MASTERARBEIT

MICROSERVICES IM PRAXISEINSATZ

Modernisierung von Legacy-Systemen mit Microservices

ausgeführt am

Studiengang

Informationstechnologien und Wirtschaftsinformatik

Von: Stefan Schweiger

Personenkennzeichen: 1510320019

Graz, am 21. März 2017 .....

Unterschrift

EHRENWÖRTLICHE ERKLÄRUNG

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

.....

Unterschrift

DANKSAGUNG

Ich möchte mich an dieser Stelle bei allen Personen bedanken, die mich im Laufe meines Studiums und insbesondere bei der Erstellung dieser Masterarbeit unterstützt haben.

Besonderer Dank gilt meinem Betreuer Herrn Dipl.-Ing. (FH) Günther Zwetti, der mich während der Erstellung dieser Arbeit stets unterstützte und hilfreich zur Seite stand.

Bei Janine bedanke ich mich herzlichst für das wiederholte Durchlesen und Korrigieren meiner Arbeit sowie die zahlreichen Diskussionen, die dazu geführt haben, dass diese Arbeit in dieser Form verfasst werden konnte.

Meinem Arbeitgeber und meinen Kollegen danke ich dafür, dass mir ein berufsbegleitendes Studium ermöglicht wurde und ich die notwendige Unterstützung dafür bekommen habe.

Nicht zuletzt möchte ich mich bei meiner Freundin, meiner Familie und meinen Freunden dafür bedanken, dass sie mir in allen Lebenslagen stets zur Seite stehen und den nötigen Rückhalt während meines Studiums gegeben haben. Ohne eure unendliche Geduld mit mir und dem entgegengebrachten Verständnis wäre dies so nicht möglich gewesen.

KURZFASSUNG

Durch die Aufteilung eines Systems in eine Vielzahl kleine, autonom agierende Services versprechen Microservices viele Vorteile wie beispielsweise die Komplexität von Software-Systemen zu reduzieren sowie deren Flexibilität und Wartbarkeit zu erhöhen. Dabei sollen Services von selbstständigen, voneinander unabhängigen Teams entwickelt, bereitgestellt und betrieben werden, um schnell auf geänderte Anforderungen reagieren und einzelne Systemteile ohne Beeinträchtigung des Gesamtsystems ändern zu können. Dies ermöglicht in weiterer Folge die gesonderte Skalierung einzelner Teile eines Systems, um so die Verfügbarkeit gezielt erhöhen und Betriebskosten reduzieren zu können.

Diese Arbeit führt eine kritische Betrachtung dieses Architekturansatzes durch und ermittelt, welche Voraussetzungen notwendig sind, um damit monolithische, webbasierte Systeme zu modernisieren oder zu erweitern. Aufbauend auf einer theoretischen Ausarbeitung, welche in die Grundlagen von Microservices-Architekturen einführt, Methoden zur Erweiterung und Modernisierung monolithischer Legacy-Systeme beschreibt sowie Wege zeigt, wie ein bestehendes System aufgeteilt werden kann, wird ein Fragebogen ausgearbeitet, um die Erwartungshaltung von Personen im Umfeld der Softwareentwicklung zu erheben.

Die Untersuchung zeigte, dass Unternehmen ein großes Interesse und eine hohe Erwartungshaltung an diesen technologischen Ansatz haben, sich aber auch der damit einhergehenden Nachteile, wie unter anderem die Erhöhung der technischen Komplexität des Gesamtsystems oder den steigenden Anforderungen an Betrieb und Infrastruktur, bewusst sind. Weiters wurde festgestellt, dass sich dieses Architekturparadigma nicht für jedes Unternehmen eignet und in der Regel organisatorische Änderungen für deren Einführung notwendig sind.

Abschließend wird eine Handlungsempfehlung präsentiert, die, unter Berücksichtigung der zuvor ermittelten Ergebnisse, Möglichkeiten aufzeigt, diesen Architekturansatz auf bestehende Systeme anzuwenden sowie notwendige Voraussetzungen veranschaulicht, welche eine Organisation für die Einführung von Microservices zu erfüllen hat.

ABSTRACT

By splitting a system into small, autonomous services, the microservice paradigm promises many advantages, such as reducing the complexity of software systems and increasing their flexibility and maintainability. These services are independently deployable units and should be fully developed, deployed and operated by only one team in order to be able to react quickly to changed requirements and to replace individual components without affecting the overall system. Furthermore, this enables each service to be scaled independently and can help to reduce overall operating costs.

The goal of this thesis was to determine the prerequisites for using this architectural approach to modernizing or extending monolithic, web-based systems. First, a literature review was performed to provide an introduction to the basic principles of microservice architectures and to describe several ways to divide and extend an existing system. This information was used to design a survey to identify the expectations of people in the software development industry for this topic.

The results show that companies have a great interest in and high expectations for this technological approach, but are also aware of the related disadvantages, such as the increased technical complexity of the overall system or the increasing demands on operation and infrastructure. Furthermore, it was found that this architecture paradigm is not suitable for every company, and organizational changes are usually necessary for its implementation.

Finally, a recommendation for action is presented, which shows the possibilities to apply microservice architectures to existing systems and describes prerequisites that an organization has to fulfill for the introduction of this kind of software architecture.

INHALTSVERZEICHNIS

1 EINLEITUNG 1

1.1 Abgrenzung des Themas 1
1.2 Forschungsmethodik und Forschungsfrage 2
1.3 Aufbau der Arbeit 2

2 EINFÜHRUNG IN MICROSERVICES 4

2.1 Begriffsdefinitionen 4
2.1.1 Software-Architekturen 4
2.1.2 Module, Komponenten und Services 5
2.1.3 Legacy-Code und -Systeme 7
2.1.4 Monolithen 7
2.1.5 Service-orientierte Architekturen 8

2.2 Grundlagen 10
2.2.1 Begriffsdefinition Microservice 10
2.2.2 Eigenschaften und Prinzipien 11
2.2.3 Gründe für die Einführung von Microservices 12
2.2.4 Nachteile von Microservices 13

2.3 Microservices-Architekturen 15
2.3.1 Größe von Microservices 15
2.3.2 Kommunikation 17
2.3.3 Datenmanagement 19
2.3.4 Cross-Cutting Concerns 20

2.4 Zusammenfassung 21

3 ERWEITERUNG UND MODERNISIERUNG VON LEGACY-SYSTEMEN 23

3.1 Softwareevolution 23
3.1.1 Softwarewartung 24
3.1.2 Migrationsstrategien 26
3.1.3 Software-Reengineering 27

3.2 Schrittweise Erweiterung und Ablöse von Altsystemen 30

3.2.1 Chicken-Little-Strategie 30
3.2.2 Butterfly-Methode 32
3.2.3 Split Backend and Frontend 34
3.2.4 Change by Split 34
3.2.5 Branch by Abstraction 35
3.2.6 Strangler Application 36
3.2.7 Extract Services 38

3.3 Schnittpunkte finden 39
3.3.1 Fachliche Trennung mittels Domain-Driven Design 39
3.3.2 DDD im Kontext von Microservices und Legacy-Systemen 41
3.3.3 Dezentrales Datenmanagement 44
3.3.4 Benutzerschnittstelle 49

3.4 Zusammenfassung 52

4 EMPIRISCHER TEIL 54

4.1 Hypothesen 54
4.1.1 Hypothese 1 54
4.1.2 Hypothese 2 55
4.1.3 Hypothese 3 56
4.1.4 Hypothese 4 57
4.1.5 Hypothese 5 58

4.2 Befragung 59
4.2.1 Aufbau und Inhalt 59
4.2.2 Durchführung 61
4.2.3 Auswertung 61
4.2.4 Analyse und Interpretation 70

4.3 Prüfung der Hypothesen 72
4.3.1 Hypothese 1 72
4.3.2 Hypothese 2 74
4.3.3 Hypothese 3 74
4.3.4 Hypothese 4 75
4.3.5 Hypothese 5 76

4.4 Marktbetrachtung 77
4.4.1 Amazon 77

4.4.2 Netflix 79
4.4.3 Volkswagen 82
4.4.4 Fazit 83

5 ERGEBNISSE, ZUSAMMENFASSUNG UND AUSBLICK 85

5.1 Handlungsempfehlung 85
5.1.1 Rahmenbedingungen für Microservices im Unternehmen schaffen 85
5.1.2 Erweiterung und Modernisierung monolithischer Legacy-Systeme 87

5.2 Zusammenfassung und Diskussion 88
5.3 Ausblick 90

ANHANG A - FRAGEBOGEN 91

ANHANG B - ERGEBNISSE DER BEFRAGUNG 96

ABKÜRZUNGSVERZEICHNIS 102

ABBILDUNGSVERZEICHNIS 103

TABELLENVERZEICHNIS 104

LITERATURVERZEICHNIS 105

1 EINLEITUNG

Jede Erweiterung von monolithischen Software-Systemen um neue Funktionalitäten bedingt in der Regel auch die Ausdehnung der Codebasis und führt somit zu einer Erhöhung der Anzahl an neuen Source Code Zeilen. Im Laufe der Zeit werden diese Systeme immer komplexer und Änderungen immer schwieriger umzusetzen, sodass in der Regel dafür immer mehr Aufwand benötigt wird. Zusätzlich muss ein solches System immer als Ganzes ausgetauscht werden, auch wenn beispielsweise nur ein internes Modul fehlerhaft ist oder erweitert wurde. Dies führt unter anderem zu langen Durchlaufzeiten und einem immer höher werdenden Testaufwand, da bei jeder Änderung das vollständige System getestet werden sollte.

