte eine monolithische Funktion sein. Serverless setzt beim Schnitt der Funktionen keine Grenzen, sondern liefert schlicht die infrastrukturelle

Grundlage für Deployment, Skalierung usw. Allerdings kann eine monolithische Funktion schnell an die vom Provider gesetzten Limits bezüglich der Funktionsgröße oder Ausführungszeit stoßen [Hefnawy, 2017]. Dies würde eine monolithische Umsetzung von Serverless unmöglich machen. AWS geht dabei so weit, das monolithische Pattern als Antipattern zu bezeichnen, da die eigentlichen Vorteile verloren gehen und die allgemeine Entwicklung eher erschwert wird, zudem widerspricht dieses Muster dem zweiten Prinzip von Serverless, zustandslose, einen Zweck erfüllende Funktionen zu schreiben. Als Kompromiss wird ein Microservice Pattern vorgeschlagen, welches durch eine Zerlegung des Monolithen erreicht werden kann. Weitere Antipatterns sind die Nutzung von Funktionen als Orchestrator, was dem dritten Serverless-Prinzip widersprechen würde. Rekursive Muster oder Muster, in welchen sich Funktionen gegenseitig aufrufen sollten ebenfalls vermieden werden, da beides negative Auswirkungen auf Performance und Kosten haben kann [Beswick, 2021], wie dass die Entwicklung unnötig erschwert wird und sich die Kosten verdoppeln [Johnston, 2018].

Weitere Best Practices sind zum einen, dass die Nutzung von Bibliotheken im Code möglichst komplett unterlassen werden sollte. Viele Abhängigkeiten haben einige Nachteile, während bei Serverless speziell die Startzeit, insbesondere der erste sogenannte Cold Start signifikant länger wird. Zum anderen sollten verbindungsbasierte Services dringend vermieden werden, da jede Funktion eine Verbindung aufbauen müsste, was das Skalieren und die Latenzen negativ beeinflusst. Damit zusammenhängend weist er darauf hin, dass die komplette Datenschicht überdacht werden muss. Die Daten sollten immer “fließen”, Funktionen sollten zustandslos sein und Datenbanken sollten nicht klassisch relational sein, sondern ebenfalls verteilt, mit optimierter Leseform usw.
Des Weiteren funktionieren serverlose Applikationen nach Johnston am besten, wenn die Kommunikation asynchron verläuft. Also über Nachrichten Busse und Queues. Insgesamt sollten Prinzipien verteilter Systeme erlernt und beachtet werden. Durch asynchrone Kommunikation können Abhängigkeitsketten gebrochen werden, wodurch beispielsweise Ausfälle nicht mehr so viel ausmachen. Zuletzt weist er darauf hin, dass immer bedacht werden sollte wie eine Anwendung skaliert. Wird die Anwendung gebaut, ohne auf diesen Faktor zu achten, kann dies zu Problemen führen. Beispielsweise kann bei einer größeren Last die Nebenläufigkeit der Funktionen stark erhöht werden oder bei falschem Einsatz das Limit der möglichen Verbindungen erreicht werden. Es ist essentiell von Anfang an die Anwendung auch unter großer Last zu betrachten und nicht davon auszugehen, dass alle Aspekte problemlos mit skalieren [Johnston, 2018].

3 Fallbeispiel

Zur Auseinandersetzung mit Serverless ist es sinnvoll eine Beispielanwendung zu entwickeln, um eigene Erfahrungen zu sammeln und Versuche durchführen zu können. In diesem Kapitel soll es um die Konzeption dieser Anwendung gehen, damit in den darauffolgenden Kapiteln klar ist, worauf sich die Aussagen bezüglich der praktischen Umsetzung beziehen. Es soll zuerst geklärt werden was die Fachlichkeit der Beispielanwendung ist. Danach soll eine Referenzarchitektur vorgestellt, und die dafür gewählten Technologien erläutert werden.

3.1 Analyse und Anforderungen

Im praktischen Teil der Arbeit besteht die Aufgabe darin, eine strukturell ausreichend anspruchsvolle Anwendung entwickeln. Da es zeitlich nicht möglich ist alle Kombinationsmöglichkeiten in einer angemessenen Ausführlichkeit umzusetzen, wird sich die praktische Erkundung auf einige aktuell relevante Technologien, unter Berücksichtigung des gegebenen kostenlosen Kontingents, beschränken.

Die angestrebte Anwendung muss möglichst viele verschiedene Anforderungen abdecken und aus mehreren, miteinander kommunizierenden Subkomponenten bestehen. Damit das zu entwickelnde System diesen Ansprüchen gerecht wird, fiel die Entscheidung auf einen Webshop. Diese Domäne kommt in der Realität häufig vor und kann beliebig einfach oder komplex gestaltet werden. Zudem kann hier der Fokus auf der Umsetzung der Architektur und der Versuche liegen, ohne das tiefergehendes Wissen in der jeweiligen Branche vorhanden sein muss.
Ein Shop kann viele verschiedene Komponenten besitzen, unter anderem gehört dazu ein User Interface. Das Interface an sich spielt für die Arbeit keine große Rolle, soll aber trotzdem zumindest grundsätzlich ansprechend und gut benutzbar sein, da auch dies den

benötigten Funktionsumfang erhöht.
Die Funktionen des Webshops umfassen in erster Linie das Anzeigen und Verwalten von Produkten. Die erste Komponente hat demnach die Produkte und Produktpflege im Mittelpunkt. Des Weiteren müssen sich User anmelden, Produkte ansehen, in den Warenkorb legen und anschließend bestellen können. Dies benötigt zum einen eine Benutzerkomponente mit Authentifizierung und eine Bestellverwaltung. Die Bestellungen sind auf der einen Seite im Profil der User sichtbar und müssen auf der anderen Seite für ggf. nötige Nachbestellungen überwacht werden. Diese fachlichen Elemente bieten eine Reihe von verschiedenen Anforderungen und Möglichkeiten für den Einsatz von Technologien und um verschiedene Herangehensweisen zu testen. In Abb. 3.1 ist ein typischer Ablauf einer fehlerfreien Bestellsequenz im Webshop zu erkennen.

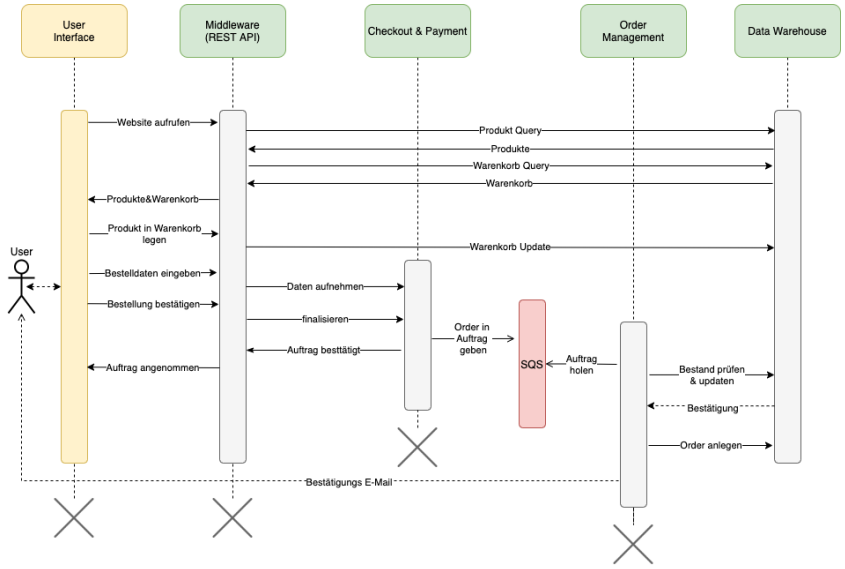


Abbildung 3.1: Bestellsequenz einer fehlerfrei verlaufenden Bestellung

3.2 Entwurf und Architektur

Nachdem die Anforderungen dargestellt wurden, geht es nun darum die geplante Umsetzung zu beschreiben. Aus Forschungszwecken wird in der Anwendungslogik eine reine Umsetzung mit Serverless-Funktionen angestrebt, obwohl in der Realität diverse Mischformen (z.B. mit Microservices) möglich und auch vertreten sind. Dieser Abschnitt dient nun als Entwurf und soll für die einzelnen Komponenten erläutern, welche Technologien

warum und wofür eingesetzt werden. Zuletzt wird die aus dem Entwurf resultierende Referenzarchitektur vorgestellt.

3.2.1 Provider

Aktuell existieren einige Cloud Provider die Serverless Lösungen anbieten. Neben dem bereits erwähnten Amazon AWS sind die größten Provider Google Cloud und Microsoft Azure [Stalcup, 2021]. Diese Provider bieten umfangreiche Services und Unterstützung sowohl für FaaS und BaaS. Natürlich besteht auch noch die Möglichkeit Serverless on-premise zu hosten, oder andere Cloud- oder VM-Anbieter zu nutzen, die keinen so großen Funktionsumfang bieten.
Alle großen Provider bieten eine Reihe von ähnlichen Services und Funktionen an, darunter z.B. Datenbanken, Authentifizierung oder Messaging Systeme. Die angebotenen Services unterscheiden sich zwar in Art und Umfang, aber für den hier gewählten Anwendungsfall wäre jeder Provider ausreichend gewesen, da keine komplexen Technologien wie Maschinelles Lernen o.ä. benötigt werden.
Alle aufgezählten Provider unterstützen die Serverless Architektur und erlauben es Funktionen zu deployen. Damit die Arbeit möglichst repräsentativ ist, wurde der etabliertesten Provider gewählt: AWS. Ein weiteres Argument für Amazon ist das breite kostenlose Kontingent, welches mindestens 12 Monate verfügbar ist, während beispielsweise die GCP nur 3 Monate anbietet [AWS, c][Google]. Diese Aspekte, zusammen mit der sehr ausgereiften Dokumentation, führten zu der Entscheidung.
Zuletzt sei angemerkt, dass die in der Arbeit folgenden Aussagen sich zwar auf eine Umsetzung mit Amazon stützen, aber die Cloud Plattform trotzdem nur ein Werkzeug darstellt. Die einzelnen erforschten Aspekte und die Anwendung an sich können auf den anderen Plattformen ähnlich realisiert werden und haben dadurch eine gewisse Allgemeingültigkeit.

3.2.2 Backend

Das Backend des Shops besteht, wie anfangs erwähnt, komplett aus Serverless-Funktionen, welche bei Amazon als Lambdas bezeichnet werden. Die einzelnen Funktionen sollen orientiert am zweiten Serverlessprinzip sehr feingranular gestaltet werden. Trotzdem sind diese in mehreren logischen Komponenten unterteilt, welche im Folgenden erläutert werden.

Für die Definition und das Deployment des Stacks, also einer Sammlung zusammengehöriger Cloud-Anwendungsressourcen, ist das AWS Cloud Development Kit (CDK) verantwortlich. Bei AWS wird der Stack als CloudFormation-Stack bezeichnet. CDK erstellt diesen, mit Hilfe der Nutzung von bereitgestellten, standardisierten und wiederverwendbaren Konstrukten, die nach Bedarf angepasst werden können und die Anwendung definieren, konfigurieren und letztendlich deployen [AWS, f]. Dadurch wird die Entwicklung möglichst einfach gehalten, da alles über CDK Dateien konfiguriert werden kann, statt dies aufwendig im Einzelnen per Hand oder Skripten zu tun.

Lambda unterstützt nativ die JVM, Go, PowerShell, Node.js, C#, Python und Ruby Code [AWS, l]. Um eine möglichst leichtgewichtige Sprache zu wählen, in der zumindest rudimentäre Erfahrung vorliegt, wurde TypeScript ausgewählt. Ein weiterer positiver Punkt war die Verwendbarkeit von TypeScript im Frontend, wodurch nur eine neue Sprache erlernt werden muss.

Datenbank

Nach der wichtigen Provider-Entscheidung ist die Datenbankwahl naheliegend. Amazons serverloser Dienst für die Datenspeicherung ist die DynamoDB, weshalb der Empfehlung von Amazon gefolgt, und diese in der Referenzarchitektur genutzt wird. Zudem entspricht die Nutzung von verbindungslosen Diensten den Serverless Best Practices.
DynamoDB ist eine Schlüssel-Wert- und Dokumentendatenbank. Amazon beschreibt sie als "[...] eine vollständig verwaltete, multiregionale, multiaktivfähige, dauerhafte Datenbank mit integrierter Sicherheit, Sicherung und Wiederherstellung sowie In-Memory-Caching für Anwendungen im Internetmaßstab. DynamoDB kann mehr als 10 Billionen Anforderungen pro Tag bearbeiten und Spitzen von mehr als 20 Millionen Anforderungen pro Sekunde unterstützen [...]"[AWS, h]. Damit eignet sie sich natürlich sehr gut für Hyperscaler aber auch für jede andere Anwendung im Serverless-Realm, die einen Datenzugriff mit niedriger Latenz benötigt.
Die Integration der Datenbank funktioniert sehr einfach, da es sich um einen AWS eigenen Dienst handelt. Wie oben erwähnt können Tabellen einfach über die CDK Config-Datei definiert werden. Bis auf die Schlüsseldefinition wird bei Dokumentendatenbanken kein konkretes Schema benötigt.

API

Die API für den User und für das Admin-seitige Management basiert auf REST, bzw. HTTP. Wird eine REST API in Kombination mit Lambda-Funktionen eingesetzt, wird normalerweise Amazons API Gateway genutzt. Dieses kann über CDK konfiguriert werden.
API Gateway ist ein von Amazon verwalteter Service, der das Erstellen, Veröffentlichen, Warten, Überwachen und Sichern von APIs unterstützt. Abgerechnet wird für eingehende Aufrufe und ausgehende Datenübertragungen, demnach skaliert auch die API mit der Anwendung und ist so bestens für Serverless geeignet [AWS, b].

Datenschutz

In den Grundlagen wurde hervorgehoben, dass Serverless nicht nur auf FaaS basiert, sondern dass auch BaaS ein Teil der Architektur ist. Das zeigt sich an dieser Stelle bei Authentifizierung und Autorisierung der User über Amazons Cognito Service.
Der Cognito Service kann für Registrierung und Anmeldung sowie Zugriffskontrollen verwendet werden [AWS, g]. Der Service kann mit Hilfe weniger Zeilen Code in die Anwendung integriert werden und dann beispielsweise auch Zugriffe auf das API Gateway beschränken.

3.2.3 Frontend

Das Frontend wird mit Vue.js 3.0 entworfen und ist als Single Page Application (SPA) konzipiert. Im Gegensatz zum traditionellen Ansatz muss der Server nun keine Clientanfragen mehr rendern, da dies vom Client selbst übernommen wird. Dadurch ist die Trennung der Schichten deutlicher und das Backend kann sich voll auf die Logik konzentrieren. Dies spart, vor allem bei der Nutzung von Serverless, Aufrufe und damit Ressourcen, bzw. Geld, da die benötigte Rechenzeit für die Darstellung auf den Client verlagert wird. Zudem entspricht die Entscheidung für eine SPA dem vierten Prinzip der Serverless Architektur.
Das Design spielt, wie eingehend erwähnt, keine Rolle und wird aus diesem Grund und aus mangelnder Erfahrung eher simpel bleiben. In Abb. 3.2 ist eine einfache Skizze zu sehen, die als Orientierung für das Shopdesign dient.

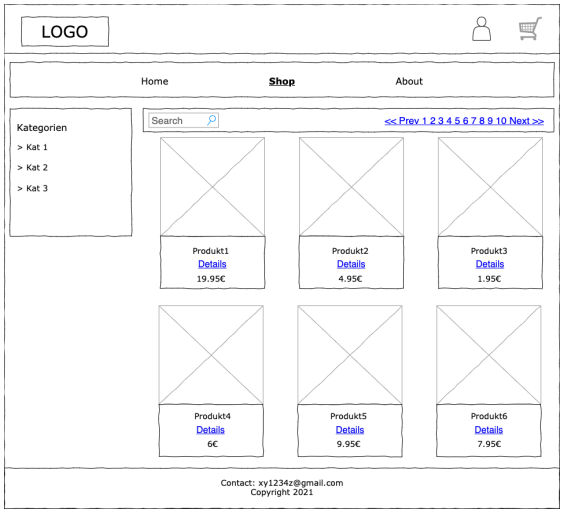


Abbildung 3.2: Grober Wireframe der Shop Seite des Webshops

Hosten des Frontends

Der Simple Storage Service (S3) von Amazon ist ein Objektspeicherservice, der Skalierbarkeit, Datenverfügbarkeit und Sicherheit bietet. Die sogenannten Buckets können für die Speicherung unterschiedlichster Daten genutzt werden und sind im Kostenmodell an den jeweils benötigten Umfang angepasst [AWS, n]. Im hier angestrebten Beispiel wird das HTML-Dokument der entwickelte Single Page Application in einen S3 Bucket hochgeladen und so gehostet.

3.2.4 Referenzarchitektur

Als Ergebnis des Entwurfs ist eine erste Referenzarchitektur für den Webshop entstanden. Die Abbildung 3.3 soll nun einen Überblick über die unter Backend erläuterten Services und Abläufe geben. Genauso wie die obigen Services, orientiert sich die Architektur an dem von AWS bereitgestelltem Stack, sie könnte demnach auch anders gestaltet oder ähnlich bei einem anderen Provider umgesetzt werden. Da die Anwendung relativ schlicht ist und aus einer Reihe von AWS Services gewählt werden konnte, war die Zusammenstellung schnell klar. Grundsätzlich stützt sich die Architektur auf die Prinzipien von Serverless (siehe 2.3.1) und versucht die Serverless Best Practices (siehe 2.3.3) zu beachten.

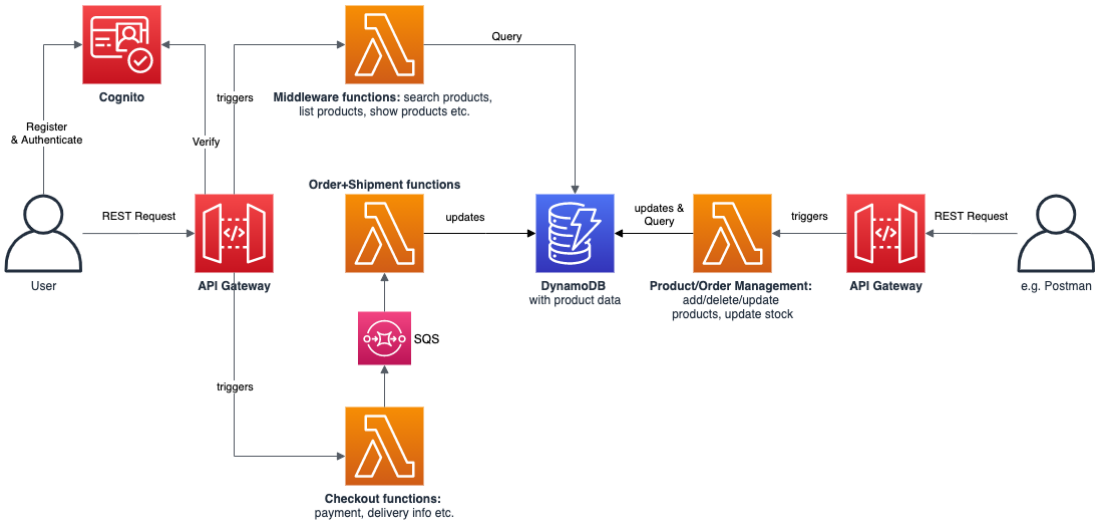


Abbildung 3.3: Angestrebte Referenzarchitektur des Webshops auf AWS

Die linke Seite umfasst die Interaktion mit dem User, welcher sich mit Hilfe von Amazon Cognito anmelden kann und anschließend über eine RESTful API (bzw. das Frontend) mit dem API Gateway kommuniziert. Des Weiteren sind vier Hauptkomponenten zu erkennen, welche die einzelnen Lambda Funktionen enthalten und entweder über das eben genannte API Gateway getriggert werden oder über Messages (Amazons SQS Service) kommunizieren. Ausgehend vom User spielt die als Middleware bezeichnete Komponente eine große Rolle, denn diese ist für die Kommunikation zwischen dem Frontend und der restlichen Anwendung verantwortlich. Die Checkout Komponente verwaltet die Bestellinformationen des Users und kommuniziert asynchron mit der Order/Shipment Komponente. Alle eben beschrieben Komponenten nutzen die Datenbank, um entweder Daten zu bekommen oder einzutragen. Auch die Middleware nutzt direkt die Datenbank, da einfachheitshalber eine Produktsuche wegfällt, welche sonst beispielsweise über Elasticsearch zwischengeschaltet wäre.
Auf der rechten Seite bleibt damit ein Admin Teil, welcher ohne Frontend über eine weitere REST API bedient werden kann. Zuletzt gibt es die Product/Order Management Komponente, welche dazu dient, Produkte anzulegen, Bestände zu kontrollieren und aufzufüllen oder Bestellungen einzusehen. Damit dies möglich ist besteht natürlich auch hier die Verbindung zur Datenbank.

4 The Good - Vorteile

In diesem Kapitel geht es um die guten Seiten von Serverless. Es werden diverse Vorteile und typische Use Cases dargestellt. Es ist dabei natürlich nicht auszuschließen, dass die folgenden Punkte auch negative Aspekte besitzen, jedoch werden diese ausführlich in den nächsten Kapiteln behandelt. Am Ende wird die Umsetzung der Referenzarchitektur evaluiert, der Versuchsaufbau erläutert und eine Übersicht über die Gesamtergebnisse gegeben.

4.1 Vorteile

Serverless ist eine Weiterentwicklung des Cloud Computing, genauer gesagt des Infrastructural Outsourcing und besitzt damit grundsätzlich die fünf, in Abschnitt 2.1.1 unter IaaS, genannten Vorteile. Im Folgenden soll auf diese und einige weitere Vorteile spezifischer eingegangen werden.

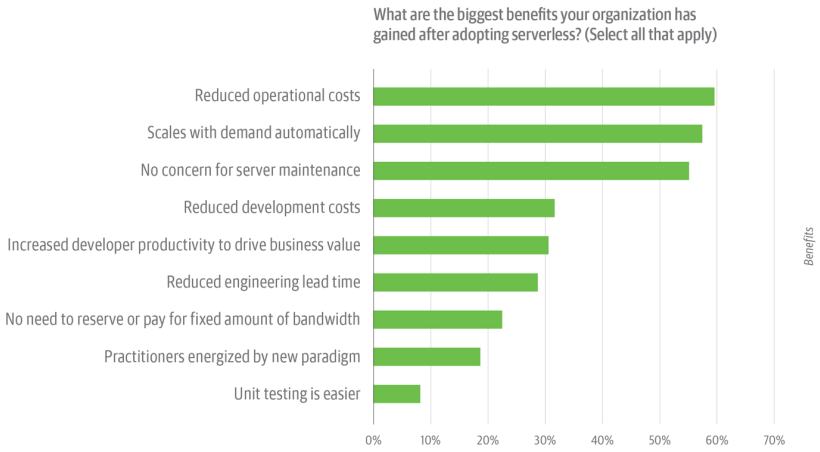


Abbildung 4.1: Serverless Vorteile nach einer O'Reillys Serverless Studie 2019

Abb. 4.1 zeigt einen Teil der Ergebnisse der Serverless Studie von O'Reilly, bei welcher über 1500 Menschen aus verschiedenen Unternehmen und Branchen zu Serverless befragt wurden [Roger Magoulas, 2019]. Dabei wurde deutlich, dass gerade die Kostenersparnis in den verschiedenen Bereichen, sowie die Abgabe von Skalierungs- oder Serverinstandhaltungsaufgaben, die größten Vorteile darstellen.

4.1.1 Kostenersparnis

Einer der größten Faktoren, der für den Einsatz von Serverless spricht ist die Kostenersparnis gegenüber traditionellen Serverlösungen, wie auch anderen Cloudlösungen.
Wie schon bei IaaS und allem was darauf aufbaut, entsteht die Kostenersparnis in erster Linie dadurch, dass keine eigenen Server benötigt werden. Kosten für spezielle Mitarbeiter, Hardware und allgemein Zeit wird eingespart. Da dies bei Serverless grundsätzlich gleichbleibend ist, sollen an dieser Stelle zwei Aspekte genannt werden, die speziell bei Serverless, bzw. FaaS und BaaS, zur weiteren Kosteneinsparung beitragen.

