Fakultät 07 Wirtschaftsinformatik

Charakteristiken einer Microservice-Architektur

Seminararbeit
Aktuelle Technologien zur Entwicklung verteilter Java Anwendungen

Eingereicht von
Patrick Rogg
im Sommersemester 2019

Betreuer:
1. Dipl. Inform. M. Theis

INHALTSVERZEICHNIS

1 EINLEITUNG

Die IT-Branche entwickelt sich in rasanter Geschwindigkeit weiter. Was vor kurzem noch aktuell war ist binnen ein bis zwei Jahre meist schon wieder veraltet. Um der rasanten Entwicklung folgen zu können benötigen Unternehmen Flexibilität, Anpassungsfähigkeit, sowie die Möglichkeit ihre Produkte schnell zu entwickeln und auf den Markt bringen zu können. Diesen Anforderungen sind große monolithischen Anwendungen auf lange Sicht nicht mehr gewachsen. Deshalb wollen immer mehr Unternehmen ihre monolithischen Applikationen in kleine voneinander unabhängige Microservices aufteilen. Durch den Erfolg vieler Unternehmen bei diesem Schritt haben Microservices in den letzten Jahren immer mehr an Bedeutung gewonnen und für viele Unternehmen kommt eine monolithische Architektur gar nicht mehr in Frage.
Kurz gesagt sind Microservices kleine autonome Anwendungen, die unabhängig voneinander bereitgestellt werden können. Die einzelnen Services kommunizieren dabei über standardisierte Schnittstellen wie beispielsweise REST miteinander. Durch die Unabhängigkeit der einzelnen Microservices kann die Technologieauswahl individuell für jeden Service erfolgen.

1.1 Zielsetzung

Der Begriff Microservices ist noch relativ jung und viele Softwareentwickler kennen die genaue Bedeutung und deren Eigenschaften nicht. In dieser Arbeit soll deshalb ein grundlegendes Verständnis über das Themengebiet Microservices und deren Architektur geschaffen werden. Die Themengebiete werden dabei so beschrieben, dass diese möglichst allgemeingültig sind. Darum wurde weitestgehend darauf verzichtet konkret auf Technologien einzugehen mit denen Microservices umgesetzt werden.

1.2 Aufbau der Arbeit

Zu Beginn der Arbeit werden die grundlegenden Eigenschaften eines Monolithen und von Microservices definiert.
Um einen besseren Eindruck davon zu erhalten, was Microservices ausmacht und welche Eigenschaften diese besitzen werden im folgenden Kapitel typische Charakteristiken von Microservices vorgestellt.
Viele klassische monolithische Anwendungen sollen in Microservices aufgeteilt werden. Welche Voraussetzungen und Bedingungen dafür zunächst erfüllt sein müssen werden im vierten Kapitel dargestellt.
Die Microservice Architektur ist aber keine Lösung für alle Probleme und bringt seine eigenen Herausforderungen mit. Deshalb beschreibt das fünfte Kapitel Schwierigkeiten und Problematiken, welche durch die Verwendung von Microservices entstehen können.
Zum Abschluss der Arbeit wird eine Zusammenfassung gegeben und das Fazit gezogen.

2 GRUNDLAGEN

In diesem Abschnitt werden die Grundlagen vermittelt, welche zum Verständnis dieser Arbeit notwendig sind. Zu Beginn wird die Microservice Architektur definiert. Anschließend wird auf die monolithische sowie die Microservice Architektur eingegangen.

2.1 Definition

Microservices sind ein Architekturstil bei dem die Applikation in kleine, voneinander unabhängige, autonome Komponenten aufgeteilt werden. Diese Komponenten werden als Services bezeichnet und sind darauf spezialisiert eine bestimmte Aufgabe im System zu erfüllen. Ein Microservice besitzt dabei standardisierte Schnittstellen, über welche er mit den anderen Services im System kommuniziert und ist dadurch nicht an bestimmte Technologien gebunden. Jeder Service kann dabei durch automatisierte Bereitstellungsketten und unabhängig von anderen Services zur Verfügung gestellt werden.

2.2 Monolithische Architektur

Der monolithische Ansatz gilt als das Architekturmuster nach dem standardmäßig Applikationen entwickelt werden. Eine Architektur nach monolithischem Muster lässt sich vor allem dadurch beschreiben, dass sich der gesamte Quellcode der Applikation in einer Codebasis befindet zu welcher alle Entwickler am System beitragen. Zudem entsteht beim Bauprozess einer monolithischen Anwendung ein ausführbares Artefakt in dem alles enthalten ist, um die komplette Anwendung auszuführen. Weitere Instanzen des Monolithen können einfach durch Kopien des Artefakts auf weiteren Servern bereitgestellt werden, weshalb eine monolithische Architektur sich gut horizontal skalieren lässt. Durch vorschalten eines Load Balancers kann die Last der Useranfragen auf die laufenden Artefakte auf den Servern verteilt werden.

In Abbildung 1 wird beispielhaft eine E-Commerce Anwendung als Monolith dargestellt. Im darauffolgenden Kapitel 1.3 Microservice Architektur ist die gleiche Anwendung abgebildet, aber mit Microservices als Architekturkonzept.

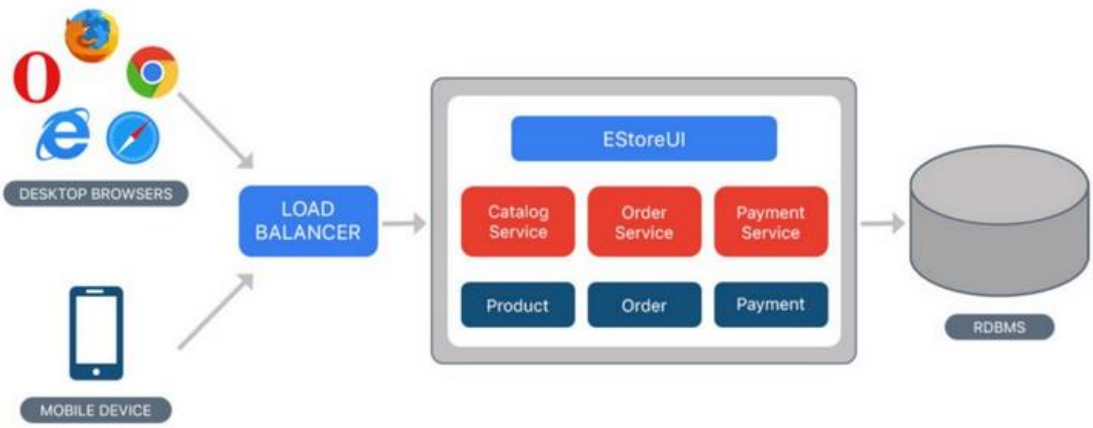

Abbildung 1: Monolithische Architektur für eine E-Commerce Applikation
Quelle: (Haq, 2019)

Wie bereits angesprochen besitzt ein Monolith eine Codebase. In dieser befinden sich das UI, die Businesslogik und die Data Access Layer um auf die Datenbank zuzugreifen. Für die Datenbank wird üblicherweise eine große relationale Datenbank verwendet.