Das relativ neue Architekturparadigma von Microservices soll diesen Problemen entgegensteuern, indem Software-Systeme aus einer Vielzahl kleiner, eigenständiger Services zusammengebaut werden und voneinander unabhängig funktionieren. Viele große Unternehmen wie beispielsweise Amazon, Google, Netflix oder Zalando setzen bereits auf dieses Paradigma, welches eines der großen “buzzwords” der jüngeren Vergangenheit darstellt. In fast jedem aktuellen Entwicklermagazin, technischem Blog oder Fachbuch sowie auf einer Vielzahl von Entwicklerkonferenzen wird dieses Thema von verschiedensten Seiten betrachtet und eine Vielzahl möglicher Einsatzszenarien von Microservices vorgestellt.

Ziel dieser Arbeit ist es nun, die Architekturansätze dieses neuen Paradigmas, dessen Vor- und Nachteile und die Technologien dahinter zu beleuchten und aufzuzeigen, unter welchen Umständen und wie monolithische und schwergewichtige Legacy-Systeme schrittweise durch eine Vielzahl autonom agierender Microservices abgelöst beziehungsweise erweitert werden können. Weiters soll erhoben werden, welche Erwartungen Personen im Umfeld der Softwareentwicklung an Microservices haben und welche Best-Practice-Ansätze in der Praxis existieren.

Daraus soll schlussendlich eine Handlungsempfehlung entstehen, welche einerseits Unternehmen bei der Entscheidung für oder gegen den Einsatz von Microservices unterstützen soll und andererseits einen Weg zeigt, wie bei einer schrittweisen Ablöse von monolithischen Webapplikationen vorzugehen ist.

1.1 Abgrenzung des Themas

Die vorliegende Arbeit beschränkt sich auf monolithische Legacy-Anwendungen im Java Enterprise Umfeld und betrachtet Möglichkeiten, diese mit Hilfe des Microservices-Architekturansatzes schrittweise abzulösen und zu modernisieren.

1.2 Forschungsmethodik und Forschungsfrage

Als Methode wird eine deduktive Vorgehensweise gewählt. Die Evaluation erfolgt mittels quantitativen und qualitativen Forschungsmethoden, welche nachfolgend beschrieben werden.

Im theoretischen Teil der Arbeit wird das genannte Forschungsthema anhand wissenschaftlicher Fachliteratur aufbereitet und diskutiert. Anschließend werden auf Basis der theoretischen Ausarbeitung Annahmen in Form von Hypothesenpaaren aufgestellt, welche in weiterer Folge geprüft werden. Auf Basis der theoretischen Erkenntnisse und der davon abgeleiteten Hypothesen wird ein Fragebogen für Personen aus den Fokusgruppen Softwareentwicklung, Software-Architektur und IT-Management ausgearbeitet, um deren Erwartungen an Microservices und möglichen Einsatzszenarien zu erheben. Durch diese Befragung sollen dem Autor entsprechend tiefe Einblicke in die speziellen Anforderungen, Bedürfnisse und Erwartungen der Zielgruppen geliefert werden, welche anschließend ausgewertet, analysiert und interpretiert werden. Aus den somit gesammelten Informationen kann in weiterer Folge die Prüfung der Hypothesen durchgeführt werden, um diese verifizieren oder falsifizieren zu können.

Mit dem gewonnenen Wissen aus der theoretischen Analyse und den Ergebnissen der Befragung wird im Anschluss eine Marktbetrachtung durchgeführt. Dabei werden publizierte Microservices-Beispielarchitekturen von verschiedenen Unternehmen kritisch betrachtet, gegenübergestellt und mit den gesammelten Erkenntnissen aus der Umfrage und der Theorie verglichen.

Diese Vorgehensweise soll dazu führen, die Forschungsfrage dieser Masterarbeitet zu beantworten, welche wie folgt lautet:

Unter welchen Voraussetzungen können monolithische Java Enterprise Webapplikationen durch autonom agierende Microservices schrittweise abgelöst oder erweitert werden?

1.3 Aufbau der Arbeit

Nach einer grundlegenden Literaturrecherche und -analyse zu den Themen Software-Architekturen und Microservices werden die wesentlichen Elemente dieses Architekturmusters betrachtet und anderen Architekturansätzen gegenübergestellt. Weiters werden die Vor- und Nachteile von Microservices ermittelt und Unterschiede zu monolithischen Architekturen diskutiert.

Als nächster Schritt werden Methoden zur Erweiterung und Modernisierung von monolithischen Legacy-Systemen betrachtet und aus der Literatur entsprechende strategische Ansätze abgeleitet. Darauf folgend wird die Anwendbarkeit von Microservices auf eine schrittweise Migration und dabei Möglichkeiten zur Aufteilung eines monolithischen Systems untersucht.

Im nächsten Kapitel werden aus den theoretischen Erkenntnissen Hypothesen abgeleitet und diese operationalisiert. Um diese zu prüfen, wird in weiterer Folge auf Basis der zuvor diskutierten Inhalte ein Fragebogen erstellt. Dieser soll unter anderem Fragen zu Problemen mit aktuell eingesetzten Architekturen beinhalten sowie die Erwartungshaltung an Microservices und die Bereitschaft, diesen Architekturansatz einzuführen, ermitteln. Nach der Durchführung der

Befragung werden die Ergebnisse ausgewertet und einer detaillierten Analyse unterzogen, um daraus Rückschlüsse auf die zuvor festgelegten Hypothesen ziehen zu können.

Bei der anschließenden Marktbetrachtung erfolgt eine kritische Analyse und Gegenüberstellung publizierter Beispielarchitekturen und Best-Practice-Ansätzen. Dadurch sollen die Ergebnisse aus der theoretischen Analyse und der zuvor durchgeführten Befragung evaluiert und in weiterer Folge geprüft werden, ob sich in der Praxis Ansätze für die Erwartungen der Experten wiederfinden.

Abschließend soll eine Handlungsempfehlung für die schrittweise Ablöse von monolithischen Applikationen unter Berücksichtigung des Microservices-Architekturparadigmas erstellt, die ermittelten Ergebnisse dieser Arbeit zusammengefasst und diskutiert sowie die zuvor festgelegte Forschungsfrage beantwortet werden.

2 EINFÜHRUNG IN MICROSERVICES

Um die Unternehmensqualität und die Unternehmensleistung zu steigern und dadurch Wettbewerbsvorteile zu schaffen, sollten die Geschäftsprozesse eines Unternehmens einem ständigen Verbesserungsprozess unterliegen (Becker, Mathas, & Winkelmann, 2009). Die mit dem Verbesserungsprozess einhergehenden Änderungen an den Abläufen und Aktivitäten innerhalb des Unternehmens wirken sich in der Regel auch auf die eingesetzten Software-Systeme aus (Schmelzer & Sesselmann, 2013).

Geänderte Anforderungen haben gewöhnlich zur Folge, dass Anpassungen an den eingesetzten Systemen notwendig sind, welche effizient erfolgen und bereitgestellt werden sollen. Um auf Änderungen schnell reagieren zu können, strebt die Softwareindustrie seit jeher nach Systemen, die aus einzelnen Komponenten zusammengebaut werden und einen Austausch einzelner Teile auf einfache Art und Weise ermöglichen (Fowler & Lewis, 2015). Der Microservices-Architekturstil soll unter anderem die Änderbarkeit einer Software verbessern (Wolff, 2015) und die Dauer für die Bereitstellung einer neuen Programmversion reduzieren (Newman, 2015).

Dieses Kapitel bietet eine Einführung in Microservices und Microservices-Architekturen. Zu Beginn werden grundlegende Begriffe im Kontext von Software-Architekturen beschrieben und verschiedene Architekturansätze diskutiert und gegenübergestellt. In weiterer Folge werden die Grundlagen von Microservices erörtert und dabei die Vor- und Nachteile einer solchen Lösung beleuchtet. Abschließend wird der Aufbau einer Microservices-Architektur und dabei zu berücksichtigende Herausforderungen untersucht.

2.1 Begriffsdefinitionen

Nachfolgend werden grundlegende Begriffe, welche im Rahmen dieser Arbeit wiederholt verwendet werden, definiert und abgegrenzt.