Reduzierte Entwicklungs- und Infrastrukturkosten

Unter IaaS wurde als Vorteil das Wegfallen der Verwaltung eigener Server genannt. Dies war bereits ein großer Schritt, den Serverless weiterführt. Gerade durch die Nutzung von BaaS Produkten wie Datenbanken, Messaging- oder Authentifikationsdiensten fällt ein weiterer großer Teil an Verantwortung weg. Insgesamt muss weniger Logik intern entwickelt werden und die Arbeitskraft kann sich auf businessspezifische Bereiche konzentrieren.
Auch FaaS trägt zu reduzierten Betriebskosten bei, da die Infrastruktur bei der Plattform liegt und somit die Softwareentwicklung, wie auch das Deployment vereinfacht werden. (Vgl. [Roberts and Chapin, 2017])

Skalierbarkeit

Durch die initiale Anschaffung von eigenen Servern wird häufig eine große, eventuell gar- oder noch nicht nötige Investitionen getätigt. Wie auch schon in Bezug auf die Entwicklungs- und Infrastrukturkosten entstand also bereits bei IaaS ein klarer Vorteil. Aus den Werken von Mike Roberts geht hervor, dass das IaaS Modell eine Planung,

Verteilung und schließlich Bereitstellung (eng. Provision) der allokierten Ressourcen benötigt. Damit bleibt der Nachteil des sog. over-provisioning bestehen. Das bedeutet, dass immer die Kapazität der höchsten erwartbaren Last bereitgestellt wird, um komplette Funktionalität zu garantieren, obwohl diese die meiste Zeit nicht benötigt wird. Während andere Cloudlösungen diesem Problem natürlich auch Abhilfe schaffen, kann Serverless dies mit seiner Präzisität am besten.
Eine vorhergehende Planung, Verteilung und Bereitstellung werden nicht mehr benötigt. Stattdessen wird genau die Menge an Kapazität bereitgestellt, welche die Services gerade benötigen. Dies zahlt sich insbesondere in Ruhezeiten aus, in denen sonst für gar keine Last der Preis der Höchstlast bezahlt werden würde, wobei Serverless auf null herunter skalieren kann.
Der Vorteil flexibler Skalierung wurde also bereits deutlich. Serverless benötigt dafür allerdings viel weniger bzw. keinen Aufwand. Die Funktionsinstanzen skalieren automatisch von null bis zu einem Limit von nebenläufigen Instanzen (Limit zur DDoS Vermeidung). Die Skalierbarkeit von sowohl FaaS und BaaS ist damit vor allem für Anwendungen mit inkonsistenter Auslastung kosteneffizienter. Ein letzter, doppelter Vorteil, welcher sich durch FaaS ergibt, ist dass durch eine Verbesserung der Performance der Anwendung gleichzeitig die Betriebskosten sinken, da präzise nach Bedarf skaliert und abgerechnet wird.
(Vgl. [Roberts and Chapin, 2017] und [Roberts, 2018])

4.1.2 Simplifizierung

Unter Simplifizierung sollen die Aspekte gesammelt werden, bei denen Serverless die Arbeit erleichtert und vereinfacht. Trivial zusammengefasst lässt sich feststellen, dass durch die Abgabe von Betriebsaufgaben und der Integration von BaaS Komponenten benötigtes Wissen und Arbeitskraft verringert werden.

Usability

Serverless verändert die Entwicklung und Benutzung von Technologien. Schon in den Grundlagen wurde deutlich, dass der Fokus nun mehr auf der eigentlichen Applikationsentwicklung liegt, statt auf der Infrastruktur und dem Betrieb. Da dies für die meisten Anwendungsentwickler:innen mehr dem gewünschten Tätigkeitsbereich entspricht, kann hier von einer größeren Usability gesprochen werden [Coppel, 2019].

Zudem sind der Einstieg und die komplette Entwicklung sehr leichtgewichtig und schnell. Roberts und Chapin berichten davon, dass selbst unerfahrene Entwickler:innen komplexe Anwendungen mit Serverless schreiben können, die zuvor ohne Hilfe nicht zu erwarten gewesen wären [Roberts and Chapin, 2017]. Die Ergebnisse der O'Reilly Studie legen ebenfalls nahe, dass erhöhte Produktivität und neue Motivation durch die Nutzung von Serverless entstehen kann, wie aus Abb. 4.1 abzulesen ist.
Die in den Grundlagen erläuterte Flexibilität von Serverless, in Bezug auf den Schnitt von Funktionen, die Anbieterwahl (zumindest initial, siehe Vendor Lock-in) und unterstützten Sprachen, bietet insgesamt ein hohes Level an Usability für die Entwickler:innen. Dies kann letztendlich neben größerer Motivation und ggf. Spaß an der Arbeit auch zu produktiveren, effizienteren Teams führen.
Trotzdem muss an dieser Stelle erwähnt werden, dass Serverless nicht "No Ops" bedeutet. Es gibt immer Operations die verwaltet werden müssen. Support, Monitoring, Sicherheit, das Deployment der Software und viele mehr bleiben erhalten. Serverless kann einige Aufgaben auslagern oder andere vereinfachen, aber es kommen auch neue dazu [Roberts and Chapin, 2017].

Auf der anderen Seite kann die Usability des Produktes an sich durch Serverless profitieren. User Experience (UX) und das User Interface (UI) sind heutzutage unverzichtbar und ein wichtiger Faktor für kommerziellen Erfolg [Thinkwik, 2018]. Trotzdem werden UX und UI noch häufig vernachlässigt. Der Einsatz von Serverless kann dem Unternehmen helfen mehr Geld in diese Bereiche zu investieren und Entwickler:innen mehr Zeit dafür einzuräumen [Fee, 2020].

Reduzierte Vorlaufzeit

Eine reduzierte Vorlaufzeit, also die Zeit von der Idee bis zum ersten umgesetzten Code, ermöglicht kürzere Entwicklungszyklen und Innovation. Während der Effekt schon beim Infrastructural Outsourcing entstand, wird auch dieser durch Serverless verstärkt. Es wird berichtet, dass Teams schneller als je zuvor hochverfügbare, stabile Anwendungen bauen können, sogar ohne sehr erfahren zu sein [Roberts and Chapin, 2017]. Während die Verkürzung der Entwicklungszyklen schon durch viele andere Technologien unterstützt wurde (wie CI/CD, Docker etc.), ist das Verkürzen der Vorlaufzeit eine entscheidende Neuerung. Sie erlaubt ein Level an Innovation und Experimentierfreudigkeit, welches es so vorher nicht gab. Nach Mike Roberts kann die Möglichkeit der "continuous experimentation" die Art wie Unternehmen Software ausliefern revolutionieren [Roberts,

2018]. Auch die O'Reilly Serverless Studie zeigt, dass vor allem Unternehmen die Serverless schon länger als drei Jahre nutzen, die verringerte Vorlaufzeit sehr zu schätzen wissen [Roger Magoulas, 2019].

Verringertes Risiko

Das verringerte Risiko in Bezug auf Hardware erweitert sich nun ebenfalls durch die ausgiebige Nutzung von BaaS. Risiken beziehen sich hier vor allem auf Fehler und Ausfälle. Diese werden nun von Expertenteams statt von den eigenen behoben. Dies zahlt sich vor allem aus, weil die auslagerbaren Services häufig nicht die sind, in denen die größte Expertise im Team vorhanden ist. Somit können Ausfälle und Risiken in den meisten Fällen von speziellen Provider-Teams viel besser beobachtet und behandelt werden. Im Umkehrschluss sorgt dies auch dafür, dass insgesamt weniger Risiken, und wenn eher in bekannteren Systemen, übrig bleiben [Roberts and Chapin, 2017].

4.1.3 Nachhaltigkeit

Ein positiver, anhaltender Trend in der heutigen Gesellschaft ist es, nachhaltigere, grünere Lösungen zu finden. Dieser Trend gewann in den letzten Jahren auch in der Informatik an Relevanz, wie zum Beispiel der neue Blaue Engel für Softwareprodukte zeigt.
Gerade die großen Datenzentren verbrauchen viele Ressourcen. Nach The Shift Project's Lean ICT Report ist der globale CO_2 Fußabdruck der Informations- und Kommunikationstechnik (ICT) seit 2013 von 2,5% auf 3,7% gestiegen. Davon kommen 19% aus Datenzentren [Burnicki, 2020]. Trotzdem liegt die durchschnittliche CPU-Nutzung vieler Server weit unter 100%. Diese Unterbenutzung ist bei selbst gehosteten Servern natürlich ein größeres Problem als bei den modernen Cloudlösungen. Nach einem einige Jahre alten Report von NRDC liegt die Serverauslastung on-premise bei lediglich 15%, während ein typischer, großer Cloudprovider ca. 65% erzielt [Burnicki, 2020]. Serverless ist im Vergleich zu traditionellen Cloudanwendungen, wie oben erläutert, präziser und kann damit potenziell noch stärker gegen die Ressourcenverschwendung vorgehen.
Ob cloudbetriebene Anwendungen wirklich weniger Emissionen produzieren, hängt natürlich von den Stromquellen des jeweiligen Anbieters ab. Die drei großen Provider, Amazon, Google und Microsoft, arbeiten alle mehr oder weniger erfolgreich daran, eine nachhaltige Lösung zu schaffen. Insbesondere Google setzt bereits heute auf 100% erneuerbare Energien [Burnicki, 2020].

Während Serverless nicht einmal annähernd eine Lösung für dieses internationale Problem darstellen kann, ist es ein Schritt in die richtige Richtung. Die Nachhaltigkeit zum Thema zu machen, erhöht zudem den Druck auf die großen Provider, grünere Lösungen zu schaffen.

4.2 Typische Use Cases

Dieser Abschnitt soll typische Einsatzzwecke aufzeigen, in denen Serverless heute sinnvoll eingesetzt wird. Natürlich sind dies nur einige Beispiele auf einem großen Spektrum, da Serverless eine sehr flexible Architektur ist.
Grundsätzlich und auch kostentechnisch eignen sich Anwendungen mit stoßweiser Belastung besonders gut. Grund dafür ist die hohe Skalierungsfähigkeit, welche auch auf null herunter skalieren kann. Zusätzlich werden diese Aufgaben vom Provider übernommen, was den Einsatz noch vereinfacht. [Castro et al., 2019]

Durch die Zustandslosigkeit der Funktionen ähnelt Serverless funktional reaktiven Programmierparadigmen. Daher eignet es sich gut für event-basierte Muster [Castro et al., 2019]. Beispiele dafür sind das Bereitstellen von Event Streams, das Verarbeiten von Bildern oder allgemeine Eventverarbeitung von beispielsweise SaaS Anwendungen wie OAuth oder Github [Framework, b]. Gerade Netflix ist bekannt dafür Serverless für die Videoverarbeitung einzusetzen. Des weiteren nutzen sie Serverless für ihr Backupsystem, als Sicherheitsmechanismus für ausfallende Instanzen, wie auch für event-basiertes Monitoring [Retter, 2020].

Der letzte typische Use Case bezieht sich auf die Nutzung und Erstellung von APIs. Zum einen können sog. composite APIs, also die Nutzung von mehreren externen APIs, die durch Serverless in einer Sequenz aufgerufen werden sinnvoll umgesetzt werden. Dadurch kann entweder der Datenfluss zwischen zwei Services kontrolliert, oder der clientseitige Code vereinfacht werden. Zum anderen, als der wahrscheinlich verbreitetster Einsatzzweck, kann Serverless sehr gut für autoscaling APIs und Websites genutzt werden [Framework, b][Castro et al., 2019]. Letzteres entsprich auch dem, was im praktischen Teil dieser Arbeit umgesetzt wurde und im Folgenden behandelt wird.

4.3 Praktische Umsetzung I

Dieses Unterkapitel, wie auch die gleichnamigen in den folgenden Kapiteln, soll die Umsetzung der entwickelten Anwendung in Bezug auf das jeweilige Kapitelthema zeigen. Dieser Abschnitt beschäftigt sich zusätzlich mit der Umsetzung der in Kapitel 3 konzipierten Referenzarchitektur. Es werden die Umsetzungsergebnisse, durchgeführte Tests und ein Gesamtüberblick dargestellt.

4.3.1 Umsetzung der Referenzarchitektur

An dieser Stelle soll das Ergebnis des Fallbeispiels vorgestellt und einige notwenige Änderungen an der ursprünglichen Referenzarchitektur erläutert werden. In Abbildung 4.2 ist die letztendlich resultierte Architektur zu sehen. Im Vergleich zu der unter 3.2.4 abgebildeten Architektur, fällt zum einen AWS Cognito weg und der Messaging-Dienst Simple Queue Service (SQS) wurde durch den Simple Notification Service (SNS) ersetzt. Cognito wurde ursprünglich erfolgreich implementiert, allerdings fordert es, dass das Registrieren und Anmelden mit einer echten, validierten E-Mail-Adresse erfolgt. Es wurde schnell deutlich, dass dadurch die geplanten Load Tests mit vielen verschiedenen virtuellen Nutzern unnötig erschwert werden würden. Zudem trägt die Authentisierung nichts zu den Versuchen bei, sondern nur zur Vollständigkeit der Anwendung.

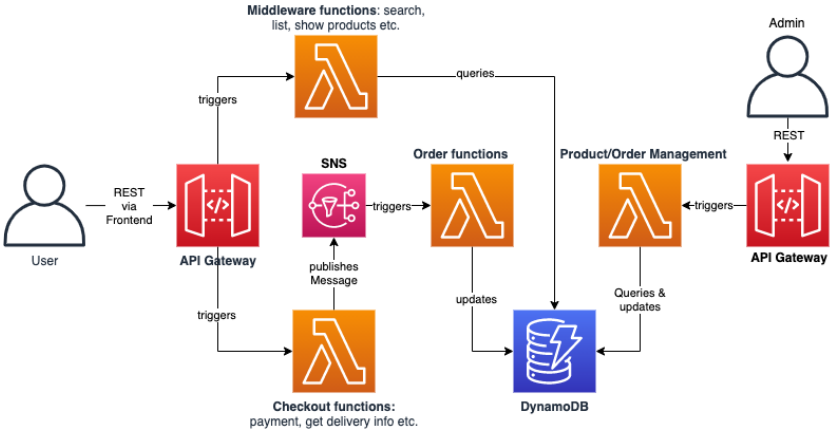


Abbildung 4.2: Umgesetzte Referenzarchitektur des Webshops ohne Cognito

SQS wurde ursprünglich gewählt, da dieser Dienst eine sicherere, verlustlose Art der Kommunikation zwischen den Services bietet. Allerdings prüft der Eventhandler von SQS über active polling, ob neue Nachrichten bereitstehen. Da diese Funktionalität dauerhaft Ressourcen verbraucht und das kostenlose Kontingent dafür nicht ausreicht, fand der Wechsel zu SNS statt. Dieser Dienst basiert auf dem Abonnieren von Eventstreams und ist damit nicht zwangsläufig verlustlos. Im aktuellen Beispiel stellt dies aber kein Problem dar, da das Messaging lediglich für das Anlegen einer Bestellung genutzt wird, welches nicht hochfrequentiert ist.

Frontend

Das Frontend ist ebenfalls live und kann öffentlich abgerufen werden [1]. Abbildungen 4.3, 4.4 und 4.5 zeigen Screenshots mit Auszügen der Anwendung. Diese dienen hier nur der Visualisierung und Vollständigkeit und sind im Folgenden nicht weiter relevant. Bei der Entwicklung war das Frontend für manuelle Tests allerdings sehr hilfreich.

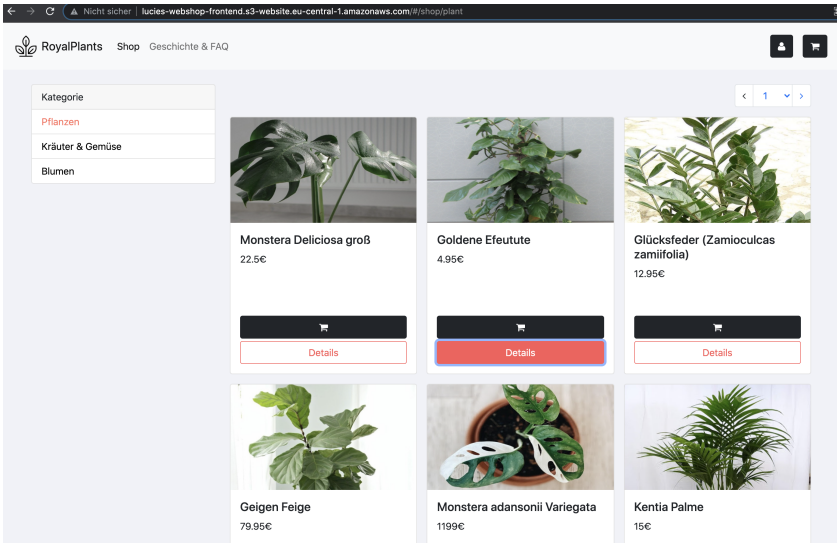


Abbildung 4.3: Eine Seite des Shops

[1] Aktuelle Informationen und Links können dem der Arbeit zugrundeliegenden Repository entnommen werden

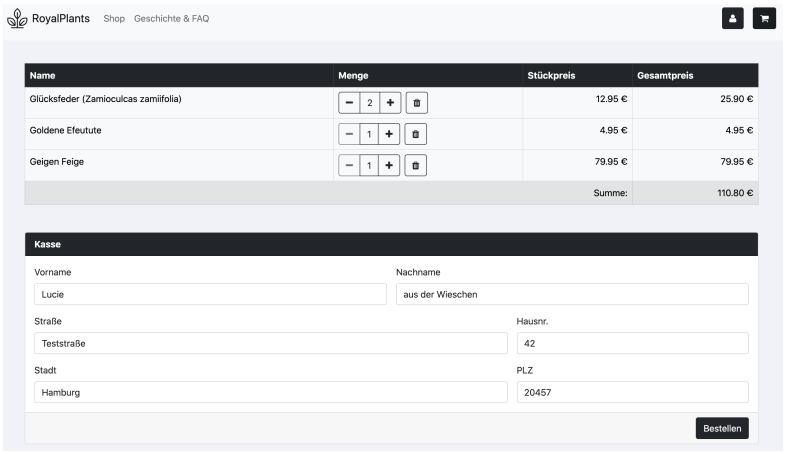


Abbildung 4.4: Gefüllter Warenkorb mit ausgeklappter Kasse

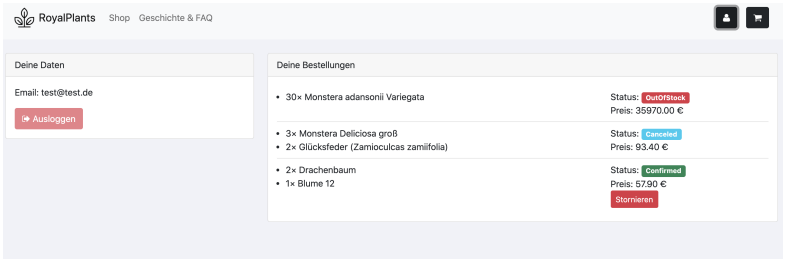


Abbildung 4.5: Vergangene Bestellungen

4.3.2 Versuchsvorbereitung - Monitoring & Tests

Für das benötigte Monitoring konnten die von Amazon bereitgestellten CloudWatch Logs und Metriken genutzt werden. Diese wurden von Grafana importiert, um eine bessere visuelle Übersicht zu ermöglichen. Ähnliches ist auch in CloudWatch selbst möglich, allerdings sind die Einstellungsmöglichkeiten sehr beschränkt.
Für die Erzeugung von Daten war das Durchführen von Lasttests nötig. Die Wahl fiel dabei auf Artillery, da dort Tests mit uneingeschränkt vielen virtuellen Nutzern kostenlos durchgeführt werden können [Artillery].

Testvorbereitung

Für die Durchführung der Lasttests war es wichtig, dass alle entwickelten Stacks die gleichen Voraussetzungen haben und denselben Testablauf durchlaufen. Dafür wurde bei allen getesteten Lambdas ein Timeout nach 30 Sekunden und ein Speicherlimit von 256

MB gesetzt. Alle anderen genutzten Dienste orientieren sich am kostenlosen Kontingent. Die genauen Instanzen und Einstellungen werden bei der Vorstellung der jeweiligen Stacks erläutert. Da die meisten Stacks Amazons DynamoDB nutzen, wird diese bereits hier näher erläutert.
Alle DynamoDB-Stacks greifen auf dieselben Tabellen zu, wodurch sich absolut gleiche Bedingungen ergeben. Die DynamoDB hat hat zwei Modi, den on-demand Modus, der komplett frei und automatisch zur benötigten Kapazität skaliert und einen Modus mit bereitgestellter Kapazität. Im kostenlosen Kontingent für die Region eu-central-1 (Frankfurt) werden 25 GB Speicher, 25 Lese- und Schreibzugriff-Kapazitäten bereitgestellt, welche sich frei auf alle Tabellen verteilen lassen. Der on-demand Modus ist daher nicht kostenlos nutzbar. Eine Lesekapazitätseinheit (LK) ist ein stark konsistenter Lesezugriff, oder zwei eventuell konsistente Zugriffe pro Sekunde. Eine Schreibkapazitätseinheit (SK) stellt einen Schreibzugriff pro Sekunde dar [AWS, i].
Lese- und Schreibkapazitäten wurden manuell auf die Tabellen verteilt, um die bestmögliche Performance zu ermöglichen. Die Aufteilung lässt sich auf den Entwurf der Tests zurückführen, welcher im Folgenden erläutert wird.

Testerstellung

Bei der Erstellung des Tests lag der Fokus darauf, eine realistische Balance zwischen Produktsuche, Warenkorbnutzung und Bestellungen darzustellen. Die Produktsuche, und damit die Lesezugriffe auf die Produkttabelle, werden in der Realität viel häufiger stattfinden als der eigentliche Bestellvorgang. Die Produkttabelle ist also stärker frequentiert als die Lese- oder Schreibzugriffe auf Warenkorb oder Bestellungen (siehe Abb. 4.6).

Tabellenname	LK	SK
Produkte	15	1
Warenkorb	5	17
Bestellungen	5	7

Tabelle 4.1: Einteilung der Lese- und Schreibkapazitätseinheiten

Da die Produkte nur gelesen werden ist die Schreibkapazität auf eine Kapazitätseinheit gesetzt worden, entsprechend wurden die anderen Kapazitäten angepasst. Daraus ergeben sich die Aufteilungen in Tabelle 4.1.
Zudem sollte möglichst jede Funktionalität für einen normalen Bestellvorgang angesprochen werden, um die REST API möglichst ausführlich zu testen. Es wurden lediglich Funktionen zur Erstellung oder Änderung von Produkten ausgelassen, sowie Endpunkte mit ähnlicher Funktionalität, die primär

für die bessere Darstellung im Frontend gedacht waren. Zusätzlich wurde zwischen jedem Aufruf eine Pause gemacht, um ein realistischeres Einzelverhalten zu erzeugen und Probleme auf Grund von kurzen, normalen Latenzen zu vermeiden. Die Lasttests müssen

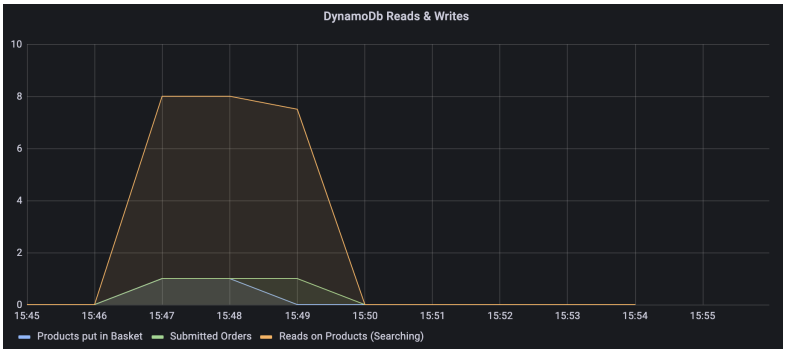


Abbildung 4.6: Verhältnis der Zugriffe auf Produkte (orange), Warenkorb (blau) und Bestellungen (grün)

passend zu dem geringen kostenlosen Kontingent entsprechend klein ausfallen. Trotzdem entsprechen sie einer möglichst starken, realitätsnahen Auslastung für den Höchstbetrieb eines kleinen Webshops.