2.3 Microservice Architektur

Microservices sind ein Ansatz zur Entwicklung von Software in dem eine große Applikation in kleine modulare Services aufgeteilt wird. Modular bedeutet, dass die Services lose voneinander gekoppelt sind und innerhalb eines jeden Microservice eine hohe Kohäsion vorzufinden ist. Jedes Modul ist dabei für einen bestimmen Geschäftsbereich in der Applikation verantwortlich und über den Daumen gepeilt soll ein Microservice etwa so groß sein, dass dieser innerhalb von zwei Wochen neu implementiert werden kann (Newman, 2015). Durch die relativ kleine Größe von den Services können diese folglich einfacher ersetzt und ausgetauscht werden und somit wird die Wartbarkeit eines Systems verbessert. Außerdem besitzt im Gegensatz zu einem Monolithen jeder Microservice seine eigene Datenbank und nur der entsprechende Microservice hat direkten Zugriff auf diese Datenbank. Damit andere Services trotzdem die Möglichkeit haben die sogenannten CRUD-Operationen (**C**reate, **R**ead, **U**pdate, **D**elete) in der Datenbank durchzuführen stellt jeder Service hierfür entsprechende Schnittstellen zur Verfügung. Die Kommunikation zwischen den Services erfolgt meist über REST-Schnittstellen, welche als HTTP Endpunkte implementiert werden. Das Deployment eines jeden Service in einer Microservice Architektur erfolgt unabhängig vom Rest des Systems. Deshalb können neue Features und auch Bugfixes schnell in das System integriert werden ohne dabei die gesamte Applikation neu bauen und bereitstellen zu müssen.

In Abbildung 2 wird nun erneut eine E-Commerce Applikation dargestellt, wie aber bereits angesprochen dieses Mal nach dem Microservice Architekturmuster.

Abbildung 2: Microservice Architektur für eine E-Commerce Applikation
Quelle: (Haq, 2019)

Zunächst fällt bei der Betrachtung der Abbildung auf, dass das es nicht mehr eine große Applikation gibt (welche vorher beim Monolithen als ein großer grauer Kasten dargestellt war), sondern jeder Service ist für eine bestimmte Funktionalität innerhalb der Anwendung zuständig. Die Benutzer des Systems rufen zunächst den Online Shopping Service auf. Anschließend leitet dieser die Anfragen an die entsprechenden Microservices weiter, welche daraufhin die angefragten Daten zurückliefern. Ein weiterer Punkt der ins Auge springt ist, dass jeder Microservice seine eigene Datenbank besitzt und nicht wie zuvor beim Monolithen eine relationale Datenbank alle Daten des Systems verwaltet.

3 CHARAKTERISTIKEN VON MICROSERVICES

Martin Fowler, James Lewis und Sam Newman haben Microservices Charakteristiken zugewiesen, welche diese besitzen sollten. Dabei muss nicht jede Microservice Architektur alle der folgenden Eigenschaften aufweisen, sondern nur die meisten davon. Im folgenden Kapitel werden diese Charakteristiken beschrieben und einem monolithischen Architekturstil gegenübergestellt.

3.1 Services anstatt Komponenten

Software in einzelne Komponenten aufzuteilen ist ein schon lange definiertes Paradigma der Software Entwicklung. Deshalb wird auch in Monolithen die Software in Komponenten und Module aufgeteilt. Diese Komponenten sind durch definierte Schnittstellen auch einzeln austauschbar und veränderbar, aber befinden sich immer noch alle in einer Applikation. Dies hat aber zur Folge, dass die gesamte Anwendung neu gebaut und bereitgestellt werden muss, selbst wenn nur kleine Änderung am Quellcode vorgenommen wird.
Durch die Realisierung von Komponenten als eigene unabhängige Services entsteht der Vorteil, dass nicht mehr die gesamte Applikation neu bereitgestellt werden muss, sondern nur der Microservice an dem Änderungen vorgenommen wurden muss neu gebaut, getestet und bereitgestellt werden.
Eine weitere Folge einer Microservice Architektur ist, dass die Schnittstellen weitaus expliziter werden und die Kopplung zwischen den Komponenten somit gezwungenermaßen gering bleibt. Manchmal sind Entwickler unaufmerksam oder bequem und verursachen durch Methodenaufrufe anderer Komponenten eine Kopplungen im System. Da Services strikt voneinander getrennt sind und nur über entfernte Schnittstellen aufgerufen werden können, sind Kopplungen einfacher zu vermeiden.

3.2 Teamaufteilung nach Geschäftsbereich

Conway´s Law besagt, dass jede Organisation die ein System entwickelt ein Produkt entwickelt, dessen Struktur eine Kopie der Kommunikationsstruktur in der Organisation darstellt. Klassischerweise werden die Personen in einem Softwareentwicklungsunternehmen nach Technologie eingeteilt. So gibt es häufig ein Team, welches für das Frontend und Design von Applikationen zuständig ist. Ein anderes Team implementiert immer die Geschäftslogik und zuletzt noch die Einheit, welche sich um die Datenbank kümmert.
Eine teamübergreifende Kommunikation stellt sich immer deutlich schwerer dar als die Kommunikation innerhalb eines Teams. Deshalb gestaltet sich der Entwurf von Schnittstellen schwerer, falls die Teams nach Technologie strukturiert werden. Wenn beispielsweise das Backendteam nicht direkt mit den Konsumenten des API kommunizieren kann, da diese räumlich voneinander getrennt sind, können dabei Schnittstellen entstehen, welche nicht optimal auf den Konsumenten zugeschnitten sind. Zudem entstehen in solchen Strukturen teamübergreifende Abhängigkeiten. Dies ist genau wie bei Software zu vermeiden und sorgt für zeitliche Verzögerungen sowie Frustration in der Entwicklung der Software.
Microservices werden nach Geschäftsbereichen aufgeteilt. Dabei gibt es zum Beispiel einen Service der für die Benutzerverwaltung zuständig ist und ein anderer der die Produkte bereitstellt, welche auf der Website angeboten werden. Jeder Service ist dabei eigenständig

dafür verantwortlich, dass ein entsprechendes User-Interface bereitgestellt wird, die entsprechende Geschäftslogik implementiert ist und die Daten korrekt persistiert werden. Darum müssen in einer Microservice Architektur gezwungenermaßen andere Teamstrukturen gewählt werden. Jedes Team braucht nun Personen, welche sich auf den entsprechenden Teilgebieten auskennen. Die Teams stellen sich so nun hauptsächlich innerhalb des Teams Schnittstellen zur Verfügung und können den Bedarf an diese wesentlich besser abstimmen.