2.1.1 Software-Architekturen

Eine Software-Architektur – in weiterer Folge auch als Architektur bezeichnet – beschreibt den Aufbau eines Software-Systems, dessen Bestandteile und welche Beziehungen diese untereinander aufweisen (Clements et al., 2010). Jedes Software-System trägt eine Architektur in sich, auch wenn diese nicht explizit im Vorfeld modelliert wurde. Ausgehend von den Anforderungen legt eine Architektur die grundlegende Struktur einer Software fest. Interne Details der einzelnen Bestandteile, welche für die jeweilige Implementierung ausschlaggebend sind, werden dabei nachrangig behandelt. (Vogel et al., 2009)

Beim Entwurf einer Software-Architektur wird ein System in einzelne Teile zerlegt und es müssen frühzeitig Entscheidungen zur Gestaltung des Systems getroffen werden, welche später schwer änderbar sind (Fowler, 2002). Eine Architektur beeinflusst dabei wesentliche Eigenschaften des Systems wie beispielsweise Änderbarkeit, Geschwindigkeit oder Sicherheit (Fairbanks, 2010).

Die Zerlegung des Systems ermöglicht es unter anderem, einzelne Teile zu betrachten, Zusammenhänge zu erkennen und gemeinsam ein Problem zu lösen. Dies wird von Clements et al. (2010) wie folgt beschrieben:

> *"Architecture is roughly the prudent partitioning of a whole into parts, with specific relations among the parts. This partitioning is what allows groups of people—often separated by organizational, geographical, and even time-zone boundaries—to work cooperatively and productively together to solve a much larger problem than any of them could solve individually."* (Clements et al., 2010, S. 1)

Um die Struktur eines Systems darzustellen, wird ein System von verschiedenen Seiten aus betrachtet (Fowler, 2002). Jede Sicht befasst sich dabei mit einem bestimmten Aspekt eines Systems und ermöglicht dadurch, ein komplexes Gesamtsystem aus unterschiedlichen Blickwinkeln zu betrachten (Vogel et al., 2009).

Die Zerlegung von Systemen erfolgt gewöhnlich auf Basis von definierten Architekturstilen und Architekturmustern. Zu diesen zählt beispielsweise die Schichten-Architektur, welche ein System in verschiedene Ebenen unterteilt und dem Architekturstil der hierarchischen Systeme zuzuordnen ist. (Starke, 2015)

Auf eine detaillierte Beschreibung des Begriffs Architektur wird an dieser Stelle verzichtet, jedoch werden im Verlauf dieser Arbeit weitere Aspekte aufgegriffen und diskutiert. Für ausführliche Diskussionen zum Thema Software-Architekturen wird an dieser Stelle auf entsprechende Literatur wie beispielsweise Starke (2015), Vogel et al. (2009) oder Clements et al. (2010) verwiesen.

2.1.2 Module, Komponenten und Services

Um ein Software-System in einzelne Teile zu untergliedern und diese im Kontext einer Software-Architektur zu beschreiben, werden häufig die Begriffe „Module" und „Komponenten" verwendet. Dabei fehlt oftmals eine genaue Abgrenzung dieser Begrifflichkeiten.

Ein Modul ist eine Sammlung von Implementierungsartefakten wie etwa Klassen, Schnittstellen oder Konfigurationsdateien. Die einzelnen Elemente eines Moduls müssen nicht zueinander in Beziehung stehen, haben aber in der Regel ähnliche Funktionalitäten oder verfolgen dieselben Zwecke. Dies können beispielsweise mathematische Funktionen oder gemeinsam genutzte Datentypen sein, die zu einer Einheit gruppiert werden sollen (Fairbanks, 2010). Module werden auch verwendet, um Daten und Informationen zu verbergen und den Zugriff auf diese nur über definierte Schnittstellen zu ermöglichen (Clements et al., 2010).

Komponenten sind wie Module eine Sammlung verschiedener Artefakte, der Fokus liegt jedoch auf dem fertigen Produkt in Form einer ausführbaren Binärdatei und dem Verhalten zur Laufzeit, nicht an den Implementierungsdetails (Clements et al., 2010). Das interne Verhalten einer Komponente kann als Blackbox gesehen werden, die Schnittstelle nach außen muss aber

verbindlich definiert werden. Diese Schnittstellendefinition beinhaltet unter anderem eine Auflistung der nach außen zur Verfügung stehenden Funktionalitäten, wie diese zu verwenden sind und welche Fehlerzustände auftreten können. Durch diese expliziten Schnittstellen können Komponenten miteinander interagieren und in eine größere Applikation integriert werden (Brereton & Budgen, 2000). Komponenten werden zur Laufzeit instanziiert und können unabhängig ausgetauscht oder erweitert werden (Fowler & Lewis, 2015), wodurch bei einem komponentenbasierten System die Austauschbarkeit einzelner Bestandteile ermöglicht wird (Clements et al., 2010).

Da eine Komponente aus mehreren Modulen bestehen und ein Modul mehrere Komponenten bereitstellen kann (Clements et al., 2010), ist die Unterscheidung zwischen diesen Begriffen vor allem auf konzeptueller Ebene wichtig, um das Zusammenspiel der Bestandteile eines Systems zu verstehen. Ein Modul kann als Komponente bezeichnet werden, wenn es eine entsprechende Schnittstelle zur Verfügung stellt. Ist ein Modul aber lediglich eine Sammlung von unabhängigen Funktionen ohne eine Schnittstelle zu beinhalten, kann es nicht als Komponente bezeichnet werden (Fairbanks, 2010). Clements et al. (2010) skizzieren diese Unterscheidung anhand eines Systems, welches sich aus einem Client- und einem Server-Modul zusammensetzt. Das Server-Modul entspricht einer Komponente, die zur Laufzeit mit einer Vielzahl an Clients interagiert. Entsprechend der Anzahl an Clients werden aus einem Client-Modul zur Laufzeit ebenso viele Client-Komponenten. Nachfolgende Abbildung 2-1 stellt dies grafisch dar.

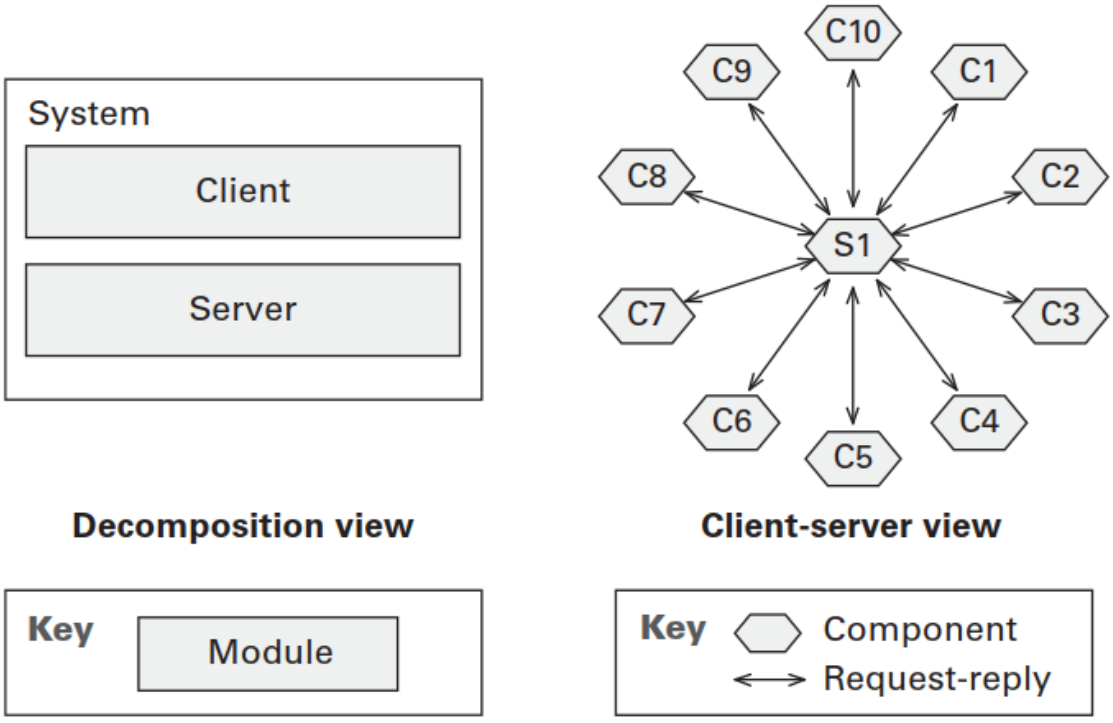

Abbildung 2-1: Module und Komponenten (Clements et al., 2010)

Nach ten Hompel und Schmidt (2008) sind Services eine Weiterentwicklung von Komponenten und werden im Umfeld von service-orientierten Architekturen verwendet. Dabei weisen sie darauf hin, dass die Kommunikation zwischen Services in der Regel über ein Netzwerk erfolgt. Auch Fowler (2004a) geht in seiner Definition von Services auf den Aspekt der Kommunikation ein und

grenzt Services von Komponenten wie folgt ab: Komponenten werden lokal in einer Applikation verwendet und kommunizieren miteinander in-memory. Services hingegen kommunizieren mittels Remote-Schnittstellen synchron oder asynchron mit anderen Services und verlassen dabei die Grenzen des jeweiligen Prozesses. Daher können Services auch als *out-of-process - Komponenten* bezeichnet werden (Fowler & Lewis, 2015).