Die Initialisierung der Tests ist aufgeteilt in drei Phasen, in denen ca. 215-225 virtuelle Nutzer erzeugt werden. In Phase 1 (Warm Up) wird 10 Sekunden lang ein Nutzer erzeugt, Phase 2 (Ramp Up) läuft 20 Sekunden und startet mit einem bis hin zu fünf Nutzern pro Sekunde. Die letzte Phase (Sustained Load) simuliert die höchste Auslastung mit fünf Nutzern pro Sekunde für 30 Sekunden. Die Gesamtzahl je Test schwankt leicht, auf Grund des Verhaltens von Artillery, welches wahrscheinlich auf unterschiedliche Antwortzeiten reagiert.
Jeder virtuelle Nutzer durchläuft den in Abb. 4.7 abgebildeten Testablauf. Abhängig von den Latenzen beträgt die Gesamtzeit der Tests ca. zwei Minuten. Die Nutzer werden alle innerhalb einer Minute gestartet und der Testablauf an sich dauert ebenfalls ca. eine Minute. Daraus ergibt sich die Zahl der parallel aktiven Nutzer, welche nach dem Erzeugen aller Nutzer mindestens bei 150 liegt. Dies lässt sich damit begründen, dass in der dritten Phase 150 Nutzer erzeugt werden, welche den Durchlauf in diesen 30 Sekunden nicht beenden können, hinzu kommen die noch arbeitenden Nutzer aus den ersten beiden Phasen.

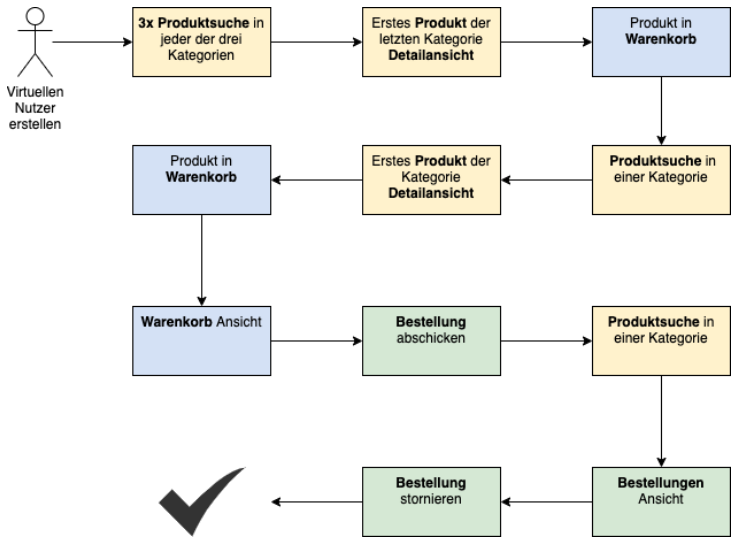


Abbildung 4.7: Ablauf für einen virtuellen Nutzer im Lasttest

Testauswertung

Zur Auswertung der Tests konnte wie oben beschrieben Grafana eingesetzt werden. Leider sind aufgrund der Kürze der Tests, und dem Fakt, dass die Metriken nur im Minutentakt importiert werden, die Grafana Dashboards nicht aussagekräftig genug. Das Problem konnte über die Test-Reports von Artillery gelöst werden. Diese Daten konnten in Excel übertragen und ausgewertet werden. Für jeden Stack wurde der Test drei Mal zu möglichst verschiedenen Zeiten ausgeführt, um ein gewisses Mittel zu haben, aber nicht die gegebenen Kapazitäten zu sehr auszuschöpfen. Zudem wurde darauf geachtet, dass alle Tests mit kalten Lambdas gestartet wurden, um eine ungleiche Voraussetzung durch die Umgehung des Cold Starts auszuschließen.

4.3.3 Ergebnisse der Referenzarchitektur

Die Durchführung der Tests an der Referenzarchitketur hat die in Abb. 4.8 und den Tabellen 4.2 und 4.8 dargestellten Ergebnisse erzielt. Wie oben erläutert, nutzt die Architektur die DynamoDB und ist mit der Sprache Typescript umgesetzt worden, weshalb sie im weiteren Verlauf und in den Abbildungen als DynamoTs referenziert wird.

Stack	Completed	Failed	Response/sec
DynamoTs	219,7	0	29,7

Tabelle 4.2: Grundsätzliche Szenariowerte für DynamoTs

Stack	Minimum	Maximum	Median
DynamoTs	16,7 ms	2254 ms	78 ms

Tabelle 4.3: Min., Max. und Median der Antwortzeiten

Tabelle 4.2 zeigt grundsätzliche Werte des Testverlaufs. Bei DynamoTs wurden in jedem Durchlauf alle Testszenarios ohne Fehler abgeschlossen. Die durchschnittliche Antwortrate liegt bei knapp 30 pro Sekunde.

Abbildung 4.8 zeigt das 95er und 99er Perzentil für die Latenzen der Anfragen. Das bedeutet, dass in diesem Fall 95% der Anfragen eine garantierte Antwortzeit von höchstens 240 ms haben. Für 99% liegt der Wert bei 1280 ms. Am besten sind diese Werte, wenn beide niedrig und nah beisammen sind. 240 ms für 95% der Anfragen ist ziemlich schnell und würde in keiner unangenehmen Wartezeit resultieren. Der zweite Wert, welcher mit über einer Sekunde Latenz schon eher negative Auswirkung haben kann, kann zwei Gründe haben. Zum einen gibt es auch bei TypeScript eine Umgebung die gestartet werden muss und damit zu kurzen Cold Starts führen kann. Zum anderen kann eine zu hohe Auslastung der Datenbank vorliegen. Die oben genannte Verteilung der Kapazitäten ist zwar auf die geplante Nutzung optimiert, aber dennoch relativ gering und kann somit bei vielen Anfragen an die Grenzen gebracht werden, was in erhöhten Latenzen resultiert.

Noch bessere Auskünfte über die genauen Antwortzeiten liefert Tabelle 4.3. Hier wird

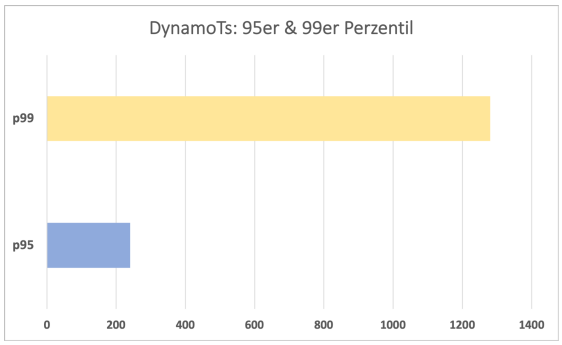


Abbildung 4.8: 95er und 99er Perzentil für DynamoTs

deutlich, dass die maximal gemessene Latenz zwar auch bei über zwei Sekunden liegt, aber der Median bei 78 ms, was sehr schnell ist und damit nochmal insgesamt schnelle

Antwortzeiten suggeriert.
Zusätzlich kann als Ergebnis, bzw. Erkenntnis bemerkt werden, dass die Umsetzung einer serverlosen Anwendung auf Codebasis für mich relativ einfach war. Allerdings reicht das Programmieren der Funktionen an sich nicht aus. Die Konfiguration der Dienste und das Verständnis für diese eröffnete eine andere Art von Komplexität, die wiederum viele Fallhöhen bereithielt. Daher ist die als Vorteil hervorgehobene Simplizität und Usability für unerfahrene Programmier:innen ebenfalls ambivalent. Mehr zu diesem Aspekt wird in Kapitel 7 und in Abschnitt 5.1 elaboriert.

4.3.4 Gesamtüberblick aller Ergebnisse

An dieser Stelle soll eine Übersicht aller Ergebnisse gezeigt werden, damit in diesem und den folgenden Kapiteln die Einzelperformance besser bewertet werden kann. Natürlich werden dort trotzdem immer die Ergebnisse der Referenzarchitektur zum Vergleich herangezogen.

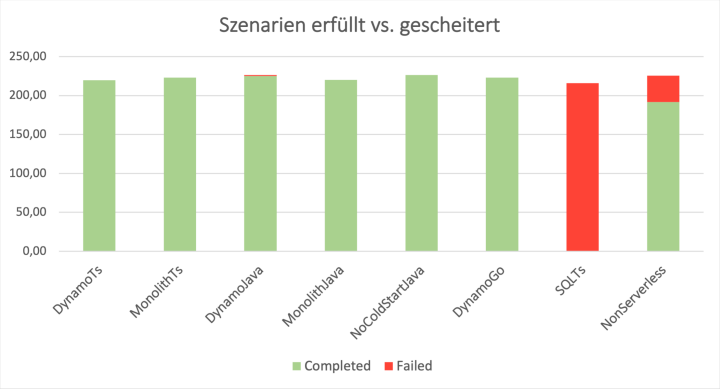


Abbildung 4.9: Szenarien geschafft vs. nicht geschafft für alle Stacks

Allgemein lässt sich feststellen, dass sich die Stacks vor allem in der mittleren Antwortzeit unterscheiden, welche gute Auskunft über die Performance gibt. Die Minimal- und Maximalwerte der Antwortzeiten, aber insbesondere das Minimum, sind häufig ähnlich, da diese zusätzlich von der unterliegenden Plattform bestimmt werden. Die Stacks mit der besten Performance waren die Referenzarchitektur, diese in monolithischer Form und die Umsetzung mit Go. Die Stacks mit der relationalen Datenbank und die Umsetzung ohne Serverless können in den Grafiken teilweise nicht gewertet werden, da sie nicht alle Szenarien abschließen konnten, wie in Abb. 4.9 zu erkennen ist.

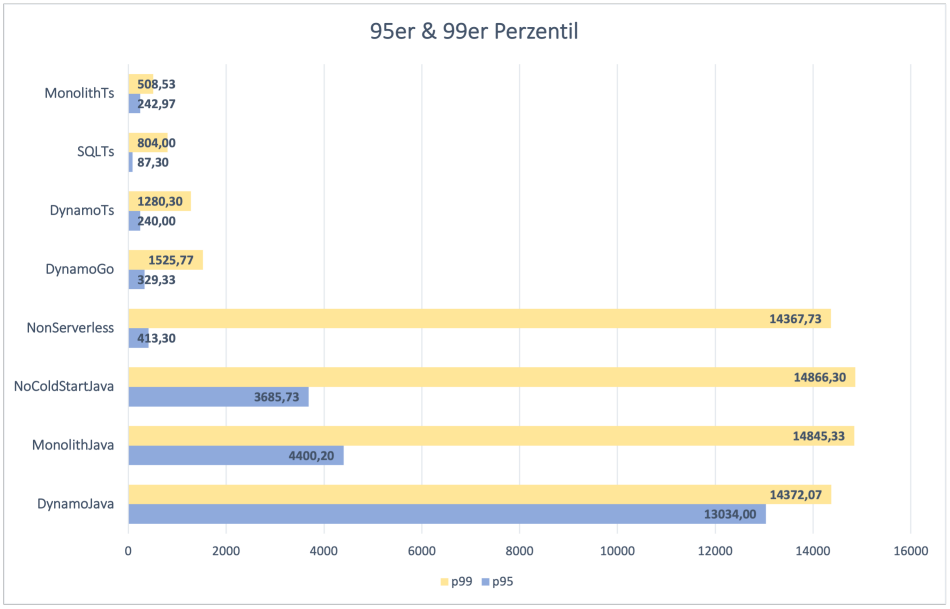


Abbildung 4.10: Alle 95er und 99er Perzentile im Vergleich

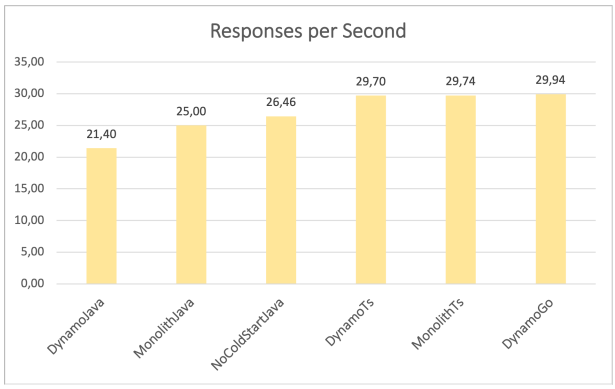


Abbildung 4.11: Antworten pro Sekunde, für alle, die alle Szenarien abgeschlossen haben

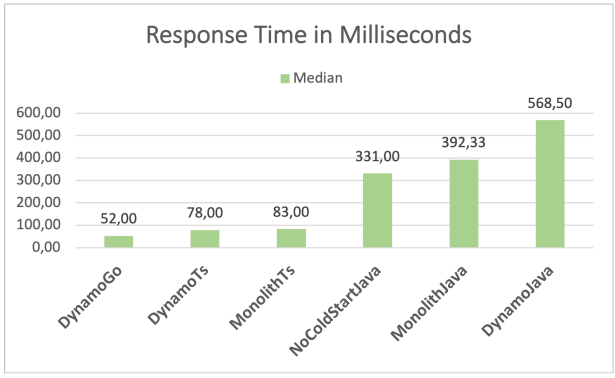


Abbildung 4.12: Mittlere Antwortzeit, für alle, die alle Szenarien abgeschlossen haben

5 The Bad - Herausforderungen, Probleme & Chancen

Im Kapitel "The Bad" werden primär Herausforderungen und Probleme der Serverless Architektur behandelt, aber auch mögliche Chancen und Optimierungen.
Zuerst wird allgemein der Faktor Mensch in Bezug auf die Nutzung neuer Technologien und im speziellen Serverless untersucht. Danach wird auf die inhärenten und die aktuell in der Implementierung wurzelnden Probleme eingegangen. Die Einteilung der Limitierungen orientiert sich grob an dem Buch "What is Serverless" von M. Roberts und J. Chapin. Zuletzt werden beispielhaft zu einigen der behandelten Themen Experimente und Messungen an der in Kapitel 3 konzipierten Anwendung durchgeführt.

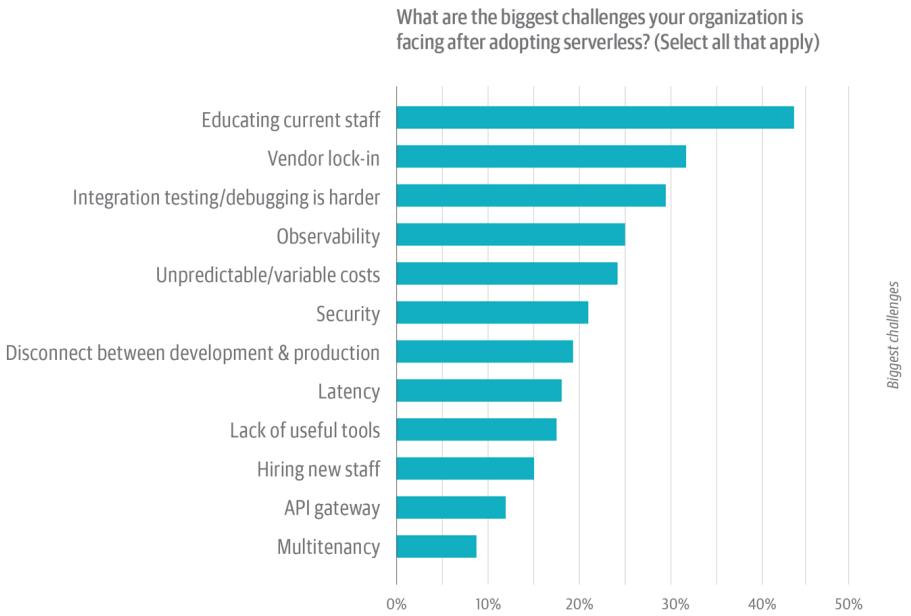


Abbildung 5.1: Herausforderungen nach der Serverless Adaption (O'Reilly Serverless Studie 2019)

Abbildung 5.1 zeigt, wie schon im Kapitel The Good, die Ergebnisse der O'Reilly Serverless Studie, diesmal in Bezug auf die größten Herausforderungen nach der Adaption von Serverless.

5.1 Faktor Mensch

Der erste Abschnitt beschäftigt sich mit den Menschen die Serverless verstehen und einsetzen müssen. Grundsätzlich wird Serverless als gut geeignet für unerfahrene Programmierer:innen beschrieben, wie auch schon unter 4.1.2 näher erläutert. Trotzdem ist die Nutzung neuer Technologien meist eine ambivalente Erfahrung, die auch Hürden birgt.
Die Entstehung von Trends und Hypes ist auch in der Softwareentwicklung normal. Trotzdem können Hypes zu einer Herausforderung für die Adaption neuer Technologien oder Architekturen werden. IT-Blogger Uwe Friedrichsen warnt in vielen seiner Postings vor den negativen Effekten Hypes zu folgen. Vor allem in seiner Serie The Microservices Fallacy erklärt er ausführlich, dass diese Architektur häufig aus den falschen Gründen eingesetzt wird. Beispielsweise, weil es dem momentanen Trend entspricht oder modern wirkt [Friedrichsen, 2021b]. Bei allen neuen Paradigmen die grundlegend die Struktur der Entwicklung verändern, sollte vorab eingehend auf Sinnhaftigkeit geprüft werden. Serverless stellt dabei keine Ausnahme dar. Zudem gilt es zu bedenken, dass es natürlich im Sinne der großen Provider ist, die vermehrte Nutzung ihrer Dienste zu fördern, egal ob wirklich notwendig oder nicht.
Ein weiterer Punkt bezieht sich auf die Anwendbarkeit und das Verständnis unter den Entwickler:innen. Trotz der positiv hervorgehobenen Simplizität kann mangelnde Erfahrung zu schwerwiegenden Fehlern führen. Das Konfigurieren der Plattformen, wie z.B. Amazon AWS, ist keine triviale Aufgabe und zeigt damit erneut, dass NoOps nicht existiert. In einem Artikel, geschrieben von Bennett Garner, wird die Umsetzung einer simplen serverlosen Anwendung auf AWS getestet. Garner beschreibt die benötigte Konfiguration selbst dabei als "schmerzhaft" und weist darauf hin, dass die Vorteile von Serverless direkt an richtige Konfiguration gebunden sind [Garner, 2018].
Ein weiterer Artikel beschreibt, wie ein Startup beinahe an dem Einsatz von Googles Serverless Plattform Firebase scheiterte. Vereinfacht aus dem Grund, dass die Einstiegshürde so gering, und alle benötigten Services erst einmal kostenlos waren [Entreprogrammer, 2021]. In diesem Beispiel, wie auch in vielen anderen dieser Art, tragen nicht unbedingt die Plattformen die Schuld. Vielmehr sind es Fehleinschätzungen die auf Grund des schnellen,

einfachen Einstiegs entstehen und den Eindruck vermitteln, es könnten komplexe Anwendungen ohne Expertise entwickelt werden. Natürlich liegt trotzdem nicht alle Schuld bei den Menschen. Die Plattformen suggerieren die Einfachheit und Flexibilität und bieten kostenlose Kontingente.
Abschließend bestehen auch heute noch grundsätzliche Probleme mit der Cloud. Oftmals wird die benötigte "cloud readiness" in Unternehmen falsch aufgefasst [Friedrichsen, 2021a], es besteht ein Mangel an Weiterbildungen für die Mitarbeiter:innen, oder neue Technologien bzw. Veränderungen werden abgelehnt. Die Serverless Studie zeigt in Abb. 5.1 ebenfalls, dass die Weiterbildung der Mitarbeiter:innen mit nahezu 50% die häufigste Herausforderung der befragten Unternehmen ist. Genauso zeigt Abb. 5.2, dass die Angst vor dem Unbekannten der zweithäufigste Grund ist, warum die befragten Unternehmen Serverless bisher nicht eingesetzt haben.

5.2 Inhärente Limitierungen

In diesem Abschnitt soll es um Limitierungen gehen, die aus den gegebenen Umständen erwachsen. Diese Einschränkungen werden wahrscheinlich nie verschwinden, aber können mit der Zeit besser werden oder sich sogar zu Vorteilen wandeln.

5.2.1 Dezentralisierung

Als erste große Herausforderung kann die Dezentralisierung der Anwendung genannt werden. Das Thema der verteilten Systeme an sich und dessen Problemstellungen, werden hier nicht ausführlich diskutiert, allerdings bleibt zu vermerken, dass verteilte Systeme eine neue Komplexität in die Anwendung bringen.
Sbarski erwähnt in seinem Buch über Serverless ebenfalls, dass die Komplexität des Unterliegenden Systems, durch die Nutzung von kleinen Funktionen anstelle eines Monolithen nicht gleichzeitig weniger werde. Er weist darauf hin, dass verteilte Lösungen eigene, neue Schwierigkeiten mit sich bringen. In erster Linie, weil die Funktionsaufrufe nicht mehr innerhalb desselben Prozesses stattfinden und insgesamt mit Fehlern und Latenzen im Netzwerk umgegangen werden muss [Sbarski, 2017]. Auch die O'Reilly Studie zeigt in der obigen Grafik einige Herausforderungen, die auf verteilte Systeme zurückzuführen sind, wie erschwerte Beobachtbarkeit und Integrationstests, erhöhte Latenzen oder auch Sicherheitsaspekte.

5.2.2 Zustandslosigkeit

Funktionen in einer serverlosen Anwendung sind grundsätzlich Zustandslos. Wird ein Zustand benötigt muss dieser in einer externen Datenbank persistiert werden.
Während Zustandslosigkeit auch Vorteile haben kann, z.B. hinsichtlich der Skalierung, sollen nun die daraus resultierenden Herausforderungen dargestellt werden. Zustandslose Funktionen müssen mit anderen Services oder Funktionen die einen Zustand besitzen kommunizieren, um Daten zu persistieren. Durch Kommunikation entstehen zwangsläufig Latenzen, sowie eine gewisse Erhöhung der Komplexität. Genauso kann Komplexität durch die Nutzung benötigter Dienste an sich entstehen, da diese nicht immer trivial sind. Bei falscher Nutzung können dann stark negative Auswirkungen auf die Performance auftreten. (Vgl. [Roberts and Chapin, 2017])
Roberts weist in einer anderen Arbeit darauf hin, dass für einige Applikationen der fehlende Zustand auf dem Server ein noch größeres Problem darstellen kann. Beispielsweise, wenn ein großer Cache benötigt wird oder schneller Zugriff auf den Zustand einer Session [Roberts, 2018]. Diesen Anforderungen kann Serverless allein, aktuell nicht gerecht werden.

5.2.3 Latenzen

Das Thema Latenzen stellt nach Abb. 5.1 für fast 20% der Unternehmen eine Herausforderung dar. Latenzen können viele Ursachen haben und an vielen Stellen entstehen. Bei Serverless Funktionen handelt es sich grundsätzlich um ein verteiltes System, weshalb Latenzen sowieso immer eine Rolle spielen.
Sie entstehen z.B. durch langsame Kommunikationsmethoden wie HTTP, entweder zwischen einzelnen internen Funktionen oder auch mit BaaS Services nach außen. Häufig ist es möglich Code oder Kommunikationsmechanismen zu optimieren und so das Beste aus den obligatorischen Latenzen rauszuholen [Roberts and Chapin, 2017]. Trotzdem kann dieses Thema schnell zur Herausforderung und in performancekritischen Fällen problematisch werden. Ein weiterer Aspekt im Serverless-Realm sind Latenzen auf Grund von sogenannten Cold Starts, welche in Abschnitt 5.3.3 näher behandelt werden.

5.2.4 Testen

Das lokale Testen von verteilten Funktionen war bereits in Bezug auf Microservices ein großes Thema und bleibt dies auch bei Serverless. Es gibt mittlerweile viele Testkonzepte, welche sich auf Microservices und damit grundsätzlich auch auf Serverless Funktionen anwenden lassen. Eines von diesen ist beispielsweise das Testkonzept von André Schaffer, welcher statt der klassischen Pyramide eine Testwabe vorschlägt. Er ordnet dabei Integrationstests als mit Abstand wichtigste Tests für Microservices ein [Schaffer, 2018].
Da einzelne Funktionen meist wenig Logik haben, ermöglichen sie eine relativ leichte Implementierung von Unittests. Das viel wichtigere Integrationstesten und auch das End-to-End Testen stellen eine größere Herausforderung dar. Dies wird durch die Serverless Studie gestützt, da Integrationstest an dritter Stelle der Herausforderungen stehen.
In Bezug auf lokales Integrationstesten gibt es zwei hauptsächliche Probleme. Zum einen wird ein Großteil der Infrastruktur wegabstrahiert, wodurch das Schreiben realistischer Tests, z.B. in Bezug auf Fehlerbehandlung oder Skalierung, lokal schwierig ist. Das andere Problem liegt in der Natur verteilter Systeme, denn gerade bei Serverless besteht normalerweise eine noch größere Anzahl an einzelnen Funktionen und BaaS-Komponenten als bei Microservices. Eine Möglichkeit dieses Problem zu umgehen ist nach Roberts und Chapin nicht lokal zu testen, sondern dies mit Hilfe von externen Tools zu tun. Diese Tests laufen dann allerdings auf der Plattform selbst, was die Kontrolle über die Testumgebung einschränkt [Roberts and Chapin, 2017].
Auf der anderen Seite wird argumentiert, dass bei klarer Trennung der Funktionen, Integrationstests sogar einfacher sein können als zuvor und der Testrahmen eher dem von Unittests ähnelt. Werden im Sinne der Best Practices keine Funktionen von Funktionen aufgerufen, muss letztendlich nur die Interaktion mit den genutzten Services getestet werden. Die Strategie sollte dann sein, die echten Dienste zu Testen und möglichst wenig Mocks zu benutzen. Dies bringt zwar ggf. Mehrkosten mit sich, ist aber der einzig wirklich sinnvolle weg realistisch zu testen [A-Cloud-Guru, 2017][Shapira, 2019].
Das Testen ist also definitiv in dem Sinne erschwert, dass eine neue Herangehensweise benötigt wird. Daher ist es naheliegend, dass Abb. 5.1 zeigt, dass fast 30% der Unternehmen Integrationstests als Herausforderung ansehen. Grundsätzlich steigt der Schwierigkeitsgrad mit der Komplexität der entworfenen serverlosen Architektur, vor allem wenn Funktionen andere Funktionen aufrufen. Trotzdem entwickeln sich auch in diesem Bereich immer neue und bessere Tools sowie allgemeine Strategien, die nach und nach Abhilfe schaffen können.