3.3 Ein Service ist Eigentum eines Teams

In der Softwareentwicklung wird klassisch projektbasiert vorgegangen und das Projekt in verschiedene Phasen unterteilt. Dabei wird jede Projektphase von einem anderen Team durchgeführt und nach Fertigstellung wird der Software wird das Entwicklungsteam meistens wieder aufgelöst und oder neuen Aufgaben zugeteilt. Die Unterteilung der Projektphasen wird in der folgenden Abbildung dargestellt.

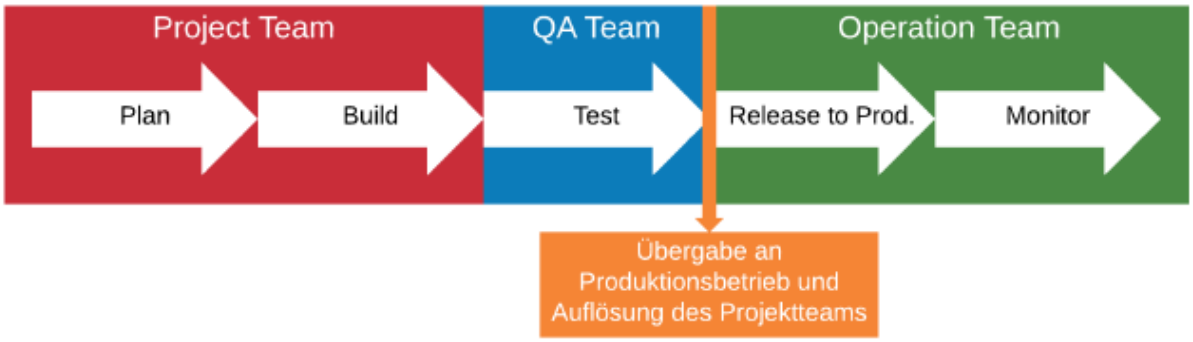

Abbildung 3: Klassischer Projektablauf

Bei der Entwicklung von Microservices gilt es ein solches Vorgehen zu vermeiden. Anstatt einen Service in die Hände von anderen zu geben, ist das Team, welchem der Service zugeteilt worden ist ebenso für die Planung, das Bauen, das Testen sowie für die Veröffentlichung, Überwachung und möglicherweise sogar den Kundensupport des Service zuständig.
Dieses Prinzip wurde von Amazon entwickelt. Dort gilt das Sprichwort „You build it, you run it“ (Vogels, 2019). Dadurch kommen die Entwickler direkt mit dem Endnutzer ihrer Software in Kontakt. Sie können sehen wie ihre Software benutzt wird und an welchen Stellen häufig Probleme auftreten, je nach dem zu welchen Themen dem Kundensupport häufig Anfragen gestellt werden. Diese gewonnen Informationen können somit in den weiteren Entwicklungsprozess des Services einfließen.
Das Prinzip der Informationsweitergabe besteht ebenso bei klassischem Vorgehen. Das Problem dabei ist zum einen, dass vielleicht das Projektteam welches für diese Komponente zuständig war so schon gar nicht mehr in dieser Konstellation besteht. Zum anderen gehen in einer Kommunikationskette häufig Informationen verloren, weil sie nicht als entscheidend aufgefasst werden oder schlichtweg nicht weitergereicht werden und können somit nicht in die Entwicklung der Software einfließen. Außerdem ist bei einer monolithischen Architektur die fachliche Kontextabgrenzung meist nicht so eindeutig abgrenzbar wie bei Microservices und deshalb eine Zuteilung von Kundensupport auf Teams nicht ohne weiteres möglich.

3.4 Dezentralisierte Führung

Zentrale Führung heißt, dass alle Entscheidungen über den Schreibtisch einer Person fließen und ohne diese nichts entschieden werden kann. Das hat in Entscheidungen rund um die Software in häufigen Fällen zur Folge, dass ein Technologiestack zu Beginn des Projektes vom Projektleiter festgelegt wird und dieser anschließend in Stein gemeißelt ist. Der Projektleiter hat die Technologien ausgewählt, wieso sollte er später seine eigene Entscheidung wieder anzweifeln?
Durch solche Strukturen innerhalb eines Unternehmens werden die Entwickler enorm in ihren Entscheidungen eingeschränkt und es fehlt ein Spielraum für Innovationen und neue Ideen. Mit Microservices besteht aber die Möglichkeit mehrere Technologiestacks in einem Projekt zu verwenden und deshalb sollte die Verwendung einer anderen Technologie immer in Betracht gezogen werden. Natürlich ist es wenig sinnvoll für jeden Service andere Technologien auszuwählen und der Technologiestack für die Microservices sollte trotzdem weitestgehend standardisiert sein. Allerdings tritt in einem Softwareprojekt oftmals der Fall ein, dass für bestimmte Anforderungen und Features ein anderes Framework, eine andere Programmiersprache oder eine andere Datenbank (SQL vs. NoSQL) wesentlich besser geeignet wären, als die anfangs Ausgewählte.
Aus diesem Grund wird dezentrale Führung in Microservices oftmals bevorzugt. Das Team hat relative freie Wahl in der ihres Technologiestacks und lediglich die Anforderung an die Schnittstelle muss erfüllt sein. So können Teams die beste Technologie für ihr spezifisches Problem auswählen. Den Teams wird so die Möglichkeit geboten neue und moderne Technologien in ihrem Service auszuprobieren und es wird Platz für Innovation geschaffen. Außerdem besitzt ein Team welches sich ständig mit einem Microservices beschäftigt weitaus mehr Expertise über diese als ein Projektleiter, welche mehrere Microservices überblicken muss, weshalb das Team die besseren Entscheidungen für einen Service treffen kann.