Nach Balakrushnan et al. (2016) ist ein Service unabhängig, kann aus anderen Services zusammengesetzt werden und stellt nach außen hin eine Blackbox dar. Weiters dient ein Service dazu, eine wiederholbare Geschäftsaktivität mit einem definierten Ergebnis abzubilden.

2.1.3 Legacy-Code und -Systeme

Als Legacy-System oder auch Altsystem wird ein historisch gewachsenes Software-System bezeichnet, dessen Funktionen wichtig für ein Unternehmen sind, aber in der Regel mit – zum aktuellen Zeitpunkt – veralteten Technologien entwickelt wurde. Bisbal, Lawless, Wu und Grimson (1999) definieren ein solches wie folgt:

> *"Legacy information systems are typically the backbone of an organization's information flow and the main vehicle for consolidating business information. They are thus mission critical, and their failure can have a serious impact on business."* (Bisbal et al., 1999, S. 1)

Legacy-Systeme bestehen in der Regel aus einer hohen Anzahl an Quellcode-Zeilen und müssen mit hohem Aufwand erweitert sowie gewartet werden (Bennett, 1995). Eine stetige Verbesserung und Weiterentwicklung ist jedoch ausschlaggebend für den Erfolg einer Software (Lehman, 1980). Durch diese Erweiterung des Quellcodes entstehen in der Regel immer mehr Abhängigkeiten von Codeteilen zueinander und die Komplexität des Systems nimmt zu. Dies kann dazu führen, dass die Software Schritt für Schritt von der geplanten Architektur abweicht und über eine Erosion der Architektur zu einem Legacy-System wird. (Wolff, 2015)

Feathers und Martin (2005) betrachten bei ihrer Definition von Legacy-Systemen den der Software zugrunde liegenden Quellcode. Dieser wird als Legacy-Code bezeichnet, wurde gewöhnlich von einer anderen Person geschrieben oder von einem anderen Unternehmen zugekauft. Aus ihrer Sicht ist die Verfügbarkeit von Tests entscheidend für die Einstufung von Quellcode als Legacy-Code. Sind ausreichend Tests zur Sicherstellung der Funktionsweise des Codes vorhanden, kann dieser gefahrlos geändert werden, fehlen solche Tests, kann jede Änderung unerwünschte Auswirkungen haben. Daher bezeichnen Feathers und Martin (2005) Quellcode ohne Tests als Legacy-Code, da nicht sichergestellt werden kann, ob Änderungen den Quellcode verbessern oder nicht.

2.1.4 Monolithen

Als Monolith oder monolithisches System wird eine Software bezeichnet, deren Bestandteile zu einem einzigen, direkt ausführbaren Programm zusammengefasst werden (Fowler & Lewis,

2015). Der interne Aufbau eines Monolithen kann dabei beliebig gewählt werden und aus verschiedenen Modulen, Komponenten und Services bestehen (Namiot & Sneps-Sneppe, 2014). Fowler und Lewis (2015) streichen dabei die Verbreitung solcher Monolithen im Unternehmensumfeld hervor und skizzieren einen typischen Aufbau einer monolithischen Webapplikation wie folgt:

- *Serverseitige Applikation*: Nimmt Anfragen an das System mittels HTTP-Requests entgegen, führt die Geschäftslogik aus und kommuniziert mit der Datenbank. Als Antwort auf den Aufruf wird in der Regel generierter HTML-Code an den Client zurückgesendet.
- *Datenbank*: Speichert die für die Anwendung relevanten Daten in Tabellen eines relationalen Datenbankschemas.
- *Benutzerschnittstelle*: Der von der serverseitigen Applikation gelieferte HTML-Code wird im Web-Browser des Clients ausgeführt.

Wird ein Teil der Software geändert, muss das gesamte System neu gebaut und bereitgestellt werden. Der Prozess eine neue Version einer Software in Betrieb zu nehmen umfasst dabei verschiedene Phasen, zu denen Tests, Abnahme und Release zählen und welche das System als Ganzes durchlaufen muss. Diese Inbetriebnahme wird als Deployment bezeichnet und kann je nach Größe des Systems unterschiedlich lange dauern, was die Flexibilität reduzieren und zu erhöhten Kosten führen kann. (Wolff, 2015)

Villamizar et al. (2015) betrachten monolithische Anwendungen im Kontext von cloudbasierten Lösungen und beschreiben diese wie folgt:

> *"[…] a monolithic application means an application with a single large codebase/repository that offer tens or hundreds of services using different interfaces such as HTML pages, Web services or/and REST services."* (Villamizar et al., 2015, S. 583)

Im weiteren Verlauf dieser Arbeit wird auf diese Definition verwiesen, wenn monolithische Anwendungen diskutiert werden.

2.1.5 Service-orientierte Architekturen

Service-orientierte Architekturen sind ein Basisprinzip für die Architektur von verteilten IT-Infrastrukturen (Becker et al., 2009) und folgen dem modularen Entwicklungsparadigma, um aus einer Auswahl von Services betriebliche Anwendungssysteme zu realisieren (Overhage & Turowski, 2007). Im Mittelpunkt stehen dabei eigenständige Services, welche über allgemein zugängliche Schnittstellen Anwendungslogik zur Verfügung stellen. Die einzelnen Services stehen auf einer technisch gleichberechtigten Ebene, sind funktional klar voneinander getrennt (Becker et al., 2009) und dienen einem bestimmten Geschäftszweck (Chu, 2005).

In der Literatur finden sich eine Vielzahl von Definitionen und unterschiedlichen Sichtweisen zum Thema service-orientierte Architekturen (SOA), eine exakte Definition des Begriffs ist jedoch nicht zu finden (Erl, 2007; Josuttis, 2007; Overhage & Turowski, 2007). In der Regel lässt sich aber

aus allen Beschreibungen die Gemeinsamkeit ableiten, dass SOA ein Architekturparadigma darstellt, welches zur Erhöhung der Flexibilität von Software-Architekturen beitragen kann (Josuttis, 2007).

Overhage und Turowski (2007) bezeichnen SOA als ein technisches Managementkonzept, das eine Ausrichtung der IT-Infrastruktur an den Geschäftsprozessen des jeweiligen Unternehmen fordert. Aus technischer Sicht fassen sie verschiedene Definitionen für SOA zusammen und definieren diese wie folgt:

> *„Sammelbegriff für ein bestimmtes Strukturparadigma für (betriebliche) Anwendungssysteme, das, gegenüber anderen Ansätzen der (Software-) Wiederverwendung auf den Aspekt der (Dienst-) Verwendung fokussiert."*
> (Overhage & Turowski, 2007, S. 4)

Erl (2007) weist darauf hin, dass SOA ein Architekturmodell etablieren, welche durch den Einsatz von Services die Effizienz, Agilität und Produktivität eines Unternehmens erhöhen und die Realisierung der damit verbundenen strategischen Ziele unterstützen sollen. Dabei soll dieses Architekturmodell auf acht Designprinzipien aufbauen, welche Erl (2007) wie folgt beschreibt:

- *Standardized Service Contract:* Jedes Service weist einen Servicevertrag auf, welcher die Eigenschaften des jeweiligen Service auf standardisierte Weise beschreibt. Dieser Vertrag beinhaltet unter anderem eine Beschreibung der zur Verfügung stehenden Funktionalitäten, welche Daten das Service entgegennimmt und welche Rückgabewerte das Service liefern kann.
- *Service Loose Coupling:* Es sollten möglichst wenige Abhängigkeiten zwischen dem Servicevertrag, dem Konsument und der Implementierung eines Service existieren. Eine Änderung der Implementierungslogik soll sich bei gleichbleibendem Servicevertrag nicht negativ auf den Konsumenten auswirken.
- *Service Abstraction:* Ein Service soll seine interne Logik nach außen hin verbergen und keine Implementierungsdetails preisgeben, um dadurch das Konzept der losen Kopplung zu unterstützen.
- *Service Reusability:* Ein Service soll wiederverwendbar sein und von unterschiedlichen Aufrufern wiederholt konsumiert werden können.
- *Service Autonomy:* Ein Service soll die Kontrolle über seine Ressourcen und seine Umgebung haben und diese durch die im Service enthaltene Logik autonom steuern.
- *Service Statelessness:* Ein Service sollte möglichst zustandslos agieren und zustandsbehaftete Operationen minimieren. Je mehr Zustandsinformationen in einem Service gekapselt werden müssen, desto höher ist der zu erwartende Ressourcenbedarf, was sich in weiterer Folge auf die Verfügbarkeit und Skalierbarkeit des Service negativ auswirken kann.
- *Service Discoverability:* Ein standardisierter Servicevertrag führt zu selbstbeschreibenden Services, welche von Suchmechanismen gefunden und in weiterer Folge von

Konsumenten genutzt werden können. Um die Wiederverwendbarkeit zu erhöhen und Services leichter auffinden zu können, kann ein Katalog in Form eines Service-Repository eingesetzt werden.

- *Service Composability:* Services sollen so gestaltet sein, dass sie zu einem größeren Service kombiniert oder in zusammengesetzten Services verwendet werden können.