5.2.5 Kontrollverlust

In den Grundlagen wurde der Trend der Cloud beschrieben, dass immer mehr abstrahierbare Funktionalitäten aus der Verantwortung der Entwickler:innen genommen werden. Durch das Abgeben dieser Verantwortung lässt sich vor allem eines gewinnen – Fokus auf die Business Logik. Während dies eines der stärksten Argumente für die Nutzung von Serverless ist, bringt dies auch einen Kontrollverlusts mit sich, der herausfordernd sein kann.
Es besteht natürlich die Möglichkeit einen Teil der Kontrolle zurückzugewinnen, indem z.B. der vom Provider empfohlene Tech-Stack angepasst wird. Dies kann aber schnell dazu führen, dass gegen die Prinzipien von Serverless gehandelt wird. Auch Roberts und Chapin argumentieren, dass das Anpassen des Stacks viel Verantwortung und infrastrukturellen Aufwand in Betrieb und Pflege der Dienste bedeuten kann [Roberts and Chapin, 2017]. Während dies in manchen Fällen Sinn ergibt, baut Serverless inhärent darauf auf, die Kontrolle beinahe vollends abzugeben. Dies hat nach den beiden Autoren Auswirkungen auf mehrere Bereiche:
Der erste Bereich beinhaltet die Konfiguration von Services. Dieser Verlust ist relativ offensichtlich, da während der Nutzung einer FaaS Plattform oder BaaS Services lediglich die vom Provider intendierten Konfigurationsmöglichkeiten gegeben sind und diese meist rar ausfallen.
Der zweite Aspekt bezieht sich auf die Performance. Bei der Nutzung von Cloud Computing ist der Zugriff auf die zugrundeliegende Hardware verwehrt. Gerade bei fortgeschrittener Nutzung, liegen viele Schichten der Abstraktion und Virtualisierung dazwischen, weshalb Code nicht extra auf Performance für die jeweilige Hardware getrimmt werden kann.
Ein anderer Performanceaspekt liegt bei den Plattformen selbst. Roberts und Chapin berichten von der Beobachtung, dass gleich konfigurierte Lambdas von der einen Minute auf die andere sehr unterschiedliche Performancewerte haben können, was einzig und allein von den vom Provider eingesetzten Mechanismen abhängt. Gleiches ist damit natürlich auch ein mögliches Verhalten für jegliche BaaS Produkte.
Der letzte Punkt bezieht sich auf die Fehlerbehandlung und Behebung außerhalb des Kontextes der eigenen Anwendung. Die einzige Möglichkeit Plattformfehler oder -störungen zu beheben, ist den Provider zu kontaktieren und zu warten. Es besteht keine Möglichkeit selbst zu handeln oder zeitlich bedingte Maßnahmen zu ergreifen. Gerade Amazon AWS ist nicht bekannt für transparente Fehlermeldungen, auch in ernsteren Problemsituationen wie großflächigen Ausfällen. Dies macht allerdings auch deutlich, dass beispielsweise

Ausfälle von kompletten Datenzentren für die großen Anbieter relativ unkritisch sind, wodurch eventuell etwas Vertrauen bei der Kontrollabgabe zurück gewonnen werden kann. (Vgl. [Roberts and Chapin, 2017])
Der Überbegriff Kontrollverlust zeigt sich auch in den Studienergebnissen in Abb.5.1. Die Punkte Beobachtbarkeit, variable, unvorhersehbare Kosten und die Diskrepanz zwischen Entwicklung und Produktion sind eindeutige Indikatoren, dass der Kontrollverlust noch häufig eine Herausforderung ist. Auch andere Bereiche wie der Vendor Lock-In (siehe 5.3.2), Latenzen und Sicherheit sind davon betroffen. Kontrollverlust steht also immer in Zusammenhang mit Serverless, es besteht jedoch Hoffnung, dass die Herausforderungen mit der Zeit kleiner werden und das Vertrauen wächst.

5.2.6 Sicherheit

Unter den Herausforderungen in Abb. 5.1 ist Sicherheit für gut 20% relevant. Deutlicher wird es in Abb. 5.2, welche zeigt, dass Sicherheit der häufigste Grund ist, warum befragte Unternehmen Serverless bisher nicht adaptiert haben. Daraus ergibt sich, dass Sicherheit auch bei Serverless einen hohen Stellenwert hat und bei der Nutzung definitiv genauer betrachtet werden sollte. Die grundlegende Sicherheitskonfiguration geht bei Serverless

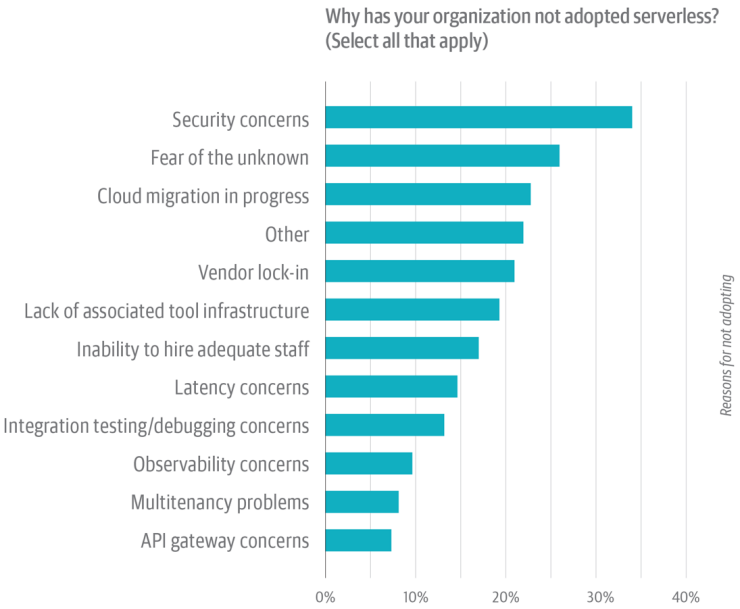


Abbildung 5.2: Gründe, warum Unternehmen Serverless bisher nicht eingetzt haben, zufolge der O'Reilly Serverless Studie 2019

vom Provider aus. Auch hier gibt es nur begrenzte Möglichkeiten die Konfiguration nach den spezifischen Bedürfnissen anzupassen. Die einzige Option besteht meist darin, plattformspezifische Einstellungen vorzunehmen, anstatt auf Systemlevel agieren zu können [Roberts and Chapin, 2017]. Während dies für viele kein Problem darstellt, kann es für sicherheitskritische Anwendungen zum Ausschlusskriterium werden. Abb. 5.3 zeigt wel-

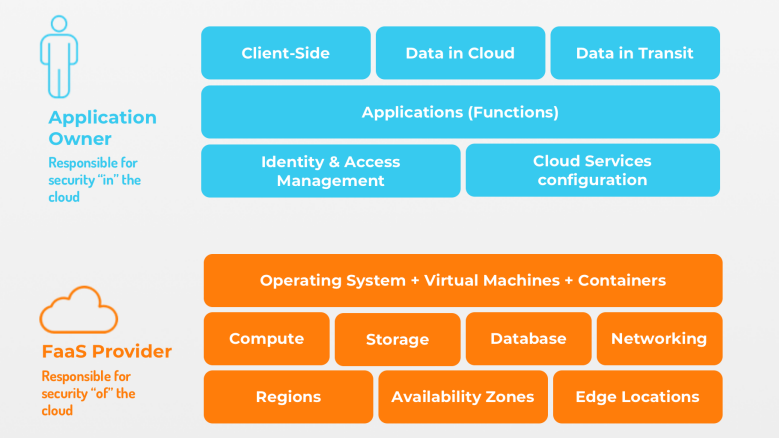


Abbildung 5.3: Verteilung der Sicherheitsverantwortung

che Verantwortungen beim Provider liegen und welche bei den Anwender:innen. Dazu kommt die Nutzung und Konfiguration der BaaS Komponenten.
Sbarski beschreibt in seinem Buch, dass es vorkommen kann, dass die benötigten Sicherheitsansprüche nicht von den Plattformen oder den Diensten der Drittanbieter eingehalten werden können. Das größte Problem dabei ist, dass die Zusicherungen in Form von SLAs (Service Level Agreements) schon bei AWS-internen Diensten, wie auch bei denen von Drittanbietern, stark variieren. Die meisten Anwendungen benötigen nicht mehr als die von AWS angebotene Zuverlässigkeit, allerdings können diese in spezielleren Fällen unzureichend sein [Sbarski, 2017].

Vor- und Nachteile

Auf der anderen Seite stellt sich nun die Frage, ob Serverless die Sicherheit nicht auch verbessern kann. Die Abgabe von komplexen Aufgaben an die Provider hat schließlich den Vorteil, dass nun spezielle Experten die Verantwortung tragen. Check Point, ein global führender Cyber Security Anbieter, führt dazu einige positive Punkte auf:

1. Cloud Anbieter übernehmen das Betriebsystem, Laufzeit-Sicherheit und Sicherheitspatches. Die großen Anbieter sind dabei zuverlässig und bieten größere Sicherheit in diesen Bereichen als durchschnittliche Unternehmen in Eigenverantwortung.

2. Die Herausforderung der Zustandslosigkeit zeigt sich nun auch als Herausforderung für Angreifende. Durch die Kurzlebigkeit und das Fehlen eines Zustands, reduziert sich das Risiko von Langzeitattacken.

3. Die Aufteilung in viele kleine Funktionen kann neben Nachteilen auch einen Vorteil für die Sicherheit bringen. In dieser Form können Tools, die die Anwendungssicherheit unterstützen sollen, die Informationsflüsse der Applikation meist besser überwachen.

4. Provider bieten die Option, für jede einzelne Funktion IAM Rollen zu definieren. Durch die Feingranularität können also detaillierte Sicherheitskonzepte leichter umgesetzt werden, sodass jede Funktion nur genau das kann was sie muss. Dies verkleinert die Angriffsfläche erheblich.

(Vgl. [CheckPoint])

Neben diesen Vorteilen gibt es aber auch sicherheitstechnische Nachteile und Herausforderungen durch Serverless. In einem Dokument, welches von führenden Anwendern der Industrie und Sicherheitsforschern kuratiert und gepflegt wird, wurden die zehn aktuell größten Risiken für Serverless aufgeführt.
Neue Herausforderungen für Serverless betreffen nach ihnen vor allem die Angriffsfläche. Zum einen wird die sie feingranularer und erfordert, dass bei Serverless jede Funktion als Eintrittspunkt betrachtet werden muss. Zum anderen entsteht eine stark vergrößerte, bzw. komplexer werdende Angriffsfläche durch die vielen möglichen Eventquellen der Funktionen [CheckPoint][Segal and Whittaker, 2019]. Insgesamt ist die Systemkomplexität erhöht, was Monitoring und Testing erheblich erschweren kann. Die letzte neue Herausforderung ist das Fehlen von traditionellen Schutzmöglichkeiten. Der Zugriff auf die Server ist verwehrt, weshalb es nicht möglich ist traditionelle Sicherheitsschichten einzubauen, wie z.B. Web Application Firewalls oder Endpoint-Protection [Segal and Whittaker, 2019]. Daraus ergeben sich für sie zehn Risiken, sortiert nach Kritikalität, von welchen die ersten drei hier kurz vorgestellt werden.

SAS-1: Funktionsinjection über Eventdaten: Abstrakt betrachtet treten Injektions-Fehler auf, wenn unsicherer Input direkt an den Interpreter gegeben und ggf. ausgeführt oder evaluiert wird. Bei Serverless geht die Gefahr über Nutzereingaben hinaus und umfasst alle Eventquellen, die die Ausführung einer Funktion anstoßen können. Dies können unter Umständen sehr viele sein.

SAS-2: Fehlerhafte Authentifizierung: Es ist ein komplexes Unterfangen robuste Authentifizierungsmethoden zu entwickeln, welche Zugangskontrollen und Schutz für alle relevanten Funktionen bieten. Es können schnell Fehler gemacht werden, welche die Folge haben können, dass Angreifende, die auf die Anwendungslogik zugreifen, diese manipulieren oder Aktionen durchführen für die sie nicht authorisiert sind.

SAS-3: Unsichere Serverless Konfiguration: Cloudservices bieten viele Konfigurationsmöglichkeiten, um die Plattform so weit wie möglich an die eigenen Bedürfnisse anzupassen. Einige von diesen Konfigurationen haben kritischen Einfluss auf die allgemeine Sicherheit der Anwendung. Die Standardeinstellungen sollten betrachtet werden, da sie nicht immer auf die eigenen Sicherheitsbedürfnisse passen. Gerade falsch konfigurierte cloudbasierte Datenspeicher sorgen für zahlreiche Vorfälle, bei denen sensible, vertrauliche Unternehmensinformationen an unauthorisierte Nutzer gehen, teilweise sogar einfach über eine Suchmaschine. (Vgl. [Segal and Whittaker, 2019])

Weitere Risiken sind dann z.B. überpriviligierte Funktionen oder unsichere Drittanbieterdienste [CheckPoint][Segal and Whittaker, 2019]. Auch die Multitenancy kann ein sicherheitstechnischer Nachteil sein. Roberts zählt dazu Datenschutzprobleme oder dass Daten von anderen Teilnehmern sichtbar sind. Auch Robustheit, negative Auswirkungen auf die Performance oder dass Errors eines Teilnehmers Einfluss auf einen anderen haben, können durch Multitenancy entstehen [Roberts, 2018].
Allgemein besteht das Problem, dass das Etablieren und die Pflege aller benötigten Maßnahmen viel Zeit kosten. Gerade Serverless soll eine schnelle, leichtgewichtige Entwicklung ermöglichen, was nicht gut mit aufwendiger IT-Sicherheit zusammenpasst. Zuletzt bleibt beim Thema Sicherheit sowieso zu bedenken, dass nie alle Risiken bekannt sein können und sich neben den Plattformen auch die Angriffsmöglichkeiten weiterentwickeln. Daher bleibt dieses Thema immer eine Herausforderung die stetige Aufmerksamkeit erfordert und kann den inhärenten Limitierungen zugeordnet werden.

5.3 Limitierungen der Implementation

Als zweite Hälfte und Gegenstück der inhärenten Limitierungen sollen nun die aktuellen Einschränkungen besprochen werden. Serverless ist ein sich schnell entwickelnder Trend. Die Plattformen und auch das Wissen unter den Nutzer:innen erweitern sich stetig. Darum werden sich diese Aspekte voraussichtlich stärker verändern oder sogar komplett wegfallen, während den Inhärenten lediglich Abhilfe geschaffen werden kann. Trotzdem sind dies relevante Punkte, die bei einer Einschätzung der Serverless Architektur zum jetzigen Zeitpunkt nicht fehlen dürfen.

5.3.1 Datenbanken

Aus der Zustandslosigkeit von Serverless ergibt sich eine große Relevanz für Datenbanken. Der klassische Ansatz ist es ein relationales Datenbanksystem (RDBS), wie mySQL zu wählen. Die vier meistgenutzten Datenbanken sind auch heute noch relational [dbR, 2021]. In den Best Practices wird allerdings von verbindungsbasierten Diensten abgeraten, was die Nutzung von den meisten RDBS in Frage stellt.
AWS selbst weist darauf hin, dass RDBS aufgrund der Verbindungen nicht gut Skalieren, und empfiehlt für Serverless, wenn möglich, die DynamoDB [Beswick, 2020]. Trotzdem werden noch häufig RDBS gewählt, was zum einen an den vorhandenen Erfahrungen liegt und zum anderen an der Mächtigkeit der Abfragesprache SQL. AWS wirbt trotzdessen damit, dass auch die DynamoDB mit sog. advanced patterns komplexe Abfragen und Tabellenkonstruktionen ermöglicht [Beswick, 2020]. Allerdings muss mit Mehrkosten gerechnet werden, wenn z.B. globale sekundäre Indizes, die für komplexe Abfragen eingesetzt werden.
Eine möglicherweise bessere Lösung kann es sein, eine Kombination von SQL und NoSQL zu nutzen. Ein Beispiel in AWS wäre es, die DynamoDB für das Aufnehmen von großen Datenmengen aus dem Frontend einzusetzen und für ggf. benötigte ad hoc, SQL-basierte Analysen kann Amazon Aurora genutzt werden [Beswick, 2020].
Falsche Datenbankentscheidungen können aus verschiedenen Gründen zu hohen Kosten führen. Zum einen fehlt häufig die Expertise in den neueren NoSQL Datenbanken. Dies kann zu hohem Aufwand und damit verbundenen Kosten führen, wie es auch das in Abschnitt 5.1 “Faktor Mensch” erwähnte Startup mit Googles Firestore Datenbank erlebt hat. Hohe Kosten können allerdings auch entstehen, wenn entgegen der Empfehlung und

den Best Practices trotzdem eine relationale Datenbank eingesetzt wird. Verbindungslimits können dann sehr schnell erreicht werden und zu Ausfällen führen, oder es werden unnötig große, teure Instanzen angeschafft. Gleiches gilt natürlich auch für NoSQL Datenbanken, die auf Verbindungen angewiesen sind, wie die MongoDB. Diese skaliert zwar automatisch und erfüllt damit grundsätzlich die Serverless Vorraussetzungen, kann aber auf Grund der Verbindungen zu Problemen führen. Ein Unternehmen, dass die MongoDB einsetzte und eine Fehleinschätzung bezüglich der Lambdafunktionalität machte, wie viele Instanzen automatisiert parallel aktiviert werden, musste letztendlich den kompletten Tech-Stack ändern und so einen hohen Preis bezahlen [Titienok, 2020].
NoSQL Datenbanken sind also grundsätzlich die bessere Wahl im Serverless Realm, aber auch dort gibt es Unterschiede und Einschränkungen, welche die Entscheidung erschweren. Zum Beispiel sind NoSQL Datenbanken, wie die DynamoDB, nicht so flexibel, dass sie Veränderung von Datenzugriffsmustern im Entwicklungsverlauf gewährleisten können [DeBrie, 2019].
Es gibt natürlich auch Lösungen, die den Einsatz von RDBS ermöglichen sollen, diese sind allerdings meist komplizierter, teurer oder noch unausgereift. Darunter fallen bei AWS z.B. eine Proxy Lösung die Verbindungspools aufbaut um lastintensive Anwendungen besser händeln zu können. Eines von Amazons neuesten Produkte ist Aurora Serverless, welches vielversprechend wirkt, allerdings noch nicht sehr ausgereift ist. Aurora Serverless bietet automatische Skalierung und auch die Verbindungen werden wegabstrahiert. Allerdings haben langlaufende Transaktionen bisher keine gute Performance gezeigt [Vuollet, 2020].
Es kommt bei Datenbanken ebenfalls auf den Einsatzzweck der serverlosen Funktion an. Wird sie selten benutzt, z.B. für tägliche Reports, sind SQL-Datenbaken problemlos einsetzbar, da die Verbindungen ausreichen. Ist allerdings die ganze Anwendung Serverless und/oder die Funktion erwartet starke Auslastung, ist der ganz klassische Einsatz von RDBS fast unmöglich. Datenbanken sind damit definitiv eine große Herausforderung in der schwere Entscheidungen getroffen werden müssen. Es bedarf guter Kenntnis der Produkte und Möglichkeiten und ggf. eine Umschulung der Entwickler:innen auf NoSQL Technologien. Dieser Umstand wird sich mit der vermehrten Nutzung von NoSQL Datenbanken natürlich verbessern, aber momentan birgt es noch viele Hürden und potenzielle unerwartete Kosten oder sogar Ausfälle.

5.3.2 Vendor Lock-In

Der Vendor Lock-In beschreibt die Abhängigkeit zu dem initial gewählten Provider, wie Google oder AWS. Oftmals ist die Portabilität zwischen den Plattformen sehr schlecht, weshalb von einem "Einschließen" ins Ökosystem eines Providers gesprochen wird.
Bereits in der Erörterung der inhärenten Limitierungen wurde oft auf die Abhängigkeit und die Einschränkungen durch den gewählten Provider hingewiesen. Damit ist dieses Problem grundsätzlich ein inhärentes. Jedoch haben die Entwickler:innen die Möglichkeit bis zu einem gewissen Grad zu entscheiden wie viele provider-spezifische Dienste genutzt werden. Roberts und Chapin beschreiben allerdings, dass die Abhängigkeit häufig gar nicht direkt aus der Nutzung der Services resultiert, sondern aus der guten Integration und Zusammenarbeit dieser. Die Integration der einzelnen Dienste von einem Provider ist meist sehr einfach, sicher und performant [Roberts and Chapin, 2017].
Der Vendor Lock-In war in den Ergebnissen der Serverless Studie, welche oben in Abb. 5.1 visualisiert wurden, ein großes Problem. Über 30%, und damit am zweithäufigsten, wurde der Lock-In als Herausforderung angegeben. Des Weiteren zeigt Abb. 5.2 dass etwas über 20% die Abhängigkeit von einem Provider als Grund sehen, Serverless bisher nicht einzusetzen.
Eine mögliche Lösung, falls diese in Frage kommt, wäre es eine Open Source Plattform zu nutzen, wie Apache OpenWhisk, die nicht direkt an einen Provider gekoppelt ist [Roberts and Chapin, 2017]. Eine weitere Möglichkeit ist es, eine andere Sichtweise auf diese Problematik einzunehmen, wie beispielsweise Uwe Friedrichsen nahelegt.
In seiner Serie "The Cloud Ready Fallacy" geht er genau auf die eben beschriebenen Bedenken ein. Ihm zufolge ist die Herangehensweise vieler Unternehmen, die in die Cloud migrieren möchten, falsch. Nach dieser sollen Anwendungen gebaut werden, die on-premise funktionieren, wie auch für möglichst jede gängige Cloud Plattform. Dadurch bleiben die Unternehmen scheinbar flexibel und unabhängig, sodass sowohl die Migration in die Cloud als auch der eventuelle Providerwechsel in der Theorie einfach sind. Für die Vermeidung des Lock-Ins nehmen Unternehmen extrem große Einschränkungen in der Wahl der Technologien, und meist großen Aufwand in Kauf. Dies erzeugt statt eines Vorteils möglicherweise sogar einen Nachteil in Zeit, Geld und Wettbewerb. Friedrichsen empfiehlt den Lock-In nicht nur hinzunehmen, sondern ihn als wirtschaftlichen Vorteil zu sehen. Natürlich muss der Nachteil dem Vorteil abgewogen, und eine individuelle Entscheidung getroffen werden, aber häufig sind die Angst und die unternommenen Maßnahmen gegen den Lock-In übertrieben. Nach Friedrichsen ist sowieso jede Produktentscheidung eine Art Lock-In und damit unvermeidbar [Friedrichsen, 2021a].

Während Friedrichsens Ausführungen sich allgemein auf die Cloud beziehen, wird in der Serverless Studie deutlich, dass diese auch auf Serverless zutreffen. Eventuell dort sogar verstärkt werden, da noch mehr Verantwortung abgegeben, und potenziell eine noch stärkere Abhängigkeitsbeziehung eingegangen wird. Welche Strategie verfolgt wird und inwiefern der Vendor Lock-In eine Einschränkung darstellen sollte ist also grundsätzlich sehr individuell. Trotzdem ist er aktuell einer der Hauptgründe, Serverless nicht oder nur mit Zweifeln zu nutzen. Da der Lock-In den Providern potenziell viel Geld bringt ist unklar wie die Portabilität zwischen den Plattformen in Zukunft aussehen wird. Bisher hängt das Problem also davon ab, ob sich die Akzeptanz in den Unternehmen verbessert oder verschlimmert.