3.5 Dezentralisiertes Datenmanagement

Klassische Systeme nutzen meist eine große Datenbank, um die Daten der Anwendungen zu persistieren. Nachteil daran ist die schlechte horizontale Skalierbarkeit von einer einzelnen großen relationalen Datenbank. Es muss immer die komplette Datenbank skaliert werden und es ist nicht möglich nur die eine Entität zu skalieren, welche an die Kapazitäten stößt.
Das viel gravierendere Problem ist aber, dass durch eine gemeinsame Datenbank wieder direkte Abhängigkeiten zur Datenbank entstehen. Wenn sich ein Datenfeld in einer Entität verändert kann es sein, dass Teile der Applikation nicht mehr funktionieren. Außerdem entsteht so das Problem, dass das komplette System so an eine bestimmte Datenbank-technologie gebunden ist.
In einer Microservice Architektur wird deswegen das Datenmanagement dezentralisiert. Jeder Service erhält seine eigene Datenbank. Somit kann für jeden Microservice die Technologie ausgewählt werden, welche für diese Form von Daten am besten geeignet ist. Weiterhin ist es wesentlich einfacher die Datenbank zu skalieren, da nun nur für den einen Service, welcher an die Kapazitätsgrenzen stößt skaliert werden muss und nicht die Datenbank des kompletten Systems was viel Overhead darstellen würde.

Microservices sollen eine hohe Kohäsion aufweisen und lose gekoppelt sein. Deshalb ist ein wichtiger Aspekt für das Datenmanagement in einer Microservice Architektur, dass auf die Daten einer Datenbank nur über den Service zugegriffen wird zu dem diese Datenbank gehört. Das heißt es können sämtliche Datenbankoperationen nur über die von diesem Service definierten Schnittstellen durchgeführt werden. So können Änderung am Datenschema zum Beispiel durch Versionierung der Schnittstellen gelöst werden. Das heißt anstatt die alte Schnittstelle direkt mit der neuen zu ersetzen, wird die alte Schnittstelle aktiv gelassen. Damit haben andere Teams welche diese Schnittstelle benutzen Zeit sich an die neue Schnittstelle anzupassen.

3.6 Automatisierung der Infrastruktur

In Kapitel zwei wurde definiert, dass Microservices kleine und eigenständige Einheiten sind, welche die Eigenschaft haben sollen, unabhängig voneinander bereitgestellt werden zu können. Bei einigen wenigen Services ist es vielleicht noch möglich die Bereitstellung dieser manuell durchzuführen. Allerdings haben Unternehmen teilweise mehrere hundert oder tausend Microservices, welche mehrfach skaliert werden. In solchen Fällen ist ein händisches Deployment schlichtweg unmöglich. Darum spielt die Automatisierung der Infrastruktur eine zentrale Rolle in der Microservice Architektur.
Neue Services oder Änderungen an alten Services sollen Kunden möglichst schnell zur Verfügung gestellt werden. Wichtig dabei ist aber, dass die neuen Features auch funktionieren. Um dies zu überprüfen müssen viele automatisierte Tests ausgeführt werden. Dazu werden Continuous Delivery und Continuous Integration verwendet. Eine beispielhafte Build-Pipeline ist in der folgenden Grafik dargestellt.

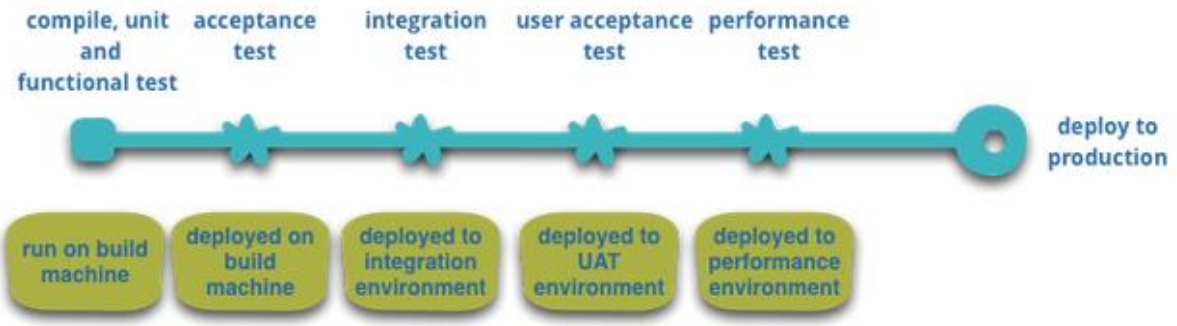

Abbildung 4: Grundlegende Build Pipeline
Quelle: (Fowler, 2019)

Durch die Automatisierung ist der Betrieb von Microservices einfacher, effizienter und risikoärmer. Jedoch ist der Schritt zur automatisierten Infrastruktur kein einfacher, auch wenn dies durch Continuous Integration Tools zunächst den Anschein macht. Wie bereits angesprochen können Services mit verschiedenen Technologien umgesetzt werden und diese benötigen unterschiedliche Konfigurationen in deren Infrastruktur. Damit erhöht sich die Komplexität innerhalb der Infrastruktur und erschwert die Einrichtung einer automatischen Infrastruktur. Der Schritt zur Automatisierung lohnt sich aber und ist bei größeren Systemen sogar unumgänglich.