2.2 Grundlagen

Aufbauend auf den Erfahrungen, die im Design, der Entwicklung und im Betrieb von Software gemacht wurden, stützen sich Microservices-Architekturen (MSA) auf weit verbreitete Paradigmen, Prinzipien und Methoden der Softwareentwicklung. Zu diesen zählen unter anderem die objektorientierte Programmierung, Domain-Driven Design (DDD) und vor allem SOA (Balakrushnan et al., 2016). MSA werden daher auch oftmals als eine Abwandlung von SOA bezeichnet (Newman, 2015; Wolff, 2015).

In den nachfolgenden Abschnitten werden die Eigenschaften sowie Vor- und Nachteile von Microservices im Detail diskutiert.

2.2.1 Begriffsdefinition Microservice

Eine eindeutige formale Definition der Begriffe „Microservices“ und „Microservices-Architektur“ lässt sich in der Literatur kaum finden. Nachfolgend werden verschiedene Sichtweisen dieses Begriffs, welche in der Literatur wiederholt vorkommen, erläutert.

Beim Microservices-Architekturstil wird eine Anwendung aus einer Vielzahl von eigenständigen Services zusammengestellt (Newman, 2015). Ein einzelnes Microservice ist in sich abgeschlossen, unabhängig von anderen Services und verfolgt einen bestimmten Zweck (Balakrushnan et al., 2016). Die Services werden dabei jeweils autonom in einem eigenen Prozess ausgeführt und kollaborieren in der Regel über leichtgewichtige Nachrichtenprotokolle (Fowler & Lewis, 2015). Hughes (2013) vergleicht Microservices-Architekturen mit der Anwendung des – in der objektorientierten Softwareentwicklung weit verbreiteten – SOLID-Prinzips[1] auf Architekturebene, welches bei Microservices auf einzelne Services statt auf Klassen angewendet werden kann. Tilkov und Wolff (2016) beschreiben Microservices als *„ein Konzept zum Modularisieren von Softwaresystemen“* (S. 100).

Die Modularisierung von Software-Systemen ist ein weit verbreiteter Ansatz, um große Systeme leichter handhaben zu können. Bei Microservices ist jedes Modul ein eigenes Programm und wird in einem eigenen Prozess ausgeführt. Dabei soll jedes Programm eine bestimmte Aufgabe erledigen und über eine universelle Schnittstelle mit anderen Programmen zusammenarbeiten (Wolff, 2015). Newman (2015) weist in diesem Zusammenhang auf das “Single Responsibility

[1] SOLID-Prinzip: siehe http://butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod

Principle" – das Erste der fünf SOLID-Prinzipien – hin, welches darauf abzielt, zusammengehörenden Code zusammen zu halten.

Wenn nicht gesondert ausgewiesen, werden im weiteren Verlauf dieser Arbeit die Begriffe „Microservice", „Service" und „Dienst" gleichgestellt und synonym verwendet.

2.2.2 Eigenschaften und Prinzipien

Aus der Literatur lassen sich verschiedene Charakteristiken ableiten, die auf eine Microservices-Architektur zutreffen. In Anlehnung an Balakrushnan et al. (2016) werden nachfolgend Eigenschaften erläutert, welche jedes Service beinhalten, sowie Prinzipien beschrieben, welche bei der Implementierung von Microservices berücksichtigt werden sollten.

Service Independence

Eine Microservices-Architektur teilt ein Software-System in eigenständige Services auf, wobei jedes Service unabhängig von anderen Services bereitgestellt werden kann. Es ist dabei möglich, mehrere Instanzen eines Service parallel auszuführen. Jede Instanz ist dabei unabhängig von anderen Instanzen des gleichen Service. Jedes Service stellt seine Funktionalität über eine definierte, öffentliche Schnittstelle zur Verfügung. (Newman, 2015)

Single Responsibility

Ein Microservice soll nur jeweils eine fachliche Aktivität abbilden. Dabei soll das Service die gesamte Geschäftslogik implementieren und bereitstellen, die für die Ausführung dieser Aktivität notwendig ist. (Tilkov & Wolff, 2016)

Self-Containment

Ein Microservice soll alle externen Ressourcen umfassen, welche für die Abbildung der fachlichen Aktivität notwendig sind. Zu solchen Ressourcen können unter anderem Datenquellen, Geschäftsregeln oder auch die Benutzerschnittstelle zählen. Dabei gilt es zu beachten, dass Abhängigkeiten auf Ressourcen vermieden werden sollen, welche nicht im Verantwortungsbereich des für das Service verantwortlichen Teams liegen. (Wolff, 2015)

Highly Decoupled

Services sollen entkoppelt von anderen Systemen lauffähig sein, um dadurch Abhängigkeiten zu anderen Systemen gering zu halten. Um diese Entkopplung zu erreichen, basiert die Kommunikation zwischen Services in der Regel auf bestehenden Protokollen zur Nachrichtenübertragung wie beispielsweise HTTP. Dabei sollte das „*smart endpoints and dumb pipes*" Prinzip berücksichtigt werden, welches aussagt, dass intelligente Endpunkte innerhalb der Services die vollständige Logik für die Transformation von Anfragen und Antworten beinhalten sollen. (Fowler & Lewis, 2015)

Highly Resilent

Ein auf einer Microservices-Architektur basiertes System muss so gestaltet werden, dass beim Ausfall eines oder mehrerer Services nicht das gesamte System beeinträchtigt wird. Dies bedingt, dass Microservices fehlertolerant agieren und auf Ausfälle anderer Services entsprechend reagieren müssen (Newman, 2015), dies wird als *Resilience* bezeichnet (Wolff, 2015). Um solche

fehlertoleranten Systeme zu erstellen, sollte bereits beim Entwurf einer Microservices-Architektur nach dem *Design for Failure* Prinzip vorgegangen werden (Fowler & Lewis, 2015).

2.2.3 Gründe für die Einführung von Microservices

Nachfolgend werden wesentliche Vorteile aufgezählt, die der Einsatz einer auf Microservices basierenden Architektur mit sich bringen kann.

Änderbarkeit

Gewachsene, monolithische Systeme oder voneinander abhängige Services sind schwer zu ändern. Änderungen können Auswirkungen auf große Teile des Systems haben und lange Deployment-Zyklen mit sich ziehen, da das gesamte System gebaut, getestet und anschließend bereitgestellt werden muss (Newman, 2015). Durch die Zerlegung eines komplexen Software-Systems in eine Vielzahl an Services, kann die Komplexität verringert und die Änderbarkeit einer Software verbessert werden (Wolff, 2015). Diese Reduktion der Komplexität auf einzelne Services kann sich sowohl auf die Entwicklung, das Management und auch den Betrieb einer Software positiv auswirken (Balakrushnan et al., 2016).

Technologiefreiheit

Aufgrund der Modularität von Microservices-Architekturen kann ein System mit unterschiedlichen Technologien entwickelt werden, um dadurch beispielsweise für jeden Anwendungszweck die geeignetste Plattform oder Programmiersprache zu verwenden (Fowler & Lewis, 2015). Dadurch können neue Technologien oder neue Versionen von bestehenden Technologien mit reduziertem Risiko in ein System eingeführt werden (Wolff, 2015).

Skalierung

Im Gegensatz zu monolithischen Systemen können bei einem System mit einer Microservices-Architektur einzelne Services skaliert werden, um dadurch die Verfügbarkeit von Bestandteilen des Systems zu erhöhen. Dies kann zu einer Kostenreduktion führen, da die Verfügbarkeit je nach Bedarf gezielt erhöht werden kann und weniger Ressourcen benötigt werden, wie wenn ein gesamtes System skaliert wird. (Newman, 2015)

Neben der Skalierung auf technischer Ebene bieten Microservices auch die Möglichkeit, Prozesse auf organisatorischer Ebene zu skalieren. Statt einem großen Team, das für eine gesamte Applikation zuständig ist, können kleinere Teams gebildet werden, welche für einzelne Microservices zuständig sind. Diese Aufteilung kann zu einer Skalierung von agilen Prozessen innerhalb der Teams führen. (Wolff, 2015)

Austauschbarkeit

Mit Hilfe von Microservices-Architekturen können Teile des Systems einzeln getauscht werden, ohne dass das Gesamtsystem neu bereitgestellt werden muss (Wolff, 2015). Kleinere Services können leichter geändert oder neu geschrieben werden und haben in der Regel kürzere Deployment-Zyklen, was die Reaktionsgeschwindigkeit bei Änderungen erhöhen (Newman, 2015) und zu kürzeren Reaktionszeiten auf geänderte Marktbedingungen führen kann (Balakrushnan et al., 2016).

Wolff (2015) weist darauf hin, dass durch die Verwendung von Microservices-Architekturen eine „*nachhaltige Entwicklung*“ (2015, S. 369) möglich ist, da es durch die Austauschbarkeit von einzelnen Services seltener zu einem Verfall der Architektur kommt und somit auch die Wartbarkeit langfristig erhalten bleibt.