5.3.3 Cold Start

Als Vertiefung des Abschnitts Latenzen (5.2.3), soll nun das Thema Cold Starts aufgegriffen werden. Cold Starts entstehen durch die Instanziierung der unterliegenden Container und des Codes. Sie sorgen für einen langsameren Start der Ausführung, also größere Latenz. Sie beeinflussen die Latenz also nicht immer, sondern nur wenn die Funktion "kalt" ist und lösen damit Performanceschwankungen aus. Eine Funktion ist kalt, wenn keine aktive Instanz verfügbar ist. Dies ist der Fall, wenn die Funktion neu ist oder aktualisiert wurde, wenn eine weitere Instanz zur Hochskalierung gestartet wird oder wenn die Plattform inaktive Funktionen nach einer gewissen Zeit herunterfährt. Meist bleiben die Funktionen lange genug "warm", allerdings kann bei selten aktivierten Funktionen oder kleineren Anwendungen mit wenigen Aufrufen die Performance nicht konsistent vorhergesagt werden [Roberts and Chapin, 2017].
Wie häufig Cold Starts auftreten ist in erster Linie von der Anwendung abhängig und davon, wie schnell der Provider inaktive Funktionen herunter fährt. Die genauen Werte sind dabei nicht bekannt und variieren, es handelt sich aber in jedem Fall um mehrere Minuten, in denen Funktionen inaktiv bleiben können, ohne deaktiviert zu werden.
Dass Cold Starts auftreten ist also in den meisten Systemen gegeben, aber normalerweise eher selten. Daher ist die Dauer von größerer Relevanz. Diese hängt von vielen Faktoren ab, z.B. der Programmiersprache, wie groß die Codebasis ist, wie die Lambda Funktionsumgebung konfiguriert ist, ob eine Verbindung zu einer Virtual Private Cloud (VPC) benötigt wird und vielem mehr. Die Dauer der Cold Starts kann demnach in vielen Bereichen beeinflusst werden [Roberts, 2018]. Trotzdem sind die Möglichkeiten begrenzt und es ist nicht immer praktikabel die Wahl der Technologien vom Cold Start abhängig

zu machen. Außerdem gibt es Systeme, die eine sehr niedrige Latenz benötigen und gar keine Cold Starts akzeptieren können. Daher gibt es Methoden den Cold Start durch Warmhalten der Funktionen komplett zu vermeiden, von denen die zwei wichtigsten vorgestellt werden.
Die erste Methode benötigt einen extra Endpunkt, der einen regelmäßigen Ping empfängt. Dadurch wird die Funktion aktiv gehalten und erlebt keinen Cold Start. Die Kosten für das Warmhalten sind in diesem Fall lediglich von der Anzahl der warmzuhaltenden Funktionen abhängig und lässt sich berechnen. Werden 100 Lambdas (je 128MB) über einen Monat im Fünfminutentakt warmgehalten, betragen die Extrakosten für das Warmhalten ca. 2,27USD.
Während die Kosten der ersten Methode also überschaubar sind, hat diese noch immer ein Problem: Durch den Ping wird nur eine Instanz der Funktion angesprochen. Ist die Zahl der Anfragen so hoch, dass AWS automatisch mehrere nebenläufige Instanzen startet wird nur ein Cold Start eingespart. Aus diesem Grund hat Amazon Ende 2019 sogenannte Provisioned Concurrency eingeführt. Dabei kann vorab bestimmt werden wie lange eine Funktion mit beliebig vielen Instanzen (bis zum max. Concurrency Level) warmgehalten werden soll, je nach den individuellen Bedürfnissen. Der Preis berechnet sich aus der Anzahl der Lambdas (100), der Speichergröße der Lambdas (128MB) und der gewünschten Zeit. Die erste Methode läuft 24h durchgehend, das wäre bei dieser allerdings extrem teuer und unrealistisch. Bei beispielsweise sechs Stunden am Tag würde das Warmhalten hier 40,62USD pro Monat extra kosten[1].
Da der Preis für diese zweite Methode im Vergleich so hoch ist, ist es wichtig, nur die Lambdas und Anzahl an Instanzen warm zu halten, bei denen es dringend notwendig ist. Eine aktueller Research Report über Serverless hat ergeben, dass von den unternehmensübergreifend untersuchten Funktionen über die Hälfte, weniger als 80% ihrer konfigurierten Provisioned Concurrency nutzen und über 40% alles benutzen, wodurch das Auftreten von Cold Starts wieder wahrscheinlicher wird [Datadog, 2021]. Die Nutzung dieser teuren, neuen Möglichkeit ist dementsprechend nicht einfach und sollte daher nur mit Bedacht eingesetzt werden.

5.3.4 Unausgereiftheit und versteckte Kosten

Serverless entwickelt sich stetig weiter, trotzdem gehen die strengsten Einschränkungen von den Providern und den Ausführungsumgebungen für die Funktionen selbst aus. CPU,

[1] Für die Zahlen zur Berechnung siehe [AWS, m]

Speicher und auch alle anderen Ressourcen sind limitiert [Roberts and Chapin, 2017]. Ein gutes Beispiel dafür ist die Beschränkung der Ausführungszeit und des Speichers, welche in AWS aktuell bei 15 Minuten und 10GB liegt [AWS, 2021a][AWS, a]. Diese Werte haben sich bereits stark weiterentwickelt, beispielsweise standen vor Dezember 2020 nur 3GB Speicher zur Verfügung. Allerdings sind dies trotzdem strenge Limitierungen die zu beachten sind. Auf der anderen Seite suggerieren diese Einschränkungen oftmals eine bestimmte Nutzung und Anwendungsfälle. Prozesse, die beispielsweise länger als 15 min. laufen sind schlicht nicht für diese Architektur gedacht, oder wurden falsch designed.

Eine weitere Herausforderung findet sich im Monitoring und Logging, welches bei verteilten Systemen, vor allem in der Cloud, seit jeher in aktiver Entwicklung steht. Heute stehen dafür bereits viele Tools von Drittanbietern und Funktionen der Provider selbst zur Verfügung, diese sind allerdings teilweise noch unzureichend oder sehr komplex. Auch das Remote-Testen der gesamten Anwendung im Gegensatz zum lokalen Testen ist problematisch. Die Provider bieten meist nur Tests für einzelne Funktionen oder Komponenten an. Für das Testen der gesamten Anwendung sind die Möglichkeiten beschränkt und im Zweifelsfall werden die Produktionsressourcen stark strapaziert, wie auch unter 5.2.4 erläutert wurde. Ein letzter Punkt des Toolings ist das Debugging, welches eine allgemeine Herausforderung verteilter Systeme darstellt. Wird dies benötigt müssen oftmals unausgereifte, externe Tools genutzt werden.
(Vgl. [Roberts and Chapin, 2017])
Unausgereiftheiten der Tools und Plattformen sind aktuelle Herausforderungen, welche sich allerdings sehr schnell wandeln und daher im Detail nicht genauer besprochen werden. Die schnelle Entwicklung bringt selbst allerdings auch den Anspruch mit sich, ständig neue Dinge zu lernen und die eigene Anwendung ggf. anpassen zu müssen.
Zuletzt soll es um die versteckten Kosten von Serverless gehen. Ob die Kosten wirklich versteckt sind oder eher das System undurchsichtig ist, bleibt Interpretationssache. Die Serverless Studie zeigt in Abb. 5.1, dass unvorhersehbare oder versteckte Kosten eine Herausforderung für fast 25% der Unternehmen sind.
Gerade AWS ist dafür bekannt zwar der günstigste Anbieter zu sein, allerdings wird auch häufig eine Intransparenz in den Kosten kritisiert. Die sichtbaren Kosten beziehen sich meist nur auf die Kosten der Anfragen und der Nutzung von CPU und Arbeitsspeicher. Gerade die Kosten der API Gateways und für verschiedene Netzwerke wie VPCs sind eher schlecht zu überblicken. Gleiches gilt auch für die Beschreibung des kostenlosen Kontingents, welche meist diese versteckten Kosten ignoriert [Shachar, 2018].

Serverless kann definitiv günstiger sein als traditionelle Serverlösungen, allerdings ist die Rechnung nicht ganz so einfach wie auf den ersten Blick erwartet. Fehler in der Auswahl der Dienste oder ungenaue Recherche können schnell zu hohen, unerwarteten Kosten führen.

5.4 Praktische Umsetzung II

Dieser Abschnitt beruht auf der in Kapitel 3 und 4 konzipierten und umgesetzten Anwendung. Die Idee war es, sich einigen Herausforderungen selbst zu stellen und zu verstehen was genau die Ursachen sind, wie auch zu prüfen, wie verheerend die Auswirkungen auf Projekte in dieser Größenordnung sind.
Die folgenden Versuche wurden ausgewählt unter Erwägung des Aufwands, der erwarteten Veränderung und den Möglichkeiten des kostenlosen Kontingents von AWS. Sie behandeln jeweils einen Aspekt oder eine Technologie, die in der Referenzarchitektur ausgetauscht wurde. Um die einzelnen Versuche vergleichbar zu machen, wurden die gleichen Tests ausgeführt und Metriken genutzt wie unter Praktische Umsetzung I beschrieben.
Der Austausch der Technologien wurde mit verschiedenen CDK Stacks vorgenommen. Grundsätzliche Gemeinsamkeiten werden aus einem Base-Stack entnommen, um die veränderte Anwendung umzusetzen. Letztendlich enthält der Base-Stack lediglich die DynamoDB Konfiguration, da der Rest spezieller angepasst werden musste. Das bedeutet, dass alle Stacks die die DynamoDB nutzen diese in ihren Stack importieren. Die Stacks an sich werden alle einzeln deployed und angesprochen.

5.4.1 Optimierung mit Go

In der Konzeption der Anwendung wurde erwähnt, dass die Auswahl der Technologien und Art der Umsetzung an den Prinzipien und Best Practices orientiert ist. Trotzdem war es nicht direkt das Ziel eine optimale Anwendung zu entwickeln, sondern eine, die aus Erfahrungstand, Dokumentationen und Tutorials resultieren konnte. Natürlich gibt es viele Möglichkeiten diese zu optimieren und mehr aus der Serverless Architektur herauszuholen. Vor allem Latenzen spielen eine große Rolle, wie in den obigen Herausforderungen deutlich wurde.

Stack	Completed	Failed	Response/sec
DynamoGo	223,3	0	29,94
DynamoTs	219,7	0	29,7

Tabelle 5.1: Grundsätzliche Szenariowerte für DynamoGo

Bei der Auswahl für mögliche Optimierungen musste sich an kostenlose Angebote gehalten werden, weshalb sich für den Austausch der Programmiersprache entschieden wurde. Die einzige andere Möglichkeit mit gleichem Funktionsaufbau die Performance zu verbessern wäre eine Erhöhung der Kapazitäten gewesen, welche natürlich mit Kosten verbunden ist. Ein kostenloser, naheliegender Versuch lag also darin eine auf Effizienz entworfenen Programmiersprache einzusetzen, d.h. hier im Backend Go anstelle von TypeScript zu nutzen. Alle anderen Technologien bleiben gleich, weshalb der Stack im Folgenden als DynamoGo referenziert wird. Tabelle 5.1 zeigt, dass wie auch bei DynamoTs,

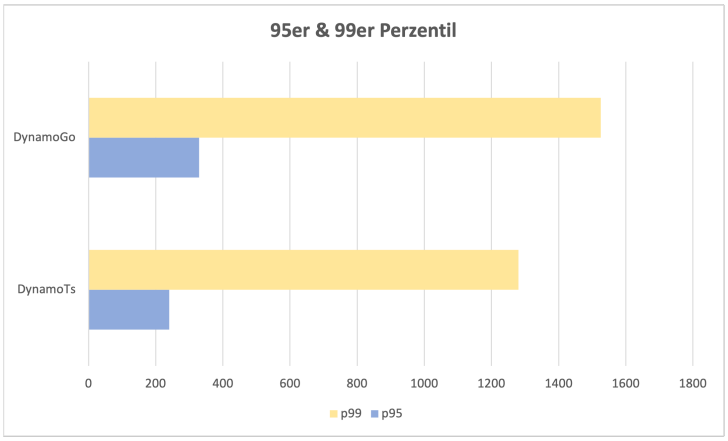


Abbildung 5.4: 95er und 99er Perzentil für DynamoGo im Vergleich zu DynamoTs

alle Szenarios fehlerlos abgeschlossen wurden und die Antwortrate ähnlich nah an 30 pro Sekunde liegt.
Die 95er und 99er Perzentile in Abb. 5.4 sind beide etwas schlechter als bei DynamoTs. 95% können 329 ms zugesichert werden und 99% 1526ms. Die Werte liegen im Vergleich zwar nicht weit auseinander, allerdings entsprechen sie in dieser Hinsicht keiner Optimierung – sondern im Gegenteil – einer Verschlechterung. Dies könnte daran liegen, dass die DynamoDB API für Typescript optimierter ist als die API für Go. Für Typescript gibt es einen DocumentClient, der als Antwort direkt ein JSON Objekt liefert. Go hat diesen Client nicht und bekommt daher lediglich eine Map die in ein Objekt transformiert werden muss. Diese Transformation ist vor allem für das Anzeigen der Produkte

sehr aufwendig, da jedes Objekt der Produktliste transformiert werden muss. Daran ist allerdings nicht nur Go Schuld. In Kapitel 3 wurde erwähnt, dass in meinem Beispiel auf eine richtige Produktsuche verzichtet wurde, um die Komplexität nicht noch weiter zu erhöhen. DynamoDB würde daher in echten Projekten eher nicht für die Suche genutzt werden, sondern ein Datensuchdienst wie Elasticsearch, der die Performance mit Go verbessern könnte.

Stack	Minimum	Maximum	Median
DynamoGo	16,7 ms	216,3 ms	52 ms
DynamoTs	16,7 ms	2254 ms	78 ms

Tabelle 5.2: Min., Max. und Median der Antwortzeiten

Tabelle 5.2 verleiht dem Optimierungsgedanken wieder Hoffnung, da hier ein deutlich niedrigerer Wert als Median herausgekommen ist. Der maximale Wert liegt wieder etwas über zwei Sekunden, aber der Median liegt bei lediglich 52 ms. Dies ist zwar in der Praxis kein großer Unterschied zu 78 ms, aber kann trotzdem als Optimierung der mittleren Antwortzeit gesehen werden. Der Median weist erneut auf die "Cold Start"- Problematik hin, denn Go ist im Gegensatz zu TypeScript eine Kompilierte Sprache, welche grundsätzlich eine längere Hochfahrzeit haben als interpretierte Sprachen. Trotzdem ist der Unterschied hier nicht groß und es ist zu erwarten, dass bei länger laufenden Tests Go eine weitaus bessere Performance zeigen kann.
Die Go Optimierung ist also in diesem Versuch eher schlecht ausgefallen. Es konnte zwar eine leichte Verbesserung erkannt werden, aber keine lohnende Optimierung. Allerdings ist der Entwickelte Code nicht hoch optimiert und zum anderen ist die Anwendung noch sehr simple und der Unterschied zwischen den Programmiersprachen könnte, wie eben erwähnt, erst bei komplexeren Apps deutlich werden.

5.4.2 Neues Funktionsdesign

Ein weiteres Experiment beschäftigt sich mit der Flexibilität von Serverless. Es wurde beschrieben, dass das Funktionsdesign zwar häufig sehr feingranular ist, aber theoretisch alle Schnitte möglich sind. Da die entwickelte Anwendung nicht sehr umfangreich ist, wäre es ohne Serverless naheliegend, zumindest erst einmal, einen Monolithen zu bauen. Im folgenden Versuch, soll also getestet werden, wie sich die Performance verhält, wenn aus den kleinteiligen Funktionen eine monolithische Lambda gemacht wird. Damit die

Umsetzung so leicht wie möglich ist, blieben die Funktionen soweit erhalten und lediglich ein Router wurde hinzugefügt. Genauer betrachtet ist die Anwendung also eher ein Modullith, aber der wichtige Punkt ist, dass nur eine Lambdafunktion existiert. Zusammengefasst gleicht dieser Stack dem DynamoTs Stack, auch die interne Kommunikation per SNS bleibt bestehen. Allerdings wird nur eine Funktion angesprochen, welche die HTTP Anfragen durch einen internen Router verarbeitet und so die passende Antwort zurück liefert. Der Stack trägt daher den Namen MonolithTs. Tabelle 5.3 zeigt, dass

Stack	Completed	Failed	Response/sec
MonolithTs	223,3	0	29,74
DynamoTs	219,7	0	29,7

Tabelle 5.3: Grundsätzliche Szenariowerte für MonolithTs

Stack	Minimum	Maximum	Median
MonolithTs	16,7 ms	2636,3 ms	83 ms
DynamoTs	16,7 ms	2254 ms	78 ms

Tabelle 5.4: Min., Max. und Median der Antwortzeiten

wieder alle Testdurchläufe erfolgreich abgeschlossen wurden und auch die Antwortrate fast exakt mit der von DynamoTs übereinstimmt. Das in den Grundlagen unter Patterns und Antipatterns beschriebene Problem, dass die Funktion über die gegebenen Limits läuft, trat demnach nicht ein.
In Abb. 5.5 und Tabelle 5.4 wird stattdessen deutlich, dass der Monolith sogar eine weitaus bessere Performance besitzt als DynamoTs. Im Gesamtergebnis der Perzentile ist sogar zu erkennen, dass MonolithTs der performanteste Stack von allen ist. Für 99% der Anfragen können 509 ms garantiert werden und für 95% 243 ms. Der Median liegt zwar mit 83 ms etwas höher als der von DynamoTs, aber die sehr guten Perzentile suggerieren trotzdem insgesamt bessere Latenzen.
Dieses durchweg positive Ergebnis kann in verschiedener Weise ausgelegt werden. Zum einen könnte damit argumentiert werden, dass monolithische Anwendungen in dieser Größe performanter und einfacher zu entwickeln sind und es deshalb nicht notwendig ist Serverless zu nutzen. Andererseits wurde von Beginn an kommuniziert, dass Serverless flexibel ist und genau solche Freiheiten im Funktionsschnitt erlaubt. Der große Vorteil in der Entwicklung ist dann, dass insbesondere unerfahrene Entwickler:innen keinen eigenen Server aufsetzen müssen, nicht selbst skalieren und Kosten sparen. Letztendlich ergibt es bei einem solchen Beispiel sicherlich Sinn, erst einmal einen Monolithen zu bauen, um

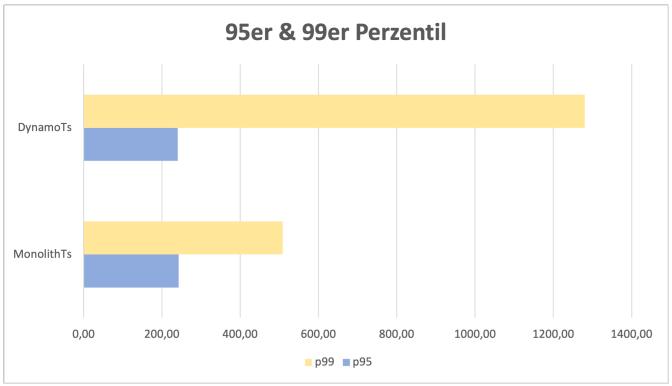


Abbildung 5.5: 95er und 99er Perzentil für MonolithTs im Vergleich zu DynamoTs

damit dann potenziell Serverless einsetzen zu können und die Vorteile maximal auszunutzen. Allerdings ist dabei trotzdem Vorsicht geboten, denn wenn die Anwendung wächst, kann schnell eine harte Grenze erreicht werden, bei der Flexibilität und Skalierbarkeit nicht helfen. Spätestens dann muss eine verteilte Anwendung entwickelt werden.

5.4.3 Alternative Java und ihre Optimierungen

In diesem Abschnitt geht es grundsätzlich darum Typescript mit Java auszutauschen. Java wird von AWS nativ unterstützt und ist häufig eine naheliegende Wahl. Aus diesem Grund wurden mit Java insgesamt drei Versuche durchgeführt, die im Folgenden dargestellt werden.

Umsetzung mit Java

Wie auch schon bei der Optimierung mit Go, wurde lediglich die Programmiersprache im Backend durch Java ersetzt. Daher wird der Stack im Folgenden mit DynamoJava referenziert. Tabelle 5.5 zeigt dass zwar wieder alle Durchläufe erfolgreich abgeschlossen

Stack	Completed	Failed	Response/sec
DynamoJava	225,3	0	21,4
DynamoTs	219,7	0	29,7

Tabelle 5.5: Grundsätzliche Szenariowerte für DynamoJava

wurden, aber erstmals eine Verschlechterung in der Antwortrate zu erkennen ist. DynamoJava schafft 21 Antworten pro Sekunde, fast 10 weniger als die vorherigen Stacks.

Auch die Perzentile in Abb. 5.6 zeigen eine sehr deutliche Verschlechterung. Für 99% liegt die Latenz bei kleiner oder gleich 14372 ms und für 95% bei 13034 ms. Das bedeutet, dass ziemlich viele Anfragen eine Latenz von über 10 Sekunden haben, was für eine Anwendung mit Endnutzerinteraktion nicht wirklich tragbar ist. Dieses Problem liegt allerdings hauptsächlich an den Cold Starts und wäre in einem höher frequentierten Webshop nicht so gravierend. Trotzdem können teilweise schon wenige Sekunden den verbleibt auf einer Website ausmachen [Planet, 2019]. Java läuft auf der JVM und ist dadurch bekannt für Cold Starts, welche eine der größten Herausforderungen sind. Der Median in Tabelle 5.6 zeigt einen dementsprechend ebenfalls einen schlechten Wert von über einer halben Sekunde.

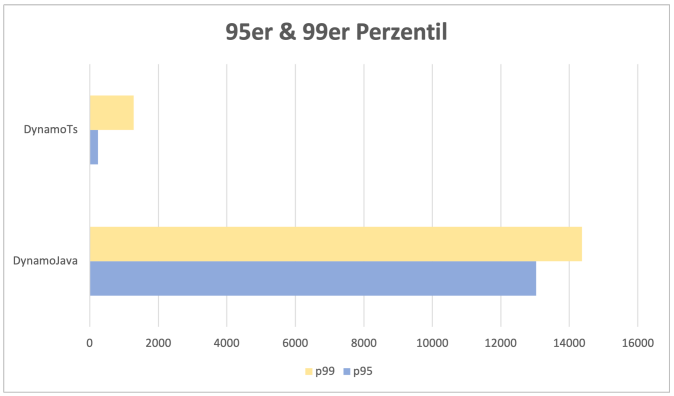


Abbildung 5.6: 95er und 99er Perzentil für DynamoJava im Vergleich zu DynamoTs

Stack	Minimum	Maximum	Median
DynamoJava	16 ms	16134,3 ms	568,5 ms
DynamoTs	16,7 ms	2254 ms	78 ms

Tabelle 5.6: Min., Max. und Median der Antwortzeiten

Daraus ergibt sich, dass Java hier nicht gut geeignet ist. Auch die Entwicklung mit Java war unschön, da sich die DynamoDB API nicht sehr benutzerfreundlich zeigte, eventuell weil Java in diesem Bereich weniger genutzt wird als andere Sprachen. Nach dem initialen Entwicklungsaufwand der Referenzarchitektur war der Java-Stack einer der schwierigsten, obwohl in Java die meiste Erfahrung vorhanden war.

Javaoptimierung – monolithische Funktion

Trotz der schlechten Erfahrung im ersten Anlauf, wurde Java weiter getestet, da es eine sehr beliebte Programmiersprache ist und somit häufig zum Einsatz kommt. Da bei TypeScript das Umschreiben in eine einzelne monolithische Funktion einen so großen Erfolg gezeigt hat, lag es nahe einen Monolithen mit Java umzusetzen. Der Stack bekam daher den Namen MonolithJava.