3.7 Ausfälle isolieren

Eine Microservice Architektur muss so gestaltet werden, dass diese mit Ausfällen von Services umgehen kann. Jeder Microservice kann aufgrund verschiedenster Gründe plötzlich nicht mehr verfügbar sein und es wäre fatal, wenn durch den Ausfall eines Microservice das komplette System nicht mehr erreichbar ist und somit ein single point of failure entstehen würde.
Durch die höhere Anzahl an verwendeten Maschinen auf denen die einzelnen Services laufen erhöht sich auch dementsprechend das Risiko, dass eine Maschine ausfällt. Des Weiteren wird in einem Monolithen über Methodenaufrufe kommuniziert und nicht über Netzwerke. Auch diese Netzwerke können nicht mehr erreichbar sein.
Darum ist es wichtig sich vorab darüber im Klaren zu werden, dass zu irgendeinem Zeitpunkt in naher oder ferner Zukunft eine Maschine, das Netzwerk, die Software oder ein anderer Teil des Systems ausfallen wird. Ein guter erster Schritt ist es aus diesem Grund sich darüber Gedanken zu machen, welche Auswirkungen der Ausfall eines Service haben könnte. Daraufhin sollten diese Auswirkungen auf den restlichen Teil des Systems minimiert werden und der Ausfall des Service sozusagen isoliert werden, damit keine Kettenreaktion im System entstehen kann.
Beispielsweise können die Timeouts im System kürzer gestaltet werden, damit eine Anfrage allein nicht einen Service blockieren kann oder wenn bei der Überwachung der Anwendung festgestellt wird, dass ein Service viele Fehler bei der Bearbeitung von Anfragen produziert, kann ein automatischer Neustart dieses Services veranlasst werden.
Weiterhin können Circuit Breakers (zu Deutsch: Sicherung) implementiert werden, welche nach einer bestimmten Anzahl an fehlerhaften Anfragen an einen Service greift. Danach werden alle weiteren Anfragen an den Service sofort und automatisch mit einer Fehlermeldungen als Antwort abgebrochen. Währenddessen kann der fehlerhafte Microservice neugestartet werden oder sich von der Überbelastung erholen, da dieser keine weiteren Anfragen mehr beantworten muss. Nach einer gewissen Zeit können Anfragen an den Service geschickt werden, um zu überprüfen ob dieser wieder ordnungsgemäß funktioniert (Newman, 2015). Daraufhin wird der Circuit Breaker zurückgesetzt und der Service kann wieder Anfragen beantworten.
Circuit Breaker sind dabei ein Teil des Bulkhead (zu Deutsch: Trennwand) Pattern. „Das Pattern wird Bulkhead bezeichnet, weil es ähnlich funktioniert wie ein Schiffsrumpf der in einzelne Partitionen aufgeteilt wird. Falls ein Teil des Rumpfes beschädigt wird, kann die Türe zu diesem geschlossen werden und nur die beschädigte Partition füllt sich mit Wasser.“ (Masashi Narumoto, 2019). Der Circuit Breaker schließt dabei sozusagen die Tür zu einem fehlerhaften Microservice und schottet den restlichen Teil der Applikation davon ab, damit diese funktionsfähig bleibt und nicht mit dem einen Service untergeht.
Zudem spielen Monitoring und Logging für das Erkennen von Fehlern und Memory Leaks im System eine wichtige Rolle, können frühzeitig vor Ausfällen eines Services warnen und somit große Ausfälle vermeiden.

3.8 Hohe Überwachbarkeit des Systems

Das Thema Überwachung geht mit dem Punkt 2.7 Ausfälle isolieren einher, denn um wie angesprochen Fehler im System zu finden, müssen die Services entsprechend überwacht werden. Die Schwierigkeit in der Überwachung von Microservices liegt darin, dass die Log Files auf mehreren virtuellen Maschinen gespeichert sind. Die manuelle Zusammenführung oder das Auswerten dieser Daten wird ab einer gewissen Anzahl an Microservices ein Ding der Unmöglichkeit und somit fällt die Auswertung der gewonnenen Daten schwer.
„Anstatt dessen benutzt man zentralisierte Untersysteme, welche die Logs einsammeln und diese zentral Verfügbar macht. Ein Beispiel dafür ist [das Tool] **logstash**, das mehrere Logfileformate zusammenfügen kann und diese an darunter liegende Systeme für eine nähere Analyse übergeben kann.“ (Newman, 2015, p. 158). Diese Untersysteme können daraufhin mit ElasticSearch Tools wie **Kibana** die vorliegenden Daten mit Abfragen analysieren und die Informationen anhand von Grafiken oder Statistiken darstellen.
Um Fehler im System zu finden ist es außerdem hilfreich zu wissen, welche Microservices bei einer bestimmten Aktion im System aufgerufen werden, was bei großen und komplexen Systemen oft nicht immer leicht zu durchschauen ist. Darum ist es eine bewährte Vorgehensweise den Aktionen im System, welche durch beispielsweise einen Buttonklick ausgeführt werden, eine Aktions-Id zuzuweisen. Diese wird im Header jeder Anfrage mitgeschickt und anschließend in die Logs des Services mitaufgenommen. So kann bei Fehlern nach der Aktions-Id im Logfile gesucht werden und der Service relativ einfach ausfindig gemacht werden, welcher für den Fehler in der Ausführungskette verantwortlich ist.

3.9 Evolutionäres und anpassbares Design

Große Monolithen sind typischerweise Systeme, welche über einen längeren Zeitraum genutzt werden. Sie werden deshalb von den Architekten und Entwicklern so geplant, dass diese auch noch in mehreren Jahren genutzt werden können. Da durch eine monolithische Architektur automatisch mehr Abhängigkeiten im System und zwischen Teams entstehen, müssen Änderungen gut koordiniert werden, um allen Teams genügend Spielraum zu geben auf Änderungen von Schnittstellen reagieren zu können. Aus diesem Grund werden oft feste Release Zeiten vereinbart, an denen eine neue Version des Systems in Produktion gebracht werden soll.
Eine Microservice Architektur hat ganz andere Charakteristiken. Ziel ist es Anpassungen an der Software dem Kunden möglichst schnell an den Endnutzer weiter zu geben, damit diese sofort von neuen Implementierungen oder Fehlerbehebungen profitieren können. Zum Beispiel können temporäre Features mithilfe eines neuen Service an das System gekoppelt werden und einfach wieder abgeschaltet werden wenn diese nicht mehr benötigt werden.
Der evolutionäre Part an Microservice wird auch durch die „Try it out“-Auffassung erkennbar. Neue Ideen können schnell ausprobiert werden und eventuell als neuer Service realisiert werden. Wenn das Ergebnis nicht den Anforderungen entspricht, hat man vielleicht wenige Wochen verloren. Bei einem Monolithen sind solche experimentellen Versuche nicht so flexibel möglich, wodurch die Evolution des Systems gehemmt ist und weniger neuen Ideen in die Applikation einfließen.

4 VOM MONOLITHEN ZU MICROSERVICES

Mit der Zeit und neuen Anforderungen wächst die Größe und Komplexität eines Monolithen immer weiter. Dies führt irgendwann dazu, dass die Codebase selbst für erfahrende Entwickler unübersichtlich und schwer wartbar wird. Daraus folgt, dass Änderungen an der Software immer zeitaufwendiger und kostspieliger werden. Diesen Problemen können Microservices aufgrund ihrer Charakteristiken entgegenwirken. Allerdings ist die Microservice Architektur kein Allerheilmittel und bringt dabei auch ihre eigenen Schwierigkeiten mit sich. Aus diesem Grund müssen die Entscheider über die Architektur des Systems abwägen, ob Microservices überhaupt der richtige Schritt sind und ob die eigenen Probleme damit überhaupt gelöst werden können. Des Weiteren muss kalkuliert werden ob sich ein solcher Schritt auch finanziell lohnt. Im Folgenden wird eine Vorgehensweise vorgestellt, nach welcher das Vorhaben Monolith zu Microservices umgesetzt werden kann.