Ausfallssicherheit / Robustheit

Im Gegensatz zu monolithischen Systemen wirkt sich ein Ausfall eines einzelnen Microservice nicht auf das gesamte System aus, sondern lediglich auf die Funktionalitäten des ausgefallenen Service (Tilkov & Wolff, 2016). Durch die Möglichkeit, mehrere Instanzen eines Microservice parallel zu betreiben, kann bei Ausfällen einer Instanz auf eine andere laufende Instanz des gleichen Service gewechselt und dadurch die Verfügbarkeit der Funktionalitäten wiederhergestellt werden (Balakrushnan et al., 2016).

Deployment / Continuous Delivery

Der Einsatz von Microservices unterstützt die Einführung einer *Continuous-Delivery-Pipeline*[2], um Änderungen am System schneller bereitstellen zu können. Ein einzelnes Microservice bereitzustellen und in Produktion zu bringen, ist in der Regel mit weniger Risiko verbunden, als dies bei einem monolithischen System der Fall ist, da hier immer das System als Ganzes neu bereitgestellt werden muss. (Wolff, 2015)

Organisation / Teams

Microservices-Architekturen ermöglichen die Bildung von kleinen, voneinander unabhängigen Entwicklerteams, welche die Zuständigkeit für ein oder mehrere Microservices übernehmen und für den gesamten Lebenszyklus dieser Services verantwortlich sind (Newman, 2015). Ein weit verbreiteter Ansatz ist dabei, bei der Zusammenstellung eines solchen Teams sowohl Angestellte aus Entwicklung als auch Betrieb zu wählen, da nur so die vollständige Verantwortung für den gesamten Lebenszyklus eines Service übernommen werden kann. Dieser organisatorische Ansatz wird als „DevOps“ bezeichnet (Wolff, 2015).

Wie in Kapitel 3.3.1 näher ausgeführt, sollte ein Microservice auf die Bereitstellung von Funktionalitäten für eine bestimmte Fachlichkeit ausgerichtet sein. Die fachlichen und technischen Teams arbeiten dabei in der Regel direkt miteinander. Dies kann zu einfacheren Kommunikationsstrukturen und einem besseren Verständnis zwischen fachlicher und technischer Seite führen. (Balakrushnan et al., 2016)

2.2.4 Nachteile von Microservices

Bei der Einführung von Microservices gilt es zu beachten, dass es sich dabei um verteilte Systeme handelt, welche mittels Nachrichtenübertragung über ein Netzwerk miteinander kommunizieren. Die Verteilung von Aufgaben auf mehrere Systeme erhöht die Komplexität, da das Zusammenspiel zwischen den Systemen koordiniert werden muss und die Kommunikation über ein Netzwerk fehleranfällig ist – grundsätzlich kann jeder Aufruf eines entfernten Systems

[2] siehe “Continuous Delivery“: https://martinfowler.com/bliki/ContinuousDelivery.html

fehlschlagen, weshalb entsprechende Routinen für die Fehlerbehandlung notwendig sind. Dies führt in der Regel zu einem organisatorischen Mehraufwand, der bei verteilten System zu berücksichtigen ist (ten Hompel & Schmidt, 2008). Weiters muss berücksichtigt werden, dass ein Aufruf eines Service über ein Netzwerk langsamer ist als ein lokaler Aufruf einer Funktionalität innerhalb des gleichen Prozesses (Fowler & Lewis, 2015). Daher sollte bereits beim Entwurf eines verteilten Systems darauf geachtet werden, die Trennung von Funktionalitäten auf einzelne Services so zu gestalten, dass diese möglichst wenig miteinander kommunizieren müssen (Wolff, 2015).

Die im vorherigen Abschnitt erwähnte Technologiefreiheit kann auch dazu führen, dass die Komplexität eines Systems erhöht wird, wenn dieses aus zu vielen unterschiedlichen Technologien besteht. Dies kann problematisch werden, wenn aus einer bestimmten Perspektive eine Gesamtsicht auf das System notwendig ist und diese ohne großen Aufwand nicht erstellt werden kann. (Wolff, 2015)

Weitere Einschränkungen bei der Entwicklung von Microservices ergeben sich hinsichtlich der Wiederverwendbarkeit von Quellcode und gemeinsam genutzten Programm-Bibliotheken. Das weit verbreitete „don't repeat yourself" Prinzip besagt, dass Redundanzen im Quellcode möglichst vermieden werden sollen, um dadurch ein bestimmtes Verhalten nur einmal zu implementieren. Bei Microservices kann dieser Ansatz jedoch dazu führen, dass Änderungen an einem gemeinsam genutzten Quellcode mehrere Services betrifft und diese erneut bereitgestellt werden müssen. Dies kann jedoch zu einer verstärkten Kopplung und Abhängigkeiten zwischen Teams führen und widerspricht dadurch dem Prinzip von unabhängigen Services. (Newman, 2015)

Eine Verteilung von fachlichen Anforderungen auf verschiedene Teams kann auch zu organisatorischen Änderungen führen, da Teams nach fachlichen Gesichtspunkten aufgeteilt werden müssen, um eine unabhängige Entwicklung zu gewährleisten. Da bei Microservices die Architektur des Gesamtsystems der Organisation gleichgesetzt wird, sind etwaige Fehler, die bei dieser Aufteilung gemacht werden, nachträglich nur schwer zu korrigieren. (Wolff, 2015)

Microservices wirken sich ebenfalls auf den Betrieb der Software aus, da eine Vielzahl von Services entsprechend viele Server oder virtuelle Maschinen erfordern und pro Service auch eine eigene Deployment-Pipeline benötigt wird (Fowler & Lewis, 2015). In weiterer Folge müssen alle Services überwacht werden, was bei vielen Services ein unter Umständen aufwändiges Monitoring erfordert, welches alle Systeme umfassen sollte (Dreifus, Leyking, & Loos, 2007).

Das Thema Datenmanagement zählt ebenfalls zu den wesentlichen Herausforderungen, die es bei der Einführung von Microservices zu beachten gilt. Unabhängige, voneinander losgelöste Services erfordern eine dezentrale Datenhaltung. Dies bedeutet, dass jedes Microservice eine eigene Datenbank beinhalten sollte, um die für die Abbildung der fachlichen Anforderungen benötigten Daten zu speichern, da ansonsten eine Integration auf Datenbankebene erfolgt und die Unabhängigkeit von Services verloren geht. (Fowler & Lewis, 2015)

2.3 Microservices-Architekturen

Traditionelle Software-Architekturen beziehen sich in der Regel auf ein einzelnes Programm, spätere Änderungen an der zuvor festgelegten Architektur sind meist mit viel Aufwand verbunden und haben oft Auswirkungen auf die gesamte Applikation (Steinacker, 2015). Da eine Microservices-Architektur gewöhnlich aus einer Vielzahl an einzelnen Services besteht – wobei jedes Service autonom agiert und ein eigenständiges Programm darstellt – wird die Architektur aus zwei Sichtweisen betrachtet und in eine Makro- sowie eine Mikro-Architektur unterteilt (Wolff, 2015).

In der Makro-Architektur wird das Zusammenspiel zwischen den einzelnen Services festgelegt, Entscheidungen, die in diesem Rahmen getroffen werden, haben Auswirkungen auf das gesamte System. Beim Entwurf müssen sowohl fachliche als auch technische Aspekte berücksichtigt sowie insbesondere die Art und Weise der Kommunikation zwischen den Services definiert werden (Wolff, 2015). Weiters ist zu berücksichtigen, dass – analog zu den eingangs beschriebenen traditionellen Software-Architekturen – nachträgliche Änderungen an der Makro-Architektur oft weitreichende Auswirkungen auf ein System haben können (Steinacker, 2015). Um eine hohe Flexibilität bei der Entwicklung von Microservices-basierten Systemen zu erreichen, sollten sich die Vorgaben der Makro-Architektur auf das Wesentliche beschränken (Freudl-Gierke, 2014) und lediglich Standards definiert werden, welche jedes Team bei der Entwicklung eines Microservices einhalten muss (Wolff, 2015).

Für die Mikro-Architektur ist hingegen jedes Team selbst zuständig und diese kann pro Microservice unterschiedlich sein. Dabei können grundsätzlich für jede Anforderung die geeignetsten Technologien verwendet werden, es muss nur darauf geachtet werden, dass durch die Technologiewahl die in der Makro-Architektur festgelegten Standards eingehalten werden. (Newman, 2015)

Welche Vorgaben auf Ebene der Makro- und Mikro-Architektur gemacht werden sollten, müsste von den jeweiligen Teams selbst entschieden werden und wird im Zuge dieser Arbeit nicht weiter untersucht. Für weiterführende Informationen wird an dieser Stelle auf entsprechenden Literatur verwiesen: Freudl-Gierke (2014), Steinacker (2015) sowie Wolff (2015) beschäftigen sich ausführlich damit.

In den nachfolgenden Unterkapiteln werden wesentliche Eigenschaften von Microservices-Architekturen im Detail betrachtet und diskutiert.