Stack	Completed	Failed	Response/sec
MonolithJava	220,3	0	25
DynamoJava	225,3	0	21,4

Tabelle 5.7: Grundsätzliche Szenariowerte für MonolithJava

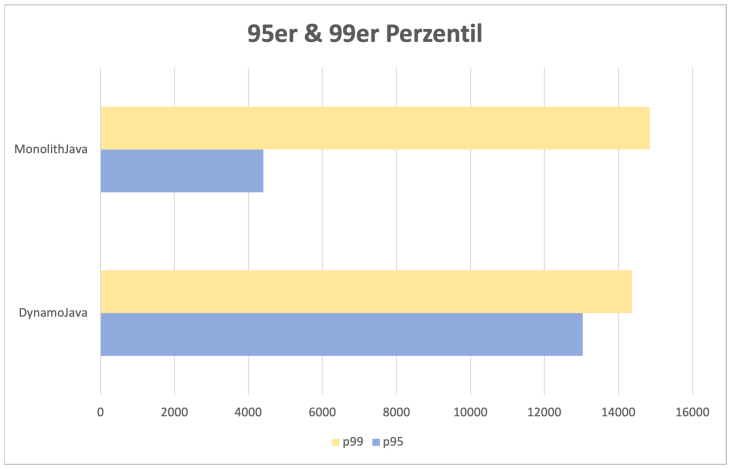


Abbildung 5.7: 95er und 99er Perzentil für MonolithJava im Vergleich zu DynamoJava

Die Ergebnisse zeigen im Vergleich zu DynamoJava eine deutliche Verbesserung. Die Antwortrate in Tabelle 5.7 hat sich um knapp vier Antworten pro Sekunde verbessert und vor allem das in Abb. 5.7 dargestellte 95% Perzentil ist um über die Hälfte gefallen von 13034 ms auf 4400 ms. Auch der Median in Tabelle 5.8 hat sich um fast 180 ms verbessert. Die Verbesserung lässt sich darauf zurückführen, dass jetzt nur noch eine Funktion einen Cold Start hat und nicht mehr alle. Leider reicht diese Optimierung wegen des Cold Starts lange nicht aus, um an die effizienten Stacks oder die Referenzarchitektur heranzukommen. Trotzdem ist dies eine schnelle Methode, um den Latenzen

Stack	Minimum	Maximum	Median
MonolithJava	17 ms	16289,2 ms	392,3 ms
DynamoJava	16 ms	16134,3 ms	568,5 ms

Tabelle 5.8: Min., Max. und Median der Antwortzeiten

Abhilfe zu schaffen. Die möglichen negativen Aspekte, welche im vorherigen Abschnitt über MonolithTs diskutiert wurden, bleiben natürlich zusätzlich bestehen.

Javaoptimierung – Kein Coldstart

Die letzte, aus dem vorherigen Versuch entstandene, Optimierung bezieht sich auf den Cold Start. Wie unter 5.3.3 erläutert, ist dies ein wichtiger Faktor für die Performance von Serverless. Gerade in Java hat er sich als verheerend erwiesen. Dieser Versuch dient nun dazu, eine gängige Strategie zu testen den Cold Start zu vermeiden. Der Stack wird daher als NoColdStartJava Referenziert. Die Optimierung wurde an dem MonolithJava Stack durchgeführt, da es einfacher und kostengünstiger ist eine Cold Start Vermeidung für eine einzelne, monolithische Funktion einzubauen. Die gewählte Strategie ist, die Funktion in regelmäßigen Abständen, hier 5 min., über einen neuen Ping-Endpunkt anzusprechen, der den Statuscode 200 zurückgibt. Das Pingen konnte über eine CloudWatch Rule in AWS realisiert werden, welche in dem CDK Stack definiert ist.

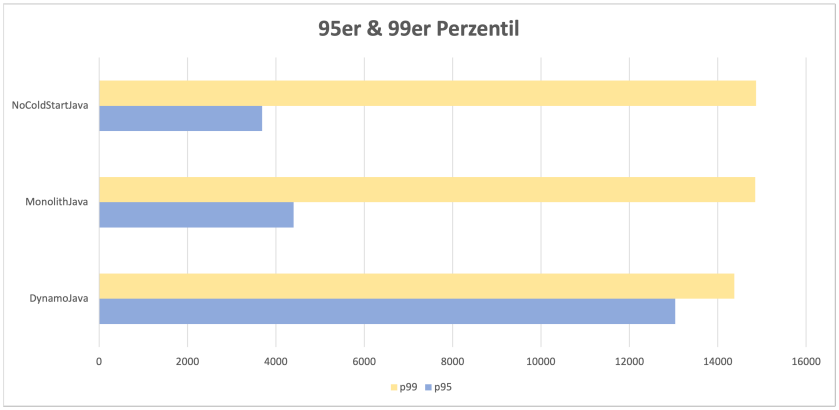


Abbildung 5.8: 95er und 99er Perzentile im Vergleich, für alle drei Java-Varianten

Die Ergebnisse zeigen, dass lediglich eine geringe Verbesserung stattgefunden hat. Tabelle 5.10 zeigt einen Anstieg der Antwortrate im Vergleich zum MonolithJava um 1,5

Antworten pro Sekunde. In Abbildung 5.8 ist das 95er Perzentil ist um weitere 700 ms gefallen und auch der Median in Tab. 5.9 singt leicht von 392 ms auf 331 ms.

Das die Verbesserung trotz Vermeidungsstrategie so unbeträchtlich ist, liegt an den

Stack	Minimum	Maximum	Median
NoColdStartJava	15,3 ms	16716 ms	331 ms
MonolithJava	17 ms	16289,2 ms	392,3 ms
DynamoJava	16 ms	16134,3 ms	568,5 ms

Tabelle 5.9: Min., Max. und Median der Antwortzeiten

Lambda Instanzen. Wie bereits unter 5.3.3 erläutert gibt es zwei Arten der Vermeidung, wobei hier das Pingen gewählt wurde. In diesem Fall reagiert also nur eine Instanz ohne Coldstart und diese reicht bei der Zahl an Anfragen im Lasttest nicht zur Bewältigung aus. AWS startet automatisch weitere Instanzen, welche wieder unter dem Phänomen leiden. Die Alternative, das nebenläufige Warmhalten der Funktionen über Provisioned Concurrency, ist viel teurer und wurde deshalb hier nicht getestet. Es ist davon auszugehen, dass bei der richtigen Menge an bereitgestellten Instanzen kein negativer Effekt durch Cold Starts zu verzeichnen wäre.

Stack	Completed	Failed	Response/sec
NoColdStartJava	226,7	0	26,5
MonolithJava	220,3	0	25
DynamoJava	225,3	0	21,4

Tabelle 5.10: Grundsätzliche Szenariowerte für alle drei Java-Varianten

Somit konnte Java nicht sehr stark optimiert werden. Die Ping-Methode eignet sich eher für einzelne, nicht häufig angesprochene Funktionen in nicht monolithischen Serverless Architekturen. Trotzdem konnte eine Verbesserung festgestellt werden und die Optimierungen waren nicht allzu kompliziert durchzuführen und konnten ohne extra Kosten getestet werden.

6 The Ugly - Einschränkungen & Grenzen

Serverless wurde bisher als sehr flexible Architektur beschrieben. Trotzdem existieren auch dabei Grenzen, die permanent sein können aber auch nur aktuelle Grenzen darstellen können. Zudem gibt es harsche Kritik, die Serverless als Konzept anzweifelt und es teilweise als bereits gescheitert betrachtet. All diese Aspekte werden im letzten Kapitel des Hauptteils behandelt und bieten letzte wichtige Punkte für die abschließende Diskussion.

6.1 Wo scheitert Serverless?

Die O'Reilly Serverless Studie hat gezeigt, dass trotz des jungen Alters von Serverless bereits 40% der befragen Unternehmen die Technologie adaptiert haben. Allerdings bleiben 60%, die aus den in Abb. 6.1 aufgeführten Gründen bisher kein Serverless nutzen. In der Studie findet sich zusätzlich eine Befragung zur Erfolgsquote nach der Adaption, welche vor allem für Unternehmen, die Serverless schon über drei Jahre nutzen sehr positiv ausgefallen ist. Insgesamt wird von denen die Serverless adaptiert haben der Einsatz als erfolgreich bis sehr erfolgreich beschrieben. Trotzdem berichtet von den anderen 60%, fast die Hälfte, dass auch in Zukunft keine Pläne bestehen Serverless zu nutzen [Roger Magoulas, 2019].
Die Frage ist also, woher kommen die Einschränkungen und was sind Grenzen, die manche Anwendungen von einer Serverlessnutzung ausschließen? Alles mit dem Hintergrund, dass Serverless als eine revolutionäre Technologie beschrieben wird, die vielleicht schon bald den Großteil der Server ersetzt.

6.1.1 Permanente Grenzen

Es gibt einige Eigenschaften von Serverless die für spezielle Anwendungen nicht tragbar sind. Zuallererst seien dabei die technischen Limits genannt, welche unter 5.3.4 erläutert

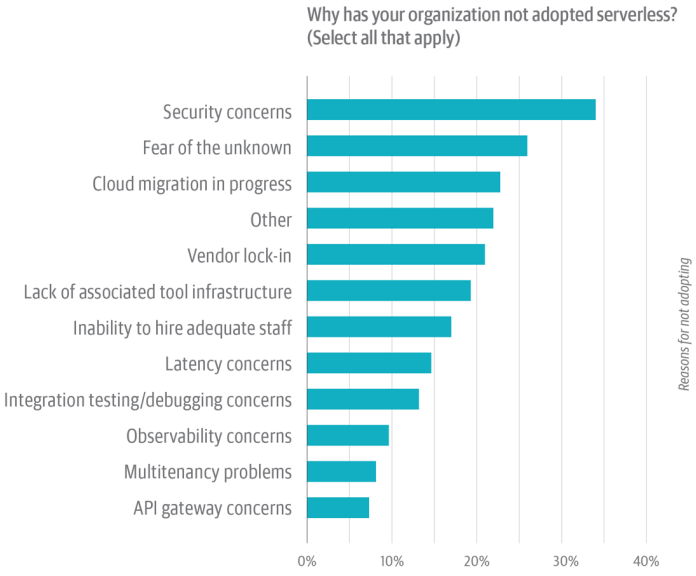


Abbildung 6.1: Gründe, warum Unternehmen Serverless bisher nicht eingetzt haben, zufolge der O'Reilly Serverless Studie 2019

wurden. Hat das System Anforderungen, die diesen nicht entsprechen, kann Serverless nicht eingesetzt werden. Diese Grenze ist recht trivial, aber es gibt natürlich auch spezielle Anforderungen, bei denen die Einschränkungen nicht direkt auf der Hand liegen. Dazu gehört zum Beispiel der Sicherheitsaspekt. Serverless wird im Normalfall auf einer öffentlichen Cloud Plattform betrieben. Es gibt Systeme die, eventuell sogar nach Unternehmensrecht, nicht auf öffentlichen Plattformen basieren können oder sollten, da die angebotene Verlässlichkeit nicht hinreichend ist. Auch die teils sehr unterschiedlichen SLAs (Security Level Agreements) von Drittanbieter Services können für einige Enterprise Use-Cases nicht ausreichend sein [Sbarski, 2017]. Unternehmen haben die Möglichkeit ihre Sicherheitsanforderungen anzupassen oder die benötigte Infrastruktur komplett privat oder als Hybrid zu entwickeln. Dies kann zum Beispiel rechtliche Probleme lösen, aber es entstehen auch neue Herausforderungen und großer Aufwand.
Eine weitere harte Grenze sind die obligatorischen Latenzen. Es gibt Systeme die eine garantierte, sehr niedrige Latenz für alle Aufrufe benötigen. Diese Garantien kann Serverless nicht bieten. Dazu kommt, dass die Anbieter bezüglich der Latenzen nicht sehr transparent sind und so der Einsatz in performance-kritischen Systemen weiter erschwert wird [Brode, 2020].

6.1.2 Aktuelle Grenzen

Serverless hat wenige strenge Grenzen, trotzdem gibt es viele weitere Gründe, warum es von Unternehmen bisher nicht genutzt wurde. Ein Überblick darüber sollen die aktuellen Grenzen geben.
Als ersten Punkt soll nochmal der Vendor Lock-In erwähnt werden. Während die Thematik bereits im vorherigen Kapitel besprochen wurde, lohnt sich auch hier ein Blick auf die O'Reilly Serverless Studie. Abb. 6.1 zeigt deutlich, dass der Lock-In definitiv eine Grenze ist. Er kann zu einem gewissen Grad umgangen werden oder sich in Zukunft verbessern, aber trotzdem hält der Umstand Unternehmen davon ab Serverless zu nutzen, eventuell dauerhaft.
Eine weitere Grenze ist der Cold Start. Im Gegensatz. zu den allgemeinen Latenzen, die unumgänglich sind, kann dieser zwar, wie unter 5.3.3 besprochen, umgangen werden, aber es bleibt zu bedenken, dass mit den aktuellen Warmhaltungstechniken ein anderer großer Vorteil von Serverless verloren geht. Die Kosteneffizienz und das Herunterskalieren auf null sind dann nicht mehr im gleichen Maße gegeben [Brode, 2020]. Darum ist es verständlich, dass Unternehmen, die keinen Cold Start in Kauf nehmen wollen, diesen als Grenze für die sinnvolle Nutzung wahrnehmen.
Es existieren zudem Perspektiven, die Serverless insgesamt sehr kritisch sehen. Ein Artikel argumentiert, dass es fragwürdig sei ganze Anwendungen mit Serverless zu realisieren, zumindest wenn vorher bereits Code besteht. Existiert beispielsweise bereits ein funktionierender Monolith ist es sehr kostenaufwendig eine klassische Serverlessarchitektur daraus zu bauen, nur damit am Ende duzende Funktionen mit zahllosen Gateways, Queues und Datenbankinstanzen verbunden sind, die eigentlich nicht gebraucht wurden. Die Grundaussage dabei ist, dass sich Serverless meist nur für grüne Wiese Projekte eignet, da bei einer Migration trotzdem von null gestartet werden müsse [Brode, 2020].
Ein Paper von Hellerstein et al. geht so weit zu sagen, dass Serverless zwar ein Schritt nach vorne geht, aber gleichzeitig zwei zurück. Die hauptsächliche Grenze für sie ist die Zustandslosigkeit. Die aktuellen Möglichkeiten hindern demnach verteilte Rechenmodelle, da die Funktionen nicht adressierbar sind und der Datenaustausch über sehr langsame und teure Speichermedien geschehen muss. Simple benötigte Abstimmungs- oder Wahlalgorithmen können nicht sinnvoll durchgeführt werden. Sie weisen darauf hin, dass es zwar das eventgetriebene, verteilte Rechenmodell mit einem globalen Zustand gibt, welches aber in Dualität mit ersterem existiert und auch dabei eine Kommunikation mit einem langsamen Speicher benötigt wird [Hellerstein, Faleiro, Gonzalez, Schleier-Smith, Sreekanti, Tumanov, and Wu, 2018].

Eine weitere aktuelle Grenze ist nach Hellerstein et al., dass es bisher nicht möglich ist spezielle Hardware mit Serverless zu nutzen. Die Auswahl und gezielte Nutzung von CPU und RAM ist durch die Provider stark begrenzt. Außerdem arbeitet Serverless nach der Prämisse benötigte Daten zum Code zu bringen, was für datenintensive Prozesse nachteilig ist. Zusammenfassend ist die Aussage, dass es nicht möglich sei, die unzähligen Kerne der Cloud effizient mit Serverless zusammenarbeiten zu lassen, außer für Fälle die eine unkoordinierte, extreme Parallelität vorweisen [Hellerstein, Faleiro, Gonzalez, Schleier-Smith, Sreekanti, Tumanov, and Wu, 2018].

Als letzten, ebenfalls sehr naheliegenden Aspekt, sei der Mangel an erfahrener Arbeitskraft genannt. Auch dieser kann eine Grenzen sein, da ohne erfahrene Entwickler:innen der Umstieg viel schwieriger ist. Dieser Punkt wurde auch in der Serverless Studie von nahezu 20% als Grund für keine Adaption angegeben. In Zukunft wird sich dies mit hoher Wahrscheinlichkeit aus den Grenzen heraus bewegen, aber momentan ist es eine legitime Einschränkung. Die aktuellen Möglichkeiten der Plattformen und von Drittanbietern können bei einigen dieser Grenzen bereits Abhilfe schaffen. Trotzdem dauert es lange bis Unternehmen diese Grenzen nicht mehr wahrnehmen, was auch die nicht vorhandenen Adaptions-Pläne bei fast der Hälfte der Befragten zeigen.

6.2 Grenzen als Use Cases

An dieser Stelle sollen zu Veranschaulichung einige Anwendungsfälle erläutert werden, die an die Grenzen von Serverless stoßen oder bei denen ein Einsatz von Serverless keinen Sinn ergibt, da die eigentlichen Vorteile nicht zum Tragen kommen.

Ein naheliegendes Beispiel für einen Fall, in dem Serverless zwar möglich, aber recht sinnlos ist, sind sehr konstant belastete Anwendungen. Wenn eine konstante, vorhersagbare Last auf den Servern ist, kann eine serverlose Lösung teurer sein, da kein Nutzen aus der extremen Skalierbarkeit gezogen wird [Coppel, 2019]. Die Betriebskosten können mit Serverless trotzdem geringer sein, aber die Migration ist, wie oben erwähnt, meist recht aufwendig.
Noch problematischer sind Systeme die Echtzeitdaten verarbeiten. Die Latenzen machen dies quasi unmöglich, da z.B. durch Cold Starts keine Performance zugesichert werden kann [Coppel, 2019]. Die oben zitierten Hellerstein et al., welche Serverless allgemein als nicht wirklich nützlich, bzw. sogar im Gegensatz zum modernen Cloud Entwicklung

sehen bringen drei weitere Use Cases zur Sprache, in welchen Serverless scheiterte. Zum einen ist das Trainieren von KI-Modellen nicht gut geeignet. In ihren Tests war die serverlose Architektur 21-mal langsamer und gut 7 Mal teurer als das gleiche Training auf einer EC2 Instanz. Der Grund dafür ist, dass bei dem FaaS Ansatz die Daten zum Code geholt werden müssen und nicht der Code auf den Daten ausgeführt werden kann und dies kostet Zeit. Des Weiteren haben sie versucht ein Batchverarbeitungssystem welches Vorhersagen mit geringer Latenz machen soll. Dieses System hat eigentlich dedizierte Hardwareanforderungen welche weder von einer EC2 Instanz, aber noch weniger von Serverless erfüllt werden konnten. Auch alle Versuche sinnvolles verteiltes Rechnen zu realisieren, scheiterten an viel zu langsamer Ausführung der Wahl des Koordinatorprozesses und zu hohen Kosten für den Austausch über das Speichermedium (in diesem Fall die DynamoDB) [Hellerstein, Faleiro, Gonzalez, Schleier-Smith, Sreekanti, Tumanov, and Wu, 2018].

6.3 Praktische Umsetzung III

Im letzten Teil der Umsetzung sollen Grenzen aufgezeigt werden, die an der entwickelten Anwendung erfahren wurden. Natürlich war es nicht möglich die Anwendung in jedem Stack an die technischen Grenzen zu bringen, da dies zu hohen Kosten geführt hätte. Zudem musste aus zeitlichen Gründen darauf verzichtet werden, komplett neue Anwendungen zu bauen, die eher Grenzfälle aufzeigen würden, wie die Beispiele unter den eben beschriebenen Use Cases. Trotzdem gibt es einen naheliegenden Stack der serverlose Anwendungen schnell an die Grenzen bringt. Dieser soll im Folgenden dargestellt werden. Zum Abschluss der praktischen Umsetzung wurde als Kontrast die Referenzarchitektur als klassische Spring Boot Anwendung, ohne Serverless, umgesetzt und getestet.

6.3.1 Relationale Datenbank

Bereits in den Grundlagen unter Best Practices wurde erwähnt, dass keine auf einer Verbindung basierenden Services genutzt werden sollten. Dies wurde unter 5.3.1 weiter ausgeführt und soll nun erprobt werden. Relationale Datenbanksysteme (RDBS) sind ein Standard, der noch häufig eingesetzt wird, vor allem wenn komplexe Abfragen benötigt werden. Entwickler:innen kennen sich meist besser mit ihnen aus als mit neueren NoSQL Datenbanken und schrecken eventuell vor deren Nutzung zurück. Aus diesem Grund ist

auch deren Einsatz in Kombination mit Serverless zu diskutieren
Die Frage, die sich stellte war, ob auch schon bei einem kleinen Projekt so eine große Schwierigkeit besteht, wenn ein RDBS eingesetzt wird. Für die Umsetzung des Versuchs musste eine relationale Datenbank ausgewählt werden, welche die DynamoDB ersetzt. Eine naheliegende Lösung war das beliebte mySQL, welches sich problemlos mit dem Amazon Relational Database Service (RDS) integrieren lässt. Im kostenlosen Kontingent ist eine db.t2.micro Instanz enthalten. T2 ist eine standard Instanzklasse, die eine Basis-CPU-Leistung bereitstellt und gleichzeitig in der Lage ist, Spitzenlasten zu verarbeiten. Das micro weist aber schon darauf hin, dass diese nicht sehr Leistungsstark ist. Kostenlos sind 750 Stunden Betrieb verteilt auf alle Instanzen, 20 GB SSD-Mehrzweck-Datenbankspeicher und 20 GB Sicherungsspeicher für automatisierten Datenbanksicherungen. Das standard Verbindungslimit wurde nicht erhöht, da die 750 Stunden dann zu schnell aufgebraucht werden würden. In Tabelle 6.1 wird deutlich, dass eine relationale

Stack	Completed	Failed	Response/sec
SQLTs	0	216	14,6
DynamoTs	219,7	0	29,7

Tabelle 6.1: Grundsätzliche Szenariowerte für SQLTs

Datenbank in diesem Szenario überhaupt nicht funktioniert, weshalb das Experiment unter die Grenzen von Serverless fällt. Kein Testdurchlauf konnte erfolgreich abgeschlossen werden, weshalb auch die Antworten pro Sekunde keine Aussagekraft haben. Leider gilt dies auch für die restlichen Messwerte. Abb. 6.2 stellt den Grund für das Fehlverhalten bildlich dar. Die oben beschriebene Instanz lässt 65 Verbindungen zu, danach werden 5xx Fehler geworfen. Die Anzahl der Fehler steigt also an und die einzelnen Testszenarien werden mit einem Timeout auf Grund von zu vielen Verbindungen beendet. Danach bauen sich die aktiven Instanzen langsam ab.

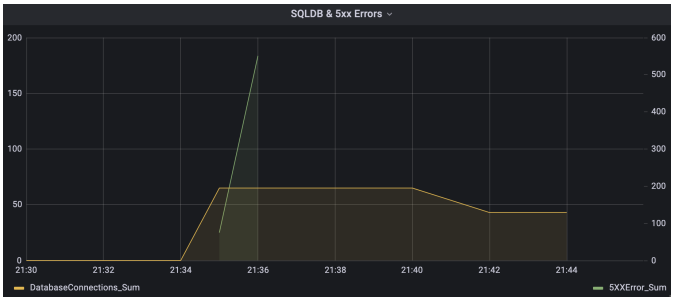


Abbildung 6.2: SQLTs aktive DB Verbindungen vs. 5xx Errors

Es wurde versucht das Problem zu beheben, indem eine "retry"-Funktion implementiert wurde, die bei einem Fehler die Anfrage bis zu dreimal wiederholt. Dies hat allerdings nicht geholfen, da die belegten Instanzen viel zu langsam wieder verfügbar werden. Die einzige in diesem Fall funktionierende Möglichkeit wäre eine Anpassung der Instanz an den Test gewesen. Da diese Änderung zum einen sehr teuer wäre und zum anderen langfristig keine Lösung ist, musste das Fehlverhalten hingenommen werden. Die Instanz zu verbessern hätte keinen Sinn ergeben, da die Tests zwar klein sind, aber sich natürlich größer skalieren lassen. Also auch in einer normalgroßen Anwendung würden die Instanzen sehr schnell an Ihre Grenzen kommen, auch wenn die Zahl der möglichen Verbindungen höher ist. Somit ist diese Art von RDB schlicht nicht mit Serverless vereinbar, weil es entweder gar nicht geht oder unverhältnismäßig große Instanzen genutzt werden müssen, welche zudem nicht mit-skalieren können.