4.1 Voraussetzungen

Zuerst muss die Grundlage dafür geschaffen werden, dass Microservices überhaupt in das neue System eingebunden werden können. Deshalb muss zunächst dafür gesorgt werden, dass eine entsprechende Infrastruktur für die Microservice Architektur geschaffen wird. Dabei steht vor allem das Thema der Automatisierung im Mittelpunkt. Wie in den Charakteristiken bereits erwähnt können Microservices ab einer bestimmten Anzahl an Services nicht mehr manuell bereitgestellt und verwaltet werden. Darum muss die Deployment Pipeline muss entsprechend angepasst werden, um die Services unabhängig voneinander bauen, testen und bereitstellen zu können. Zudem benötigt das System die Fähigkeit, dass die auf verschiedene virtuelle Maschinen verteilten Services auch dementsprechend überwacht werden können. Außerdem sind andere Teamstrukturen im Vergleich zu einem Monolithen notwendig. Wie in den Charakteristiken bereits angesprochen soll jedes Team einen eigenen Service verwalten, damit dies möglich wird müssen Personen mit verschiedenen technischen Fähigkeiten in einem Team zusammengefasst werden.

Die Voraussetzungen müssen nicht final erfüllt sein, sondern es reicht wenn die Grundlegenden Anforderungen daran erfüllt sind. Zum Beispiel reicht es für den Anfang, wenn die Fehler eines Service registriert werden, aber es muss nicht gleich eine automatische neue Bereitstellung implementiert sein. Für alle Stakeholder ist ein Wechsel vom Monolithen zu Microservice ein Lernprozess und eine enorme Umstellung in der Arbeitsweise. Deshalb würde nie das Ziel einer Microservice Architektur erreicht werden, wenn von Beginn an Perfektion verlangt würde.
Eine weitere Voraussetzung ist, dass neue Anforderungen an die Anwendung, welche einen eigenen Geschäftsbereich abbilden, direkt als Microservices implementiert werden. Ansonsten entsteht ein ewiger Kreislauf. Am einen Ende entsteht ein Microservice und am anderen Ende wird der Monolith wieder durch neue Geschäftslogik vergrößert. Dies sollte vermieden werden, denn kein Entwickler macht gerne die gleiche Arbeit zweimal.

4.2 Vorbereitung des Monolithen

Nach Erfüllung der Vorbedingungen kann nun die Vorbereitung des Monolithen in Angriff genommen werden. Zunächst wird dafür das bestehende System analysiert. Im Zentrum dieser Analyse steht vor allem den zusammengehörigen high-level Kontext der Anwendung zu identifizieren. Es wird dabei nach dem Top-Down Prinzip vorgegangen und man arbeitet sich vom hohen Abstraktionslevel des Systems zum feinen Aufbau der Applikation nach unten. Daraufhin wird mit der Umstrukturierung des Monolithen begonnen. Als ersten werden Ordner erstellt, welche den Kontext der Applikation repräsentieren und dann wird der bestehende Source Code in diese Ordner verschoben. Mit modernen IDEs kann die Verschiebung von Code automatisch durch Refactoring erfolgen (Newman, 2015), ohne dabei die Funktionalität der Anwendung zu verändern.
Nachdem der Monolith umstrukturiert wurde, müssen die Abhängigkeiten zwischen den verschiedenen Kontexten näher betrachtet werden, welche beispielsweise durch die Datenbank oder durch Methodenaufrufe bestehen. Diese direkten Abhängigkeiten zwischen Kontext muss aufgebrochen werden, da sich jeder Geschäftsbereich in einem eigenen Service befinden soll. Um die Beziehungen innerhalb einer Applikation visuell darstellen zu können, „sind Tools wie **Structure 101**“ (Newman, 2015) hilfreich.

4.3 Aufteilung des Monolithen

Der Monolith ist nun nach Kontext strukturiert und die Abhängigkeiten im System sind bekannt. Aber an welchem Teil des Monolithen beginnt man mit dem Aufbrechen der Abhängigkeiten?

4.3.1 Wo fängt man an?

Um mit der Entwicklung von Microservices warm zu werden, kann zu Beginn ein Kontext aus dem Monolithen herausgezogen werden, welcher wenige Abhängigkeiten aufweist und sich sozusagen am Rande des Systems befindet. Dies kann so oft durch iteriert werden bis der Monolith vollständig in einen Microservice aufgeteilt worden ist.
Wenn die Entwicklungsteams eine gewisse Routine in der Implementierung von Microservices gewonnen haben, kann allerdings auch so vorgegangen werden, dass besonders starke Abhängigkeiten im System zuerst aufgebrochen werden, um möglichst früh im Umbau des Systems die Abhängigkeiten zu reduzieren (Zhamak, 2019).
Ein weiterer Punkt nachdem ausgewählt werden kann ist die Entkopplung von Kontext, welcher für das Unternehmen wichtig ist oder an dem oft Änderungen vorgenommen werden müssen und dieser somit als Microservice einfacher in der Entwicklung und schneller in Sachen Bereitstellung wäre.

4.3.2 Aufbrechen von Abhängigkeiten

Abhängigkeiten bestehen in einer monolithischen Applikation über Methodenaufrufe und in der Datenbank. Methodenaufrufe können meist durch REST-Schnittstellen ersetzt werden über welche die Services kommunizieren. Die größte Abhängigkeit in einem Monolithen ist aber die Datenbank, in der durch Fremdschlüssel, geteilte Daten und Tabellen von unterschiedlichen Geschäftsbereichen zusammenhängen. Diese Abhängigkeiten müssen

aufgebrochen werden, sonst würde die Eigenschaft verletzt werden, dass jeder Microservice eine eigene Datenbank besitzt.
Sam Newman empfiehlt zuerst das Datenbankschema zwischen dem Monolithen und dem Kontext aufzuteilen und anschließend den Kontext in einen Microservices zu separieren. Dieser Prozess ist in der folgenden Abbildung dargestellt.