2.3.1 Größe von Microservices

Bezüglich der Größe eines Microservice finden sich in der Literatur verschiedene Ansichten und Diskussionen. Einige Definitionen beziehen sich dabei auf die Anzahl an Quellcodezeilen und weisen darauf hin, dass ein Microservice nicht aus mehr als 100 Zeilen Quellcode bestehen sollte (Rotem-Gal-Oz, 2014). Wolff (2015) stuft die Festlegung einer maximalen Anzahl an Quellcodezeilen als problematisch ein, da dieser Faktor von der jeweilig verwendeten Programmiersprache abhängig ist.

Nach Feathers (2014) sollte bei der Festlegung der geeigneten Größe das jeweilige Team berücksichtigt und der Umfang eines Service so gewählt werden, dass dieses von einer Handvoll Personen entwickelt und verwaltet werden kann. Newman (2015) weist darauf hin, dass Microservices nicht zu klein gestaltet werden sollen, da sich dadurch die Anzahl an Services erhöhen und in weiterer Folge zu einem höheren Aufwand bei der Bereitstellung führen kann. Weiters ist dabei zu beachten, dass eine Vielzahl an Services zu vermehrten Abhängigkeiten untereinander sowie einer Steigerung der Komplexität führen kann (Newman, 2015).

Richardson und Smith (2016) stellen diesbezüglich den eigentlichen Sinn von Microservices in den Vordergrund und beschreiben dies wie folgt:

> *"The goal of microservices is to sufficiently decompose the application in order to facilitate agile application development and deployment."* (Richardson & Smith, 2016, S. 9)

Wolff (2015) fasst die verschiedenen Ansätze zusammen und legt mittels Richtwerten eine obere sowie eine untere Grenze für Microservices fest. Bei diesen Grenzen sollten jeweils unterschiedliche Einflussfaktoren berücksichtigt werden, welche in Abbildung 2-2 grafisch dargestellt und nachfolgend erläutert werden. Diese Einflussfaktoren können sich sowohl auf die organisatorische als auch technische Seite von Microservices auswirken.

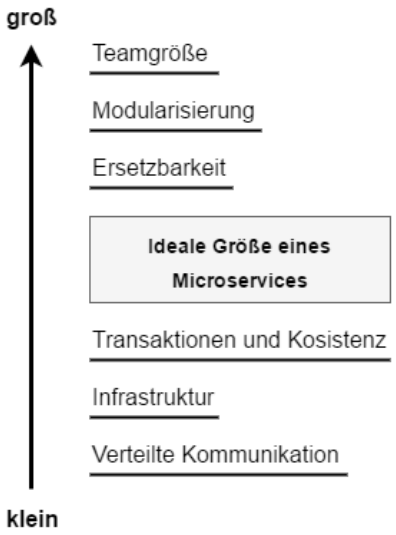

Abbildung 2-2: Einflussfaktoren für die Größe eines Microservice (Wolff, 2015)

Die obere Grenze umfasst dabei folgende drei Faktoren:

- *Teamgröße*: Ein Service muss vollständig von einem Team entwickelt werden können. Werden mehrere Teams für die Entwicklung benötigt, ist das ein Anzeichen dafür, dass das Service zu groß ist.
- *Modularisierung*: Je kleiner Services gestaltet werden, desto einfacher sind diese in der Regel zu verstehen. Ein Service sollte daher nur so groß sein, dass dieses von einem Entwickler des Teams verstanden und weiterentwickelt werden kann.

- *Ersetzbarkeit*: Ein Microservice sollte nur so groß sein, dass es einfach ersetzt werden kann. Um ein Service zu ersetzen, muss dieses auch verstanden werden, daher liegt dieser Faktor unterhalb der beiden zuvor genannten.

Die untere Grenze für die Größe eines Microservice kann sich nach Wolff (2015) aufgrund dieser Kriterien ergeben:

- *Transaktionen und Konsistenz*: Microservices sollten nicht so klein sein, dass dadurch verteilte Transaktionen erforderlich sind, um konsistente Daten zu gewährleisten. Auf diese Faktoren wird im nachfolgenden Abschnitt 3.3.3 noch im Detail eingegangen.
- *Infrastruktur*: Werden Services zu klein gestaltet, ergibt sich daraus eine höhere Anzahl an Artefakten, die allesamt einzeln bereitgestellt werden müssen, was in Folge zu einer Erhöhung des betrieblichen Aufwands führen kann.
- *Verteilte Kommunikation*: Je größer die Anzahl an einzelnen Microservices ist, aus denen sich ein System zusammensetzt, desto größer ist die Zunahme der verteilten Kommunikation. Dies kann zu den im vorhergehenden Kapitel 2.2.4 erwähnten Nachteilen bei der Einführung von Microservices führen.

In der Literatur finden sich weitere Kategorisierungen, um Services ihrer Größe nach einzuordnen, Nanoservices und Self-Contained Systems (SCS) werden hier in der Regel genannt. Als „Anti-Pattern“ werden Nanoservices gesehen. Dabei handelt es sich um zu klein gestaltete Services, deren Kosten den Nutzen übersteigen (Rotem-Gal-Oz, 2014). Tilkov (2014) beschreibt in diesem Zusammenhang SCS als eine Erweiterung zu Microservices. Diese Ansätze werden an dieser Stelle nicht weiter diskutiert und auf weiterführende Literatur wie beispielsweise Wolff (2015), Stenberg (2015) oder Rotem-Gal-Oz (2014) verwiesen.

Einen wesentlichen Einfluss auf die tatsächliche Größe eines Microservice hat die fachliche Trennung der Anforderungen eines Systems. In diesem Zusammenhang wird in der Literatur in der Regel auf das Modellierungsmuster Domain-Driven Design verwiesen, welches im Zuge dieser Arbeit im nachfolgenden Kapitel 3.3.1 diskutiert wird.

2.3.2 Kommunikation

Nach dem Gesetz von Conway werden beim Entwerfen eines Systems die Kommunikationsstrukturen der jeweiligen Organisation abgebildet (Newman, 2015). Entspricht die Aufteilung eines Systems in einzelne Microservices nicht der organisatorischen Aufteilung von Teams, kann ein erhöhter Kommunikationsbedarf entstehen und sich auf die Unabhängigkeit der einzelnen Teams – und in weiterer Folge auf die jeweiligen Microservices – negativ auswirken (Wolff, 2015).

Wie bereits in den vorherigen Abschnitten erläutert, ist eine Kommunikation zwischen Services fehleranfällig und kann zu erhöhten Laufzeiten führen. Um negative Auswirkungen auf die Laufzeiteigenschaften eines Systems zu vermeiden, sollte die Kommunikation zwischen Services möglichst reduziert und umfangreiche Nachrichten statt feingranularen Nachrichten übermittelt werden (Fowler & Lewis, 2015). Um auf die Fehleranfälligkeit bei der Kommunikation zu reagieren

und die benötigte Stabilität des Systems zu gewährleisten, müssen entsprechende Vorkehrungen getroffen werden (Newman, 2015). Dabei soll bei jeder Anfrage an ein entferntes Service ein Timeout festgelegt werden, welches den Aufruf nach einer festgelegten Zeit abbricht, um sich durch unendliches Warten nicht selbst zu blockieren (Newman, 2015). Fehlende Timeouts können in weiterer Folge zu einer Fehlerkette führen, da ein blockiertes Service auch abhängige Services beeinträchtigen kann (Nygard, 2007). Um die Stabilität bei verteilter Kommunikation sicherzustellen, finden sich in der Literatur weitere Muster, wobei an dieser Stelle auf die Ausführungen von Nygard (2007) sowie Wolff (2015) verwiesen wird. Nachfolgend wird beispielhaft das Circuit Breaker Muster beschrieben, da dieses im Umfeld von Microservices eine weite Verbreitung findet.

Ein als Circuit Breaker bezeichneter Mechanismus sorgt dafür, dass bei Nichtverfügbarkeit eines Systems oder ab einer gewissen Anzahl an rückgemeldeten Fehlern keine Anfragen mehr an dieses gesendet werden. Nach Ablauf einer definierten Zeit lässt der Circuit Breaker einzelne Anfragen an das externe System wieder zu und prüft anhand des Ergebnisses, ob dieses wieder ordnungsgemäß funktioniert. Kommt es zu erneuten Fehlern, wird weiterhin blockiert, andernfalls wird der Circuit Breaker zurückgesetzt und Anfragen werden wieder an das externe System weitergeleitet. Nachfolgende Abbildung 2-3 stellt die Zustände eines Circuit Breakers grafisch dar. Der Einsatz eines Circuit Breakers ermöglicht es, aufgerufene Systeme zu entlasten und sollte in Verbindung mit Timeouts eingesetzt werden. (Nygard, 2007)