Stack	Minimum	Maximum	Median
SQLTs	33 ms	541 ms	137 ms
DynamoTs	16,7 ms	2254 ms	78 ms

Tabelle 6.2: Min., Max. und Median der Antwortzeiten Für einen Nutzer

Die vielen Verbindungen kommen daher, dass jede einzelne Lambdafunktion die eine Anfrage an die Datenbank macht, eine eigene Verbindung öffnet, was natürlich nicht tragbar ist. In Tabelle 6.2 sind die Antwortzeiten für den Test mit einem Nutzer dargestellt. Dabei wird deutlich, dass wenn die Verbindungen nicht Überhand nehmen, alles einwandfrei funktioniert und auch die Zeiten in Ordnung sind. Ein Nutzer benötigt im Test acht verschiedene Lambdafunktionen. Benötigen diese auf Grund von hoher Last dann noch weitere Instanzen, die auch alle acht Verbindungen aufbauen, ist das Limit schnell erreicht.
Abschließend lässt sich feststellen, dass die einfache, traditionelle Nutzung einer mySQL Datenbank nicht funktioniert. Wie in Abschnitt 5.3.1 beschrieben, gibt es Dienste wie Aurora Serverless, mit denen AWS die Nutzung von RDBS und Serverless ermöglicht. Aurora Serverless konnte leider nicht getestet werden, da kein kostenloses Kontingent vorhanden ist. Grundsätzlich ist aber von RDBS abzuraten, da sie für die Funktionalität nicht zwingend benötigt werden, wie ebenfalls unter 5.3.1 erläutert wurde.
Ein zusätzlicher Kritikpunkt, der während der Nutzung von AWS RDS aufgetreten ist, bezieht sich auf das kostenlose Kontingent. Amazon wurde schon häufig für das intransparente Preismodell und die verstecken Kosten kritisiert. Auch hier standen nach kurzer

Zeit bereits mehrere Dollar auf der monatlichen Rechnung, obwohl sich an die kostenlosen Kontingente gehalten wurde. Bei der Nutzung der Datenbank wird ein VPC (virtual private Cloud) Endpunkt benötigt, die VPC an sich ist scheinbar kostenlos, aber der benötigte Endpunkt ist es nicht. Dieser kostet pro Stunde Geld, egal ob aktiv oder nicht. Dies zeigt genau das, was mit versteckten Kosten unter 5.3.4 gemeint war. Hier konnte nach den Tests der Endpunkt gelöscht werden, aber verstecke Kosten dieser Art können in Produktion kritisch werden.

6.3.2 Vergleich – Klassische Anwendung

Nachdem nun über die Grenzen von Serverless gesprochen wurde und auch im vorherigen Kapitel negative Aspekte aufgegriffen wurden, steht eine Frage im Raum: Ist der traditionelle Weg am Ende doch besser?
In diesem Abschnitt wurde darum die Anwendung mit einem klassischen Java und Spring Boot Stack umgesetzt. Sie wurde in einen Docker Container verpackt und auf einer Amazon EC2 Instanz deployed. Da der Code mit Spring entwickelt wurde, musste dieser neu geschrieben werden. Allerdings konnten viele DynamoDB API spezifische Codezeilen vom DynamoJava Stack übernommen werden. Die Logik an sich ist natürlich gleichbleibend, nur die interne Struktur ist nicht mehr in kleinteilige Funktionen unterteilt, sondern in klassische Repositories und Services. Der Stack heißt im Folgenden NonServerless.

Für das Deployment musste sich ebenfalls an das kostenlose Kontingent von EC2 gehalten werden. Leider beinhaltet dies, wie schon bei der Datenbank, nur eine t2.micro Instanz, welche mit allen aktiven Instanzen 750 Stunden betrieben werden kann. Mit diesen Serverinstanzen konnte der Test gar nicht erfolgreich ausgeführt werden. Damit das Experiment nicht scheiterte wurden drei weitere Instanzen für die Zeit des Tests aktiviert. Dafür wurde ein Cluster (genauer schreiben) benötigt, welcher kein kostenloses Kontingent besitzt. Tabelle 6.3 zeigt, dass vier Instanzen zwar einen Großteil, aber nicht alle Durchläufe beenden konnten. Trotzdem sind die Performancewerte dadurch, wie bei SQLTs, schwerer zu Bewerten. Abbildung 6.3 demonstriert das Problem anhand der Perzentile deutlich. Der 95er Wert ist recht gering mit 413 ms und der 99er Wert

Stack	Completed	Failed	Response/sec
NonServerless	191,7	34	23,9
DynamoTs	219,7	0	29,7

Tabelle 6.3: Grundsätzliche Szenariowerte für NonServerless

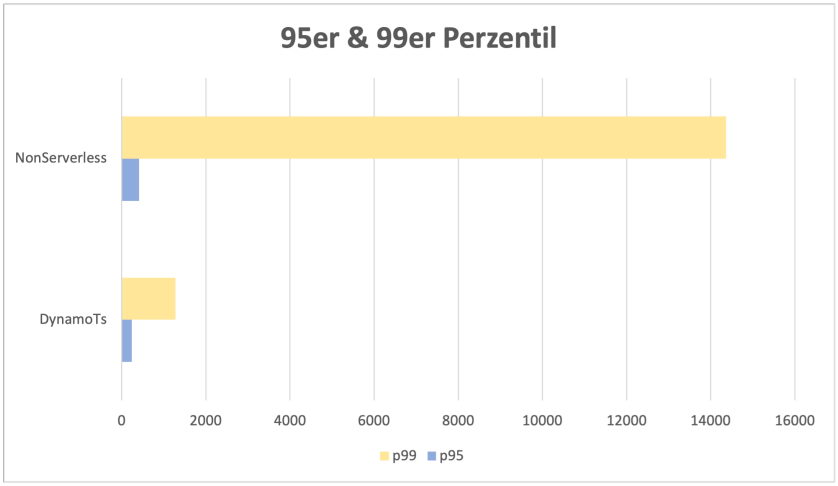


Abbildung 6.3: 95er und 99er Perzentil für NonServerless im Vergleich zu DynamoTs

liegt bei 14368 ms, was darauf zurückzuführend ist, dass die Datenbank aufgrund der Schwäche der Instanz immer langsamer antwortet bis schließlich Timeouts auftreten. Da aber die meisten Tests beendet wurden lohnt sich ein Blick auf Tabelle 6.4. Dort ist der sehr niedrige Median von 29 ms zu sehen. Die Min. und Max. Werte liegen ähnlich weit auseinander wie die Perzentile. Allerdings ist der minimale Wert mit 14 ms niedriger als bei allen anderen, da andere Technologien unterliegen und es mit diesen anscheinend etwas schneller gehen kann.

Abschließend lässt sich zum NonServerless Stack feststellen, dass er potenziell um einiges

Stack	Minimum	Maximum	Median
NonServerless	14 ms	19112,67 ms	29 ms
DynamoTs	16,7 ms	2254 ms	83 ms

Tabelle 6.4: Min., Max. und Median der Antwortzeiten

schneller ist als die anderen Serverless Stacks. Dies könnte dafürsprechen, dass ein Monolith für simplere Anwendungen die bessere Wahl ist. Allerdings war die Entwicklung weniger intuitiv und muss auf AWS von Beginn an bezahlt werden, was das Experimentieren erschwert. Demnach kann, wenn es leichter fällt, eine monolithische Architektur entwickelt werden, aber Serverless ist dann trotzdem eine sinnvolle Deploymethode. Diese Lösung ist ggf. nicht langfristig haltbar, funktioniert hier aber gut, wie MonolithTs zeigt.

Grundsätzlich sind die Grenzen von Serverless also sehr individuell und meist nur bei spezielleren Anwendungen kritisch. Trotzdem konnte auch im Rahmen der hier entwickelten Anwendung eine Grenze erlebt werden. Einige Dinge, wie das Versagen einer monolithischen Funktion, konnten nicht getestet werden, da dies voraussichtlich zu viele Ressourcen gekostet hätte. Dennoch liefert dieses gesamte Kapitel einen bisher fehlenden, kritischen Eindruck, um in die kommende Diskussion starten zu können.

7 Diskussion

In diesem Kapitel sollen in erster Linie die drei vorhergegangen Kapitel, The Good, The Bad and The Ugly abschließend diskutiert werden. Es geht darum zu evaluieren was realistische Grenzen sind, wo Serverless sich lohnt und was in der Zukunft zu erwarten ist. Hat die Server-Revolution schon begonnen oder ist es doch overhyped?
Neben diesen Punkten soll ein Resümee zu den Umsetzungsergebnissen gezogen werden.

7.1 Lohnt sich Serverless?

Viele Aspekte in dem für und wider von Serverless basieren darauf, dass Serverless eine neue Technologie ist. Für die eine Seite ist sie revolutionär und aufregend, während die anderen gerade auf Grund der Novität zweifeln, aber an die Entwicklung und Evolution der Plattformen glauben. Dabei ist die Serverless-Bewegung mit Blick auf die vorherigen X-as-a-Service Paradigmen recht naheliegend. Trotzdem scheint Serverless so neu, dass Nachteile mit Nachsicht behandelt werden.
Ein kritischerer Artikel von Bernard Brode weist nun darauf hin, dass Serverless gar nicht neu ist. Die Idee existierte nach Brode schon lange, bereits 2006 gab es in Form der PaaS Plattform Zimki, wie auch bei Google entsprechende Ansätze [Brode, 2020]. Ist die Idee also nicht so neu wie erwartet, warum wird Serverless dann erst jetzt so stark gehyped? Zum einen hat sich natürlich das Konzept, wenn auch nicht neu-, zumindest weiterentwickelt und es könnte argumentiert werden, dass es erst jetzt wirklich Relevanz hat. Ähnlich vermutet auch Brode, dass gerade heute der Bedarf an Computing Resources rapide ansteigt und auch weniger entwickelte Länder immer mehr in den E-Commerce Sektor vordringen. Diese Länder haben schlicht nicht die Infrastruktur, um ihre Anwendungen selbst zu betreiben [Brode, 2020]. Während dies natürlich nicht der einzige Grund ist, handelt es sich sicherlich um einen Accelerator.
Neben der neutralen Annahme, dass die Nachfrage nach Rechenressourcen gestiegen ist

und Serverless durch den einfachen Zugang dazu einen Hype erlebt, gibt es natürlich auch andere Argumente. Wie schon im Kapitel The Good dargestellt, sind neben der klassischen Kosteneinsparung und Skalierbarkeit, vor allem die verkürzte Vorlaufzeit und die Usability Faktoren, die für Serverless sprechen. Roberts berichtet von nie dagewesenen Möglichkeiten und einer revolutionären Entwicklung [Roberts, 2018]. Entwickler:innen können sich mit Serverless endlich auf die Teile der Arbeit konzentrieren, die ihnen am Meisten Spaß machen, nämlich das Bauen der Anwendung [Coppel, 2019]. Auch die Serverless Studie in Abb. 7.1 verzeichnet gerade bei den Unternehmen, die Serverless länger als drei Jahre nutzen große Erfolge und so gut wie keine Misserfolge.

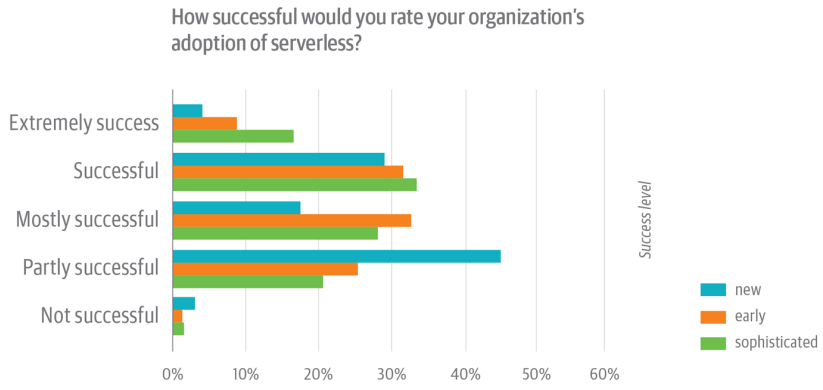


Abbildung 7.1: Erfolgsquote der Serverless Adaption unter den befragten Unternehmen, aufgeteilt nach Serverless-Erfahrungslevel

Trotzdem haben die Kapitel The Bad und The Ugly gezeigt, dass nicht alles positiv ist. Vieles kann sich noch entwickeln oder ist nur in speziellen Fällen ein Nachteil, aber trotzdem stellt sich die Frage, für wen lohnt sich Serverless wirklich?
Der erneute Blick auf Abb. 7.1 zeigt, dass im Laufe der Zeit die Erfolge größer werden, allerdings wirkt es hier so, als würde die Nutzung von Serverless letztendlich immer erfolgreich sein. Wahrscheinlicher ist aber, dass die Unternehmen Serverless nur für kleine, besonders gut geeignete Teile der Entwicklung nutzen, denn sonst wäre die Erfolgsquote wahrscheinlich geringer. Diese Annahme unterstützt auch eine aktuelle Studie über den Zustand von Serverless. In Abb. 7.2 wird deutlich, dass 65% der Cloudformation Stacks (AWS) nur eine Lambda Funktion enthalten. Es ist also wahrscheinlich, dass viele Unternehmen Serverless bisher nur ausprobiert haben oder sich in sehr frühen Stadien der Nutzung befinden. Des Weiteren können auch nur kleine Teile der eigentlichen Applikation Serverless nutzen, was die gleich folgenden negativen Thesen stützen würde.

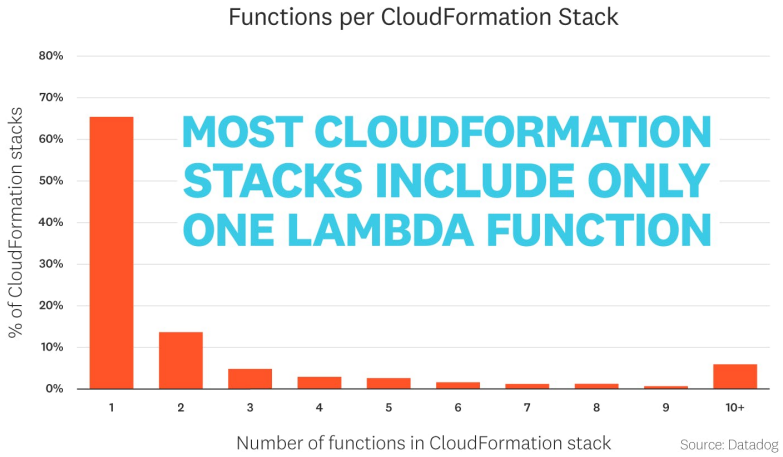


Abbildung 7.2: Anzahl der Funktionen in CloudFormation-Stacks (DataDog)

Allerdings zeigt diese Studie auch, dass Lambda Funktionen seit 2019 3,5-mal häufiger aktiviert wurden und immer kürzer laufen, was eventuell ein Hinweis auf die Verfolgung der Best Practices ist. Zudem gewinnen auch Google und Microsoft mehr an Momentum, was für Serverless positiv auszulegen ist [Datadog, 2021]. All diese Punkte zeigen eher einen positiven Verlauf, der sich noch zu steigern scheint und Serverless eine große Zukunft versprechen könnte.
Während in den vorherigen Kapiteln sowohl die typischen als auch die nicht geeigneten Use Cases besprochen wurden, hat der bereits zitierte Bernard Brode diesbezüglich eine einseitigere Meinung. Er geht davon aus, dass serverlose Architekturen nicht für gesamte Applikationen geeignet sind und somit auch Server nicht mit Serverless ersetzt werden können. Serverless ist nicht geeignet, da es nicht kosteneffizent sei, und die Migration zu aufwendig sei, da normalerweise von null gestartet werden müsse [Brode, 2020]. Das kritische Paper von Hellerstein et al. geht sogar noch weiter und bezeichnet Serverless als eine nur für Spezialfälle, mit einem automatisch skalierenden Backend, geeignete Architektur. Nach ihnen kann Serverless keinesfalls das Problem lösen, die Cloud allgemein für alle Zwecke nutzbar zu machen. Sie bezeichnen die Idee auch keineswegs als revolutionär, sondern als eine Herangehensweise, die schon zuvor jeder in der Cloud mit Hilfe von Orchestrierungsplattformen umsetzen konnte [Hellerstein et al., 2018]. Diese negative Sicht wurde hier nicht aus polemischer Motivation heraus beschrieben, stattdessen verspricht sich Hellerstein et al. viel mehr von den Möglichkeiten die FaaS eröffnet. Sie bezeichnen Serverless als lokales Minimum in der Entwicklung und erwarten ein Neudenken, welches eventuell zu einer wirklichen Revolution führen kann. Die aktuell größten

Probleme sind, dass FaaS momentan hardwaregetriebene Softwareinnovation verhindert und zudem Open Source Service Innovation entmutigt. Für sie steht das das Serverless-paradigma entgegen dem modernen Entwickeln [Hellerstein et al., 2018].
Diese Annahmen würden Serverless aktuell im Tief von Gartners Hype Cycle lokalisieren. Dem stimmt auch ein weiterer Artikel zu. Serverless sollte nicht überschätzt werden, wie es mitten im Hype häufig der Fall ist. Gerade die Kostenersparnis sei nicht so hoch wie erwartet, die Services seien teuer für das was geboten wird und Serverless an sich, habe einen sehr limitierten Nutzen. Der hier referenzierte Artikel erläutert, dass nur ein sehr kleiner Teil der Unternehmen eine zustandslose, eventbasierte Applikation hat, die zusätzlich extrem unterschiedlicher Last ausgesetzt ist. Denn nur dann kann sich Serverless wirklich lohnen. Serverless ist demnach eher ein weiteres Werkzeug unter vielen und nicht revolutionär [Martin, 2018]. Allerdings bietet sich gerade für moderne Webapps eine eventbasierte Architektur an, da diese Anforderungen, wie gute Skalierbarkeit und Verfügbarkeit, entspricht [Sucaciu, 2020]. Eventbasierte Architekturen eignen sich sehr gut für Serverless und könnten damit in Zukunft weit verbreitet sein.
Wo genau die Wahrheit liegt, bleibt also bis zum Ende unklar und kann letztendlich nur die Zukunft zeigen. Der Hype ist trotzdem echt und sorgt auch dafür, dass Serverless teilweise eingesetzt wird, wo es keinen Sinn ergibt. Allerdings ist noch nicht 100-prozentig klar wo Serverless langfristig Sinn ergibt und wo es zu umständlich oder nicht gewinnbringend ist. Wo genau sich Serverless heute auf Gartners Hype Cycle befindet, ist wahrscheinlich sehr subjektiv. Aber ein Hype ist nach Gartner definitiv nicht durchgehend positiv, bis ein produktives Level erreicht werden kann. Eine komplette Serverrevolution scheint unrealistisch, während im Bereich der Webentwicklung ein sehr verbreiteter Einsatz eher möglich erscheint.
Für einen realistischen, aber positiven Abschluss eignet sich am besten folgendes Zitat:
“Should we invest our money and time into an ecosystem that, in the end, may not save money, reduce complexity, or speed up deployments?
If you inherently believe that platforms improve over time, then the answer is yes, of course.” [Coppel, 2019].
Denn egal ob Serverless als lokales Minimum oder bereits jetzt als revolutionäre Arbeitsweise gesehen wird, die Entwicklung geht unaufhaltsam weiter. Ob zukünftige Produkte noch als Serverless bezeichnet werden oder nicht spielt dabei fast keine Rolle. Grundsätzlich sind sich Führsprecher und Kritiker schließlich einig, dass die Zukunft noch Großes bereithält, egal unter welchem Namen.

7.2 Praktische Umsetzung

Die Arbeit mit Serverless war völlig unbekannt und somit eine Herausforderung an sich. Trotzdem waren insbesondere die ersten Schritte mit Serverless relativ einfach. Dies stützt die These, dass Serverless gut für unerfahrene Programmierer:innen geeignet ist. Trotzdem gab es auf dem Weg einige Fallhöhen.

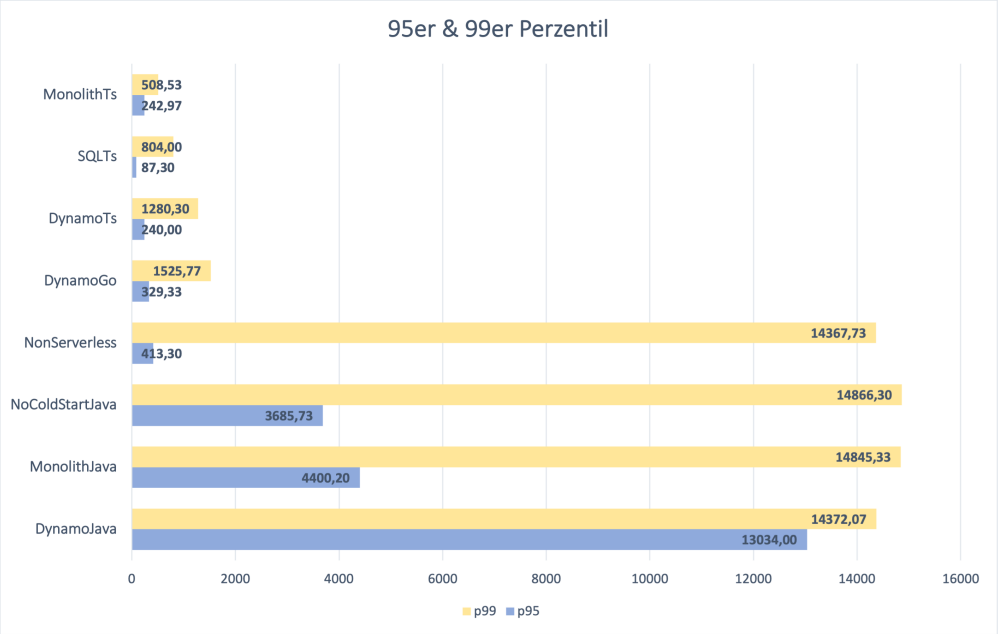


Abbildung 7.3: Alle 95er und 99er Perzentile im Vergleich

Die Ergebnisse, welche in Abb. 7.3 erneut im Gesamten dargestellt sind, zeigen dass viele verschiedene Stacks mehr oder weniger erfolgreich umgesetzt werden konnten. Sie zeigen allerdings nicht, dass auch viele Ideen verworfen oder abgebrochen werden mussten. Meist auf Grund von zu hohen Kosten oder Aufwand. Trotzdem konnten viele neue Erkenntnisse gewonnen werden und gerade die drei besten Stacks – MonolithTs, DynamoTs und DynamoGo – haben viel Potential weiter entwickelt zu werden. Es wäre spannend längere Tests auszuführen und erneut Vergleiche aufzustellen. Zudem gibt es einige Optimierungen die potenziell erkundet werden könnten, um eine wirklich optimale Implementierung zu erhalten, die nicht von den kostenlosen Kontingenten eingeschränkt ist. Unter dieser Prämisse könnten auch Stacks wie SQLTs, die aktuell an Grenzen gestoßen sind, optimiert

werden. Aurora Serverless wäre dabei eine sehr spannende Technologie. Eine weitere Untersuchung der monolithischen Stacks könnte ebenfalls spannend sein. Diese haben in dieser Arbeit sehr gut performed, werden aber eigentlich als Anti-Pattern bezeichnet. Es stellt sich also die Frage, ab wann es sich nicht mehr lohnt monolithisch zu Entwickeln.
Eine sehr positive Erfahrung war die Handhabung der NoSQL Datenbank DynamoDB. Sie war in diesem Fall leicht aufzusetzen und angenehm frei in der Nutzung, im Gegensatz zu den traditionellen SQL-Datenbanken.
Das allgemein schwierigste an der Entwicklung waren meist die unzähligen Konfigurationen die für die Stacks vorgenommen werden mussten, vor allem weil dabei keine Vorerfahrung vorhanden war, die diesen Schritt zu einem routinierteren gemacht hätte. Auch die Entwicklung mit Java war, wie unter 5.4.3 beschrieben, schwieriger als erwartet und durch die relativ schlechten Ergebnisse nicht empfehlenswert für diesen Use Case.
Schlussendlich war die Referenzarchitektur der Stack, mit dem am besten gearbeitet werden konnte und der die beste Performance gezeigt hat. Das könnte daran liegen, dass viele Empfehlungen und Beispielarchitekturen von AWS einen ähnlichen Stack verfolgen. Für größere, komplexere Anwendungen bietet sich möglicherweise eher Go als Sprache an, da es damit effizienter sein könnte. Aus dieser Arbeit geht hervor, dass monolithische Funktionen, zumindest für den Start, ebenfalls empfehlenswert sind. Sie haben eine sehr gute Performance und wahrscheinlich den minimalen Konfigurationsaufwand. Java und SQL-Datenbanken finden in der Zukunft der hier entstandenen Anwendung keinen Platz, aber mit der Entwicklung der Plattformen kann sich dies natürlich auch ändern.