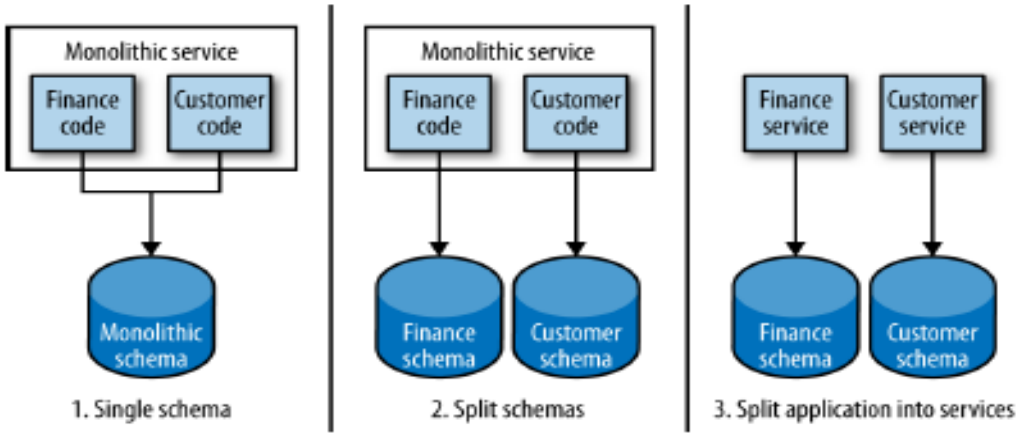

Abbildung 5: Stadien einer Serviceaufteilung
Quelle: (Newman, 2015, p. 89)

Vorteil von dieser Vorgehensweise ist, dass die Änderungen relativ einfach wieder rückgängig gemacht werden können, wenn Probleme der Aufteilung ersichtlich werden, indem einfach das alte Datenbankschema wieder benutzt wird. Wenn zuerst der Service aufgeteilt wird und anschließend die Datenbank werden Fehler in der Datenstruktur zu spät festgestellt und die Entwicklung des Service war möglicherweise umsonst.
Einen detaillierteren Einblick in das Thema der Neustrukturierung und die Aufteilung von Beziehungen in der Datenbank liefert das von Sam Newman empfohlene Buch *Refactoring Databases* von Scott j. Ambler und Pramod J. Sadalage.

5 HERAUSFORDERUNGEN VON MICROSERVICES

In den vorherigen Kapiteln wurden meist nur die positiven Aspekte von Microservices hervorgehoben. Um aber auch ein besseres Verständnis für die Herausforderungen und Schwierigkeiten einer Microservice Architektur zu erhalten, wird darauf in diesem Kapitel eingegangen.

5.1 Hohe Komplexität in der Entwicklung

Die Entwicklung von Microservices ist wesentlich anspruchsvoller für die Softwareentwickler. Diese können nicht einfach wie bei einem Monolithen eine Applikation auf dem Rechner laufen lassen, sondern die Entwickler müssen mit Microservices entweder mehrere Anwendungen gleichzeitig auf dem Rechner starten, was zu schnell einem Durcheinander führen kann oder auf die Microservices über eine entsprechende Testumgebung zugreifen. Damit entsteht allerdings eine Abhängigkeit zur Testumgebung und wenn diese nicht funktionsfähig ist können die Entwickler nicht weiter arbeiten. Desweiteren ist das Debuggen auf dem eigenen Rechner bei weitem einfacher, als die Logs auf einem entfernten System analysieren zu müssen. Somit gestaltet sich die Fehlerfindung für die Entwickler viel anspruchsvoller.

5.2 Risiko von schlechter Performance

Ein weiteres Risiko welches mit Microservices im Auge behalten werden muss, ist das Risiko einer schlechten Performance im Vergleich zu einem Monolithen. Jeder Microservices benötigt seine eigenen Ressourcen, weshalb ist die Last von verteilten Services auf die virtuellen Maschinen größer als dies bei einem Monolithen der Fall wäre. Des Weiteren haben Microservices durch die Kommunikation über Netzwerke eine weitere Komponente, welche eine schlechte Performance im System zur Folge haben kann.

5.3 Höhere Instandhaltungskosten

Wie bereits im vorherigen Punkt angesprochen benötigt eine Microservice Architektur mehr Ressourcen als eine monolithische Architektur. Dies hat zur Folge, dass mehr Server und Rechenleistung benötigt werden als bei einem Monolithen, um eine gleichartige Applikation bereitzustellen. Somit entstehen im Bereich der Instandhaltung im Vergleich mit einer monolithischen Anwendung höhere Kosten.

5.4 Komplexe Infrastruktur

Der Verwaltungsaufwand von Microservices ist bei weitem höher als bei einem Monolithen. Grund dafür ist die komplexe Infrastruktur, welche von Microservices gefordert werden. Durch die Verteilung von Services auf den virtuellen Maschinen ist es ab einer gewissen Anzahl an laufenden Instanzen nur schwer möglich herauszufinden, wo welche Instanz eines Service gestartet wurde. Somit kann der Debugprozess eine Suche nach der Nadel im Heuhaufen werden. Deshalb fordert eine Microservice Architektur Eigenschaften wie zentrales Monitoring, eine Strategie bei Serviceausfällen und die Automatisierung. Diese Anforderungen erfordert ein hohes Maß an Arbeit, welche nicht direkt in das zu entwickelnde

Produkt fließen und somit Kosten verursachen, welche bei einer monolithischen Anwendung nicht entstanden wären.

5.5 Hohe Anforderungen an die Entwickler

Wenn ein Entwickler neben der Implementierung auch noch die Bereitstellung und Verwaltung seines Services in der Produktion übernehmen soll, steigen natürlich die Anforderungen welche die Entwickler zu erfüllen haben. Jeder Entwickler muss die Deploymentpipeline verstehen und wissen die ein Container erstellt und bereitgestellt wird. Bei einem Monolithen gibt es die Rolle des Softwareentwicklers und des Systemadministratoren und jeder ist ein Experte auf seinem Gebiet. Natürlich kann diese Aufteilung auch in Microservices so vorgenommen werden, allerdings geht dann dem Entwicklerteam wiederum der direkte Kundenkontakt verloren, welcher zur Verbesserung des Services genutzt werden kann.

5.6 Unklare Kontextabgrenzung in der echten Welt

Microservices werden nach zusammengehörigem Kontext aufgeteilt. In der realen Welt ist eine solche Abgrenzung aber nicht immer eindeutig und es fällt somit schwer Geschäftslogik voneinander abzugrenzen. Wenn die Grenzen zwischen den Service nicht klar definiert sind, können die Services zwar in der Theorie unabhängig voneinander bereitgestellt werden. In der Praxis können die Abhängigkeiten zwischen Microservices aber so groß sein, dass diese unabhängig voneinander gar nicht funktionsfähig wären und somit immer gemeinsam mit einem anderen Service bereitgestellt werden müsste, womit der Vorteil einer unabhängigen Bereitstellung von Microservices verloren gehen würde.

6 ZUSAMMENFASSUNG UND FAZIT

6.1 Zusammenfassung

Eingangs der Arbeit wurden die grundlegenden Unterschiede der monolithischen und Microservice Architektur dargestellt.
Ein Monolith besitzt eine große Codebase in der sich der gesamte Quellcode befindet. Zudem entsteht beim Bauprozess der Applikation nur ein Artefakt indem sich die gesamte Applikation befindet. Eine Skalierung des Systems ist durch Kopieren des Artefakts und erneuter Bereitstellung auf weiteren Servern möglich. Eine weitere Eigenschaft der monolithischen Architekturen ist, dass diese meist eine große relationale Datenbank besitzen in der alle Daten der Anwendung gespeichert werden.
Bei Microservices besteht eine Applikation aus vielen kleinen und voneinander unabhängigen Services. Dadurch entsteht eine lose Kopplung der Applikation und die einzelnen Services können autonom bereitgestellt werden. Die Microservices kommunizieren dabei über standardisierte Schnittstellen wie zum Beispiel REST-Schnittstellen.

Daraufhin wurde auf die Charakteristiken einer Microservice Architektur eingegangen.
Ein Merkmal von Microservices ist die erforderliche Automatisierung der Infrastruktur, da eine manuelle Bereitstellung aufgrund der vielen Artefakte nicht mehr möglich ist. Außerdem ist eine Umstrukturierung der Teams notwendig, sodass ein Microservice vom selben Team entwickelt und bereitgestellt werden kann. Dafür wird ebenso eine dezentrale Führung notwendig, damit Teams eigenständig über die Implementierung ihres Microservice entscheiden können. Dezentralisierung wird auch im Datenmanagement der Applikation gefordert, weshalb jeder Service in einer Microservice Architektur seine eigene Datenbank besitzt. Eine weitere Charakteristik von Microservices ist die Isolierung von Ausfällen im System, um eine Ausfallkaskade der gesamten Applikation zu vermeiden. Hierfür ist ebenso eine hohe Überwachung des Systems notwendig, um Ausfälle im System frühzeitig erkennen zu können.

Im darauffolgenden Kapitel wurde vorgestellt wie ein bereits bestehender Monolith in Microservices aufgeteilt werden kann.
Bevor der Monolith aufgespalten werden kann werden eine autonome Infrastruktur und Bereitstellungskette sowie den Charakteristiken entsprechenden Teamstrukturen vorausgesetzt. Anschließend muss der Monolith zunächst analysiert werden und anschließend wird mithilfe von Refactoring der Code nach Kontext umstrukturiert, damit Abhängigkeiten zwischen späteren Services frühzeitig offensichtlich werden. In der Aufteilung des Monolithen geht es vor allem darum, die Abhängigkeiten der Datenbank aufzubrechen und bisherige Methodenaufrufe durch entsprechende Schnittstellen zu ersetzen.

Zum Schluss der Arbeit wurden die Herausforderungen beschrieben, welche bei der Verwendung von Microservices entstehen können.
Dabei wurde die Schwierigkeiten für die Entwickler während der Implementierungsphase aufgegriffen, welche durch die vielen kleinen Anwendungen entstehen. Des Weiteren wurde das Risiko von schlechter Performance und die höheren Instandhaltungskosten genannt,

welche aus die hohe Anzahl an Services und deren Overhead resultieren. Durch die große Anzahl an Services entsteht zudem eine Komplexität in der Infrastruktur, da nicht mehr wie bei einem Monolithen klar ist auf welcher virtuellen Maschine jeder Service läuft.
Anschließend wurde auf die höheren Anforderungen an die Entwickler eingegangen, welche aufgrund des breiteren Aufgabenbereichs entstehen und zuletzt wurde die unklare Kontextabgrenzung in der echten Welt dargestellt, welche die Abgrenzung zwischen Microservices zu einer Herausforderung macht.

6.2 Fazit

Microservices bieten vor allem mehr Optionen und Möglichkeiten im Vergleich zu einer monolithischen Architektur. Allerdings führen mehr Möglichkeiten auch dazu, dass viel mehr Entscheidungen in der Entwicklungsphase der Software getroffen werden müssen. Dies kann zu Konflikten im Team oder im Unternehmen führen und somit auch die Entwicklung verzögern.
Allgemein kann festgestellt werden, dass Microservices insbesondere für Applikationen geeignet sind, welche flexibel, anpassbar und skalierbar sein sollen. Dies trifft vor allem auf große Online Applikationen oder die die es werden wollen zu, weshalb Amazon und Netflix mit die ersten großen Unternehmen waren, welche ihre Anwendungen mit Microservices umgesetzt haben. Im Entscheidungsprozess für oder gegen Microservices sollten vor allem die Entwickler und Software Architekten im Mittelpunkt stehen, weil diese sich tagtäglich mit der Anwendung auseinander setzen müssen. Denn ohne entsprechende Erfahrungen kann eine Microservice Architektur in einer Vielzahl von unorganisierten Services enden, die nicht als ein System zusammenarbeiten können. Deshalb sollte zu Beginn eher eine monolithische Anwendung präferiert werden, welche später durch Microservices in ihrer Funktionalität erweitert werden kann.
Alles in allem bieten Microservices eine sehr gute Alternative zum Monolithen. Allerdings muss die Entscheidung für eine Software Architektur immer individuell für jedes Softwareprodukt vorgenommen werden und kann nur schwer verallgemeinert werden.

ABBILDUNGSVERZEICHNIS

LITERATURVERZEICHNIS

Fowler, M. (2019, 03 31). *Martin Fowler Website*. Retrieved from https://martinfowler.com/articles/microservices.html

Haq, S. u. (2019, 04 07). *Medium*. Retrieved from Medium: https://medium.com/koderlabs/introduction-to-monolithic-architecture-and-microservices-architecture-b211a5955c63

Masashi Narumoto, M. W. (2019, 04 09). *Microsoft*. Retrieved from Microsoft: https://docs.microsoft.com/en-us/azure/architecture/patterns/bulkhead

Newman, S. (2015). Building Microservices. In S. Newman, *Building Microservices.* O´Reilly Media.

Vogels, W. (2019, 03 31). *acmqueue*. Retrieved from https://queue.acm.org/detail.cfm?id=1142065

Zhamak, D. (2019, 04 13). *Martin Fowler*. Retrieved from Martin Fowler: https://martinfowler.com/articles/break-monolith-into-microservices.html