7.3 Zukunft

Nach der Diskussion über Serverless an sich und den Erfahrungen und Ergebnisse der entwickelten Anwendung, lohnt sich nun ein Blick auf potenzielle Zukunftsvisionen. Gerade die Zukunft ist für Serverless ein spannendes Thema, welches sich in viele Richtungen entwickeln kann. Serverless ist neu genug, um viele offene Forschungsfragen aufzuwerfen, aber auch alt genug, um weitere neue Technologien zu befeuern. Die große Frage ist letztendlich, wie sieht Serverless aus, wenn Gartners Hype Level der Produktivität erreicht wird.
Die Hoffnung für Serverless ist, dass die unter The Bad genannten Herausforderungen und Probleme sich bald verbessern. Roberts stellt dahingehend einige Vermutungen auf. Er sieht die aktuelle Nutzung von FaaS als eine Art Muster, um Code in zustandslosen Funktionen umzuwandeln. Während dies Berechtigung hat, erwartet er in der Zukunft

mehr Abstraktionen und eventuell Sprachen die dafür sorgen, dass FaaS genutzt werden kann, obwohl die Anwendung keine Sammlung verteilter Funktionen ist. Zudem kann er sich vorstellen, dass die Vorteile von Serverless auch on-premise genutzt werden können. Einen ähnlichen Verlauf hat er bei PaaS beobachtet, welches nach dem Start als reiner Cloud Service, heute auch vermehrt lokal eingesetzt wird. Allerdings hält er hybride Lösungen für wahrscheinlicher [Roberts, 2018].
Ein weiterer Bereich in dem Entwicklung zu erwarten ist liegt in den Operations. Bereits heute gibt es weniger Systemadmins als früher und Serverless wird dies weiter verstärken. Daraus resultiert, dass die DevOps Kultur weiter gefördert wird, da die Hauptaufgabe nun bei den Entwickelnden liegt. Diese werden also vermehrt lernen müssen mit dem Betrieb umzugehen. Weitere Bildung in diese Richtung, wie auch enge Zusammenarbeit sind zukünftig umso wichtiger [Roberts, 2018].
Zuletzt geht Roberts auf die Entwicklung auf Seite der Anbieter ein. Während der Migration zwischen Anbietern zwar schwierig ist, wird dies Unternehmen nicht davon abhalten unzuverlässige, undurchsichtige Plattformen zu wechseln. Er erwartet, dass die Plattformen darum transparenter und deutlicher damit werden, welche Erwartungen an sie gestellt werden können [Roberts, 2018].

Die Berkeley Universität hat sich im Cloud-Umfeld bereits einen Namen gemacht, da sie 2009 Vorhersagen für die Cloud getroffen haben, welche heute zum Großteil wahr geworden sind. Zehn Jahre später, im Jahr 2019, haben sie nun Vorhersagen für die Zukunft des Serverless Computing gemacht:

- Es entstehen neue Speicherlösungen, damit mehr Typen von Applikationen gut auf Serverless laufen können.
- Beim Serverless Computing wird es einfacher werden sicher zu entwickeln als beim klassischen Serveransatz, auf Grund des hohen Abstraktionslevels und der feingranularen Isolation der Funktionen.
- Das Abrechnungsmodell wird sich so weiterentwickeln, dass die Kosten für fast jede Art von Applikation weniger werden oder zumindest nicht mehr.
- Serverful Cloud Computing wird es nur noch zur Unterstützung von (erweitertem) BaaS geben. Es wird zwar noch existieren, aber viel weniger relevant sein, wenn Serverless seine aktuellen Limitierungen überwindet.

- Serverless wird das standardmäßige Paradigma der Cloud Ära werden und in großen Teilen serverful Cloud Computing ersetzen. Es wird die Client-Server Ära zu Ende bringen.

Grundsätzlich ist Berkeleys Blick auf Serverless sehr positiv. Sie gehen davon aus, dass durch die vermehrte Adaption auch ein breiterer Einsatz ermöglicht wird, statt nur einfache Webservices oder einzelne Funktionen, die AWS Services verbinden, umzusetzen. Allerdings bedeutet die sich weiter erhöhende Komplexität schon heute, dass Entwickler:innen verstärkt Architekturkenntnisse benötigen, um serverlose Applikationen bauen zu können. Somit ist es noch ein weiter Weg bis diese Vorhersagen überhaupt erreicht werden können, sie erfordern eine Überwindung der aktuellen Limitierungen und eine Entwicklung unter den Anwender:innen.
(Vgl. [Jonas et al., 2019])

Offene Forschungsfragen

In dieser Arbeit wurde klar, dass Serverless noch viele Limitierungen und Probleme hat, die für eine positive Zukunft überwunden werden müssen. Daher beschäftigt sich dieser Abschnitt mit noch offenen Forschungsfragen, die dafür ggf. geklärt werden müssen oder auch zu neuer Innovation führen können.
Von den negativen Punkten ausgehend argumentieren Hellerstein et al., dass Serverless erst in der Lage sein muss, die echte Rechenleistung und Speicherkapazität der Cloud zu nutzen, und zwar in einer automatisch skalierenden und kosteneffizienten Weise. Da dies eine offenes Forschungsthema ist haben sie natürlich keine Lösung, räumen aber ein, dass mit weiterer Forschung ein Framework entstehen kann, dass über aktuelles FaaS hinauswächst. Dort könnten dann dynamisch Ressourcen verteilt werden und nutzerspezifische Performanceziele für Rechenzeit und Daten erreicht werden [Hellerstein et al., 2018].
Roberts erhofft sich konkreter, dass Patterns für Serverless aufkommen. Dies sollen möglichst die Frage nach sinnvollem Logging und Debugging in hybriden Systemen, mit FaaS, BaaS, wie auch traditionellen Servern, beantworten. Ihm zufolge werden in erster Linie die Provider selbst Muster und Lösungen erforschen und präsentieren [Roberts, 2018].
Während es naheliegend ist, dass sich Tooling und Zustandsmanagement weiterentwickeln müssen, kann auch das automatische Skalieren weiter erforscht werden. Zum Beispiel könnte die aktuelle Skalierung in Zukunft eventuell mit KI-Vorhersagen noch effizienter gestaltet werden [Wu, 2019].
Ein letzter wichtiger Forschungsaspekt sind Performance-Garantien. Heute ist es teilweise

ein Hindernis, dass Performance nicht garantiert werden kann. Dieses Thema sollte daher, vor allem von den Providern, weiter verbessert werden, damit Serverless zuverlässig eingesetzt werden kann [Wu, 2019].

Weiterführende Forschung – Beyond Serverless

Abschließend soll nun ein Blick auf Technologien und Forschung geworfen werden, die in Bereichen einen Schritt weiter als Serverless gehen, bzw. daraus erwachsen.
Es bestehen viele Möglichkeiten wie KI in oder mit Serverless eingesetzt werden könnte. Im Buch "What is Serverless?" wurde beschrieben, dass sich eine neue Art von KI-Serverless ausbreitet. Darunter fällt beispielsweise Lex, Amazons Spracherkennungssoftware. Der Vorteil ist hier eindeutig, dass diese Dienste als serverlose Realisierung besser skalieren und ein besseres Kostenmanagement haben können [Roberts and Chapin, 2017].
Eine andere spannende Entwicklung ist Lambda@Edge von AWS. Dabei geht es darum Leistung zu verbessern und Latenzen zu verringern, indem Regionsgrenzen aufgehoben werden. Die Anwendung wird dann global, immer in der Nähe des Endnutzers, ausgeführt, was die erhöhte Leistung schafft [AWS, k]. Normalerweise muss eine Region gewählt werden, in der die Funktionen ausgeführt werden und diese unterscheiden sich teilweise stark in Kosten und Leistungen.
Eine weitere Innovation ist Serverless für Container, wie AWS Fargate. Dabei sollen die Vorteile von Serverless – wie automatisches Skalieren, kein Server Management und das Pay-as-you-go Prinzip – erhalten bleiben, aber anstelle von Funktionen werden Container eingesetzt. Dies ermöglicht langlaufende Prozesse (kein 15 Min. Limit) und hebt damit Grenzen auf. Das Training von Machine Learning Algorithmen wäre damit ein möglicher Use Case, während er in dieser Arbeit als Grenzfall beschrieben wurde [AWS, j].
Zuletzt sei eine Abstraktionsebene von Serverless bzw. Cloudtechnologien genannt, die einen Teil von Roberts' Vision umzusetzen versucht: Amazons Elastic Beanstalk. Dies ist ein benutzerfreundlicher Service zum Bereitstellen und Skalieren von Webanwendungen und -Services. Code kann hochgeladen werden und wird dann automatisch in allen Bereichen gemanaged. Trotzdem bleibt vollständige Kontrolle über die AWS-Ressourcen [AWS, e]. Während Beanstalk ein PaaS Produkt ist, kann so auch eine mögliche Zukunft für Serverless aussehen.

8 Fazit

Das Hauptziel dieser Arbeit war es, einen realistischen Blick auf das noch junge Serverless Computing zu bekommen. Die Fragen, die sich stellten waren, ob Serverless eine Serverrevolution auslösen kann oder schon hat, oder ob es lediglich "overhyped" ist. Dazu wurde in Kapitel 4 – The Good – über die Vorzüge und sinnvollen Einsatzzwecke von aktuellem Serverless geschrieben. Zusätzlich wurde die Umsetzung der angestrebten Anwendung dargestellt und die Testvorbereitung für die folgenden praktischen Experimente beschrieben. Zudem findet sich dort ein Gesamtüberblick über alle Ergebnisse.
Kapitel 5 und 6 – The Bad und The Ugly – haben die negativen Aspekte behandelt. Dabei ging es um Herausforderungen, aber auch um Chancen. Von Problemen ging es bis hin zu aktuellen und permanenten Grenzen der Architektur. Dazu kommen insgesamt sieben Versuche, welche die Fähigkeiten von Serverless untersuchen und evaluieren. Das vorletzte Kapitel, die Diskussion, hat sowohl die theoretischen als auch die praktischen Ergebnisse diskutiert und mögliche Zukunftsvisionen für Serverless und die entwickelte Anwendung dargestellt.

Die wichtigsten Erkenntnisse waren, dass sowohl den positiv gestimmten, also auch den kritischen Positionen gegenüber Serverless Wahrheit innewohnt. Eine komplexe, flexible Architektur einzuschätzen, bedarf immer mehrerer Seiten und durchläuft Phasen, die für unterschiedliche Bereiche unterschiedlich weit fortgeschritten sein können. Serverless ist in einigen Punkten überbewertet, aber hat wenige unüberwindbare Grenzen. Die größten Einschränkungen sind nach den Erkenntnissen dieser Arbeit: Latenzen, die Limitierungen durch die Provider und die Unausgereiftheit in vielen Bereichen. Gerade aus diesem Grund muss Serverless als Entwicklung betrachtet werden und nicht als Ziel. Es ist organisch im Zuge einer Evolution der Cloud entstanden, und es ist möglich, dass Cloudtechnologien das klassische Client-Server Modell in Zukunft ablösen. Ob diese Technologien noch als Serverless bezeichnet werden oder nicht spielt keine Rolle und es bleibt spannend die Entwicklung der Cloud zu beobachten.

Abschließend lässt sich feststellen, dass eine realistische, ausführliche Einschätzung von Serverless vorgenommen werden konnte. Der theoretische Teil ist zwar vorwiegend negativ, beinhaltet aber nicht viele strenge Grenzen, sondern eher Herausforderungen. Der praktische Anteil geht ebenfalls auf viele Aspekte ein, war allerdings stark durch die kostenlosen Kontingente und zeitlichen Möglichkeiten beschränkt. Viele Ideen mussten abgeändert oder verworfen werden, was natürlich auch an mangelnder Erfahrung lag. Dies zieht sich über die Experimente an sich und auch die Test- und Auswertungsmechanismen. In einigen Bereichen besteht damit noch Potential für weitere Versuche und Technologien, welche zeitlich oder auf Grund von Fehleinschätzungen nicht erschöpft werden konnte. Trotzdem bot gerade der praktische Teil eine enorme Lernkurve, da fast jede Technologie neu war. Eine weiterer Kritikpunkt sind implementierte Teile der Anwendung, die letztendlich nicht zu den Ergebnissen beigetragen haben, wie das Frontend oder die Nutzerauthentifikation. Diese Punkte waren teilweise viel Aufwand und haben Zeit von relevanteren Themen genommen. Allerdings waren sie für den Lernerfolg und die Vollständigkeit der Anwendung trotzdem sinnvoll.

Ausblick

Sowohl die entwickelte Anwendung als auch Serverless an sich haben viel Potential, aber auch Schwächen. Unter 7.2 wurden für die praktische Umsetzung bereits einige Aspekte genannt, die weiterhin spannend zu untersuchen wären. Gerade ohne die Limitierung durch kostenlose Kontingente und mit dem jetzt vorhandenen Grundwissen, wäre es spannend, die neuen Entwicklungen im Serverless-Realm zu begleiten und weitere Technologien zu testen.
Serverless selbst verspricht weiterhin starke Entwicklung und kann noch revolutionäre Züge annehmen. Trotzdem bestehen momentan zu viele Einschränkungen und es ist gut möglich, dass Serverless noch vor dem Tief des Hype Cycles steht. Damit kann vor dem Level der Produktivität der Ruf von Serverless weiter leiden, wenn diesen Verläufen Glauben geschenkt wird. Trotzdem kann aus dieser Arbeit geschlossen werden, dass Serverless und FaaS eine Zukunft haben. Zumindest im Kleinen wurden bereits jetzt revolutionäre Fortschritte gemacht, welche sich früher oder später weiter verstärken werden, auch wenn das klassische Client-Server Modell bisher nicht von Serverless abgelöst werden konnte.

Literaturverzeichnis

Db-engines ranking. DB-Engines, 2021. https://db-engines.com/en/ranking Accessed: 27.09.21.

A-Cloud-Guru. The serverless approach to testing is different and may actually be easier. A Cloud Guru, 2017. https://acloudguru.com/blog/engineering/testing-and-the-serverless-approach Accessed: 27.09.21.

Artillery. Artillery pro pricing. https://artillery.io/pro/. Accessed: 12.08.21.

AWS. Aws lambda now supports up to 10 gb of memory and 6 vcpu cores for lambda functions. https://aws.amazon.com/de/about-aws/whats-new/2020/12/aws-lambda-supports-10gb-memory-6-vcpu-cores-lambda-functions, a. Accessed: 20.09.21.

AWS. Amazon api gateway. https://aws.amazon.com/de/api-gateway, b. Accessed: 09.06.21.

AWS. Kostenloses kontingent für aws. https://aws.amazon.com/de/free, c. Accessed: 11.06.21.

AWS. Aws lambda—the basics. https://docs.aws.amazon.com/whitepapers/latest/serverless-architectures-lambda/aws-lambdathe-basics.html, d. Accessed: 03.06.21.

AWS. Aws elastic beanstalk. https://aws.amazon.com/de/elasticbeanstalk/, e. Accessed: 22.10.21.

AWS. Aws cloud development kit. https://aws.amazon.com/de/cdk/, f. Accessed: 20.07.21.

AWS. Amazon cognito. https://aws.amazon.com/de/cognito, g. Accessed: 09.06.21.

AWS. Amazon dynamodb. https://aws.amazon.com/de/dynamodb, h. Accessed: 09.06.21.

AWS. Read/write capacity mode. https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/I
i. Accessed: 20.08.21.

AWS. Aws fargate. https://aws.amazon.com/fargate/, j. Accessed: 02.10.21.

AWS. Lambda@edge. https://aws.amazon.com/lambda/edge/, k. Accessed: 22.10.21.

AWS. Aws lambda. https://aws.amazon.com/de/lambda/, l. Accessed: 08.06.21.

AWS. Aws lambda – preise. https://aws.amazon.com/de/lambda/pricing/, m. Accessed: 13.10.21.

AWS. Amazon s3. https://aws.amazon.com/de/s3, n. Accessed: 09.06.21.

AWS. Amazon web services announces aws lambda. https://press.aboutamazon.com/news-releases/news-release-details/amazon-web-services-announces-aws-lambda, 2014. Accessed: 11.05.21.

AWS. Serverless architectures with aws lambda. Technical report, Amazon Web Services, Inc., 11 2021a.

AWS. Security overview of aws lambda aws whitepaper. Technical report, Amazon Web Services, Inc., 02 2021b.

Ioana Baldini, Paul Castro, Kerry Chang, Perry Cheng, Stephen Fink, Vatche Ishakian, Nick Mitchell, Vinod Muthusamy, Rodric Rabbah, Aleksander Slominski, and Philippe Suter. Serverless computing: Current trends and open problems, 2017.

James Beswick. Understanding database options for your serverless web applications. AWS Compute Blog, 2020. https://aws.amazon.com/de/blogs/compute/understanding-database-options-for-your-serverless-web-applications/, Accessed: 13.09.21.

James Beswick. Operating lambda: Anti-patterns in event-driven architectures – part 3. AWS Compute Blog, 2021. https://aws.amazon.com/de/blogs/compute/operating-lambda-anti-patterns-in-event-driven-architectures-part-3/, Accessed: 24.06.21.

Bernard Brode. Why the serverless revolution has stalled. *InfoQ*, 2020. https://www.infoq.com/articles/serverless-stalled/, Accessed: 21.09.21.

Christoph Burnicki. Cloud computing and carbon footprint. INNOQ Blog, 2020. https://www.innoq.com/en/blog/cloud-computing-and-carbon-footprint Accessed: 24.06.21.

Paul Castro, Vatche Ishakian, Vinod Muthusamy, and Aleksander Slominski. The rise of serverless computing. *Commun. ACM*, 62(12):44–54, November 2019. ISSN 0001-0782. doi: 10.1145/3368454. URL https://doi.org/10.1145/3368454.

CheckPoint. What is serverless security? Check Point Blog. https://www.checkpoint.com/cyber-hub/cloud-security/what-is-serverless-security/ Accessed: 21.10.21.

Toby Coppel. Serverless is more. LinkedIn, 2019. https://www.linkedin.com/pulse/serverless-more-toby-coppel Accessed: 30.08.21.

Datadog. The state of serverless. Technical report, Datadog, 2021.

Alex DeBrie. Choosing a database for serverless applications. Serverless Blog, 2019. https://www.serverless.com/blog/choosing-a-database-with-serverless Accessed: 15.09.21.

Entreprogrammer. Firebase almost ruined my startup. Medium, 2021. https://betterprogramming.pub/firebase-almost-ruined-my-startup-4384b29979fd Accessed: 25.08.21.

N. Fee. What is serverless architecture? key benefits and limitations. New Relic, 2020. https://www.serverless.com/blog/serverless-architecture-code-patterns Accessed: 19.08.21.

Serverless Framework. Serverless manifesto. https://www.serverless.com/learn/manifesto/, a. Accessed: 11.06.21.

Serverless Framework. Use cases. https://www.serverless.com/learn/use-cases/, b. Accessed: 19.08.21.

Uwe Friedrichsen. The cloud ready fallacy. Uwe Friedrichsen Blog, 2021a. https://www.ufried.com/blog/cloud_ready_fallacy Accessed: 25.08.21.

Uwe Friedrichsen. The cloud ready fallacy. Uwe Friedrichsen Blog, 2021b. hhttps://www.ufried.com/blog/microservices_fallacy_11_recommendations/ Accessed: 25.08.21.

Bennett Garner. Aws lambda tutorial: Is serverless worth the hype? Medium, 2018. https://betterprogramming.pub/is-serverless-worth-the-hype-9bcb1842678b Accessed: 25.08.21.

Google. Kostenloses programm von google cloud. https://cloud.google.com/free/docs/gcp-free-tier?hl=de. Accessed: 11.06.21.

Eslam Hefnawy. Serverless code patterns. Serverless Framework Blog, 2017. https://newrelic.com/blog/best-practices/what-is-serverless-architecture Accessed: 19.08.21.

Joseph M. Hellerstein, Jose Faleiro, Joseph E. Gonzalez, Johann Schleier-Smith, Vikram Sreekanti, Alexey Tumanov, and Chenggang Wu. Serverless computing: One step forward, two steps back. *arXiv preprint arXiv:1812.03651*, 2018.

P. Johnston. Serverless best practices. *Medium*, 2018. https://pauldjohnston.medium.com/serverless-best-practices-b3c97d551535, Accessed: 01.07.21.

Eric Jonas, Johann Schleier-Smith, Vikram Sreekanti, Chia-Che Tsai, Anurag Khandelwal, Qifan Pu, Vaishaal Shankar, Joao Menezes Carreira, Karl Krauth, Neeraja Yadwadkar, Joseph Gonzalez, Raluca Ada Popa, Ion Stoica, and David A. Patterson. Cloud programming simplified: A berkeley view on serverless computing. Technical report, EECS Department, University of California, Berkeley, Feb 2019. URL http://www2.eecs.berkeley.edu/Pubs/TechRpts/2019/EECS-2019-3.html.

Nick Martin. Don't overestimate the benefits of serverless computing. *TechTarget*, 2018. https://searchaws.techtarget.com/opinion/Dont-overestimate-the-benefits-of-serverless-computing , Accessed: 24.09.21.

T. Grance P. Mell. The nist definition of cloud computing. https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-145.pdf, 2011. Accessed: 20.05.21.

UX Planet. How page speed affects web user experience. https://uxplanet.org/how-page-speed-affects-web-user-experience-83b6d6b1d7d7, 2019. Accessed: 12.10.21.

Mariliis Retter. Serverless case study – netflix. https://dashbird.io/blog/serverless-case-study-netflix/, 2020. Accessed: 23.08.21.

M. Roberts. Serverless architectures. *martinFowler*, 2018. https://martinfowler.com/articles/serverless.html, Accessed: 17.09.21.

M. Roberts and J. Chapin. *What Is Serverless?* O'Reilly Media, Inc., 2017. ISBN 9781491984161. URL https://www.oreilly.com/library/view/what-is-serverless/9781491984178/.

Chris Guzikowski Roger Magoulas. O'reilly serverless survey 2019: Concerns, what works, and what to expect. O'Reilly Media, Inc, 2019. https://www.oreilly.com/radar/oreilly-serverless-survey-2019-concerns-what-works-and-what-to-expect Accessed: 24.06.21.

Hrsg. S. Reinheimer. *Cloud Computing: Status quo, aktuelle Entwicklungen und Herausforderungen*, chapter 1. Springer Vieweg, 2018.

P. Sbarski. *Serverless Architectures on AWS*. Manning Publications Co., 2017. ISBN 9781617293825. URL https://www.manning.com/books/serverless-architectures-on-aws.

André Schaffer. Testing of microservices. Spotify AB, 2018. https://engineering.atspotify.com/2018/01/11/testing-of-microservices/ Accessed: 03.10.21.

Ory Segal and Simon Whittaker. The ten most critical risks for serverless applications v1.0. GitHub, 2019. https://github.com/puresec/sas-top-10 Accessed: 21.10.21.

Amiram Shachar. The hidden costs of serverless. Medium, 2018. https://medium.com/@amiram_26122/the-hidden-costs-of-serverless-6ced7844780b Accessed: 17.09.21.

Nitzan Shapira. How to test serverless applications. Epsagon, 2019. https://epsagon.com/development/how-to-test-serverless-apps/ Accessed: 27.09.21.

K. Stalcup. Aws vs azure vs google cloud market share 2021: What the latest data shows. https://www.parkmycloud.com/blog/aws-vs-azure-vs-google-cloud-market-share/, 2021. Accessed: 24.06.21.

Bogdan Sucaciu. Serverless code patterns. The Overflow, 2020. https://stackoverflow.blog/2020/03/16/how-event-driven-architecture-solves-modern-web-app-problems/ Accessed: 16.10.21.

Thinkwik. Do you know the importance of ui/ux development? Medium, 2018. https://medium.com/@thinkwik/do-you-know-the-importance-of-ui-ux-development-773eae38436e Accessed: 24.08.21.

Serhii Titienok. Lessons learned from building a low-latency system based on aws lambda. LOHIKA, 2020. https://www.lohika.com/aws-lambda-lessons-learned Accessed: 15.09.21.

Phil Vuollet. Serverless databases: The good, the bad, and the ugly. ADDO, 2020. https://www.alldaydevops.com/blog/serverless-databases-the-good-the-bad-and-the-ugly Accessed: 16.09.21.

Chenggang Wu. The state of serverless computing. RISE Lab UC Berkeley, 2019. https://www.infoq.com/presentations/state-serverless-computing/ Accessed: 02.10.21.

Erklärung zur selbstständigen Bearbeitung einer Abschlussarbeit

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

Ort	Datum	Unterschrift im Original