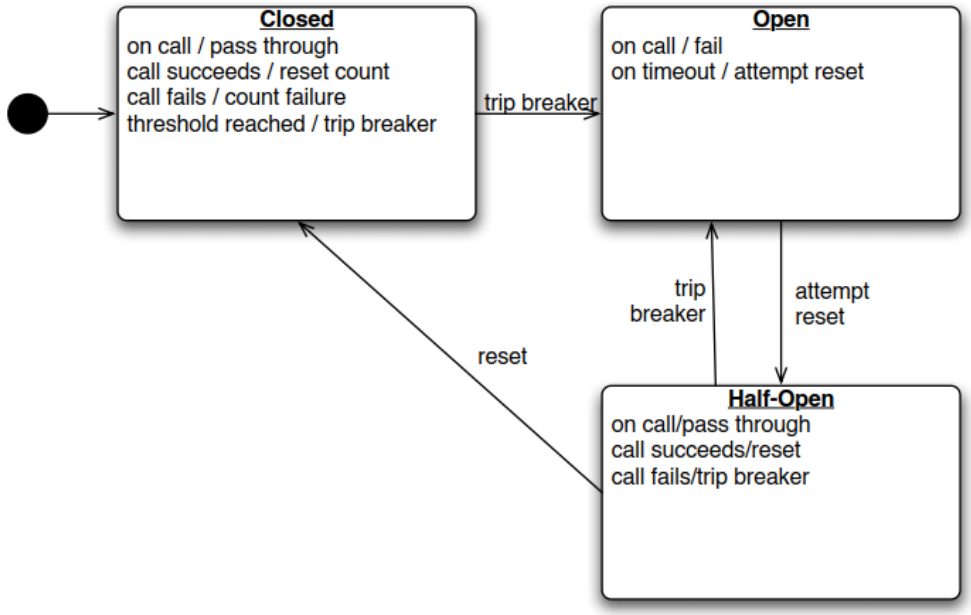

Abbildung 2-3: Circuit Breaker (Nygard, 2007)

Im Gegensatz zu einer SOA, bei welcher für die Integration von Services in der Regel das Prinzip der Orchestrierung angewendet wird, sollten bei Microservices-Architekturen die Services nicht von einer zentralen Komponente gesteuert werden, sondern dem Prinzip der Choreographie folgen. Dabei ist jedes Service selbst für die Erledigung seiner Aufgaben verantwortlich und ruft die dafür benötigten externen Services selbst auf. (Newman, 2015)

Diese direkte Kommunikation zwischen Services kann synchron oder asynchron erfolgen. Synchrone Aufrufe folgen dem Request/Response-Prinzip und warten, bis das aufgerufene System eine Antwort zurückliefert (Newman, 2015). Dabei ist zu berücksichtigen, dass synchrone Aufrufe zwischen Services zu einer starken Kopplung führen und das aufgerufene System verfügbar sein muss, um die Anfrage beantworten zu können (Flohre, 2015).

Bei einer asynchronen Kommunikation können Messaging-Systeme eingesetzt werden, welche die Verarbeitung von Nachrichten nach dem Queue- oder Topic-Verfahren bewerkstelligen (Wolff, 2015). Solche Systeme ermöglichen die Zustellung von Nachrichten, auch wenn der Empfänger zum Zeitpunkt des Versendens nicht verfügbar ist (Starke, 2015). Queues eignen sich beispielsweise als Punkt-zu-Punkt Verbindungen, wenn es sich bei der auszuführenden Aktion um eine länger dauernde Operation handelt (Flohre, 2015). Topics arbeiten nach dem Publish-Subscribe Verfahren und ermöglichen die Zustellung von Nachrichten an beliebige Empfänger, welche sich bei einem Topic zuvor registriert haben (Newman, 2015). Diese Arten von asynchroner Kommunikation werden auch bei ereignisgesteuerten Architekturen angewendet, welche zu einer losen Kopplung von Microservices führen (Wolff, 2015).

2.3.3 Datenmanagement

In vielen Unternehmen werden Datenbanken von mehreren Systemen gemeinsam genutzt, um Daten zentral zu verwalten und unter anderem die Konsistenz der Daten gewährleisten zu können. Diese gemeinsame Nutzung der Datenbasis sollte bei Microservices jedoch vermieden werden, weil dadurch eine Integration auf Datenbankebene stattfindet und dies zu einer engen Kopplung der Services führen kann. (Wolff, 2015)

Um den grundlegenden Prinzipien von Microservices-Architekturen – Unabhängigkeit und Autonomie – zu entsprechen, sollte jeder Microservice seine Daten selbst verwalten und diese in einem eigenen Datenbanksystem speichern. Dadurch entsteht ein dezentrales Datenmanagement, da jedes Microservice für seine Daten selbst verantwortlich ist und die Daten des Gesamtsystems auf mehrere Datenquellen verteilt sind. Diese Trennung von Daten ermöglicht die freie Wahl der Technologie für die Persistierung und wird als „Polyglot Persistence“ bezeichnet. (Fowler & Lewis, 2015)

Ein dezentrales Datenmanagement kann sich jedoch auf die Komplexität eines Systems auswirken und erweiterte Maßnahmen bei der Verarbeitung von Daten erfordern. Um sicherzustellen, dass Daten über mehrere Services konsistent sind, müssen verteilte Transaktionen durchgeführt werden, welche jedoch schwierig zu implementieren sein können. (Wolff, 2015)

Microservices machen daher in der Regel Abstriche bei der Konsistenz und verfolgen den Ansatz der transaktionslosen Koordination zwischen Services. Dabei werden nach der Speicherung der Daten innerhalb des Microservice Aktionen ausgeführt, welche externe Services über die Datenmanipulationen informieren. Diesen ist es dadurch möglich, kompensierende Operationen zur Wiederherstellung der Konsistenz auszuführen, was auch als *Eventual Consistency* bezeichnet wird. (Fowler & Lewis, 2015)

Um diese Koordination zwischen Services möglichst gering zu halten, sollte bereits bei der Aufteilung eines Systems in Microservices berücksichtigt werden, Services so zu teilen, dass diese alle relevanten Daten für die Abbildung der Fachlichkeit beinhalten (Newman, 2015). Da in der Regel jedoch auch andere Services über eine Änderung von Daten informiert werden müssen, können die geänderten Daten durch asynchrone Kommunikation übermittelt werden. Um eine lose Kopplung zu gewährleisten, sollten Daten dabei mittels Publish-Subscribe Verfahren versendet werden, auf welche jeder interessierte Service hören und entsprechende Aktionen mit diesen Daten ausführen kann (Flohre, 2015). Werden Daten über mehrere Services verteilt gespeichert, muss sichergestellt werden, dass nur das jeweilige Service die Hoheit über seine Daten besitzt und diese ändern darf (Wolff, 2015).

2.3.4 Cross-Cutting Concerns

In einem System, das auf einer Microservices-Architektur basiert, werden verschiedene Mechanismen benötigt, um service-übergreifende Tätigkeiten durchführen zu können. Weiters ergeben sich Vorgaben aus der Makro-Architektur, welche jedes Service einhalten muss und diese möglichst auf die gleiche Art und Weise ausführen soll. (Wolff, 2015)

Um bevorstehende Systembeeinträchtigungen frühzeitig erkennen sowie im Fehlerfall entsprechend reagieren zu können, muss ein Microservices-System ein umfangreiches Monitoring zur Verfügung stellen, welches betroffene Services aufspüren und zuständige Personen informieren kann (Goliath, 2016). Erfolgt die Kommunikation zwischen Systemen auf Basis von Ereignissen, sollte das System über ein Echtzeit-Monitoring verfügen, welches unter anderem auch die Verarbeitung von Ereignissen verfolgen kann (Fowler & Lewis, 2015). Hervorzuheben ist, dass jedes Microservice die vorgegebenen Standards der Makro-Architektur einhält und dem Monitoring-System die notwendigen Informationen über bereitgestellte Schnittstellen zur Verfügung stellt (Wolff, 2015). Damit im Fehlerfall die nachträgliche Ermittlung der Ursache möglich ist, sollten Microservices auch über ein gemeinsames Logging verfügen und Meldungen in einer einheitlichen Form protokollieren (Newman, 2015).

Zu den weiteren wichtigen Querschnittsthemen zählt Wolff (2015) unter anderem die Konfigurierbarkeit der Services, einen Service Discovery Mechanismus zum Auffinden einzelner Services sowie eine gemeinsame Vorgehensweise bei der Implementierung von Sicherheitsaspekten.

Um direkte Abhängigkeiten zwischen den Services zu vermeiden, sollte eine zentrale Service Discovery eingesetzt werden, bei welchem sich die Services registrieren können und die Kommunikation zwischen den Services von dieser koordiniert wird (Hughes, 2013). Diese zentrale Komponente kann auch als Load Balancer eingesetzt werden, um damit die Verteilung von Anfragen auf verschiedene Service-Instanzen zu steuern und dadurch die Skalierbarkeit von Services zu unterstützen (Richardson & Smith, 2016). Eine solche zentrale Service Discovery muss eine hohe Verfügbarkeit aufweisen, da eine Nichtverfügbarkeit dieser dazu führen kann, dass einzelne Services nicht mehr miteinander kommunizieren können und das ganze System ausfallen kann (Wolff, 2015).

Weiters sollten Microservices ein gemeinsames Sicherheitskonzept aufweisen, welches in der Makro-Architektur definiert und von allen Services einheitlich implementiert wird. Dabei sollte ein zentraler Server eingesetzt werden, welcher die Authentifizierung durchführt und ein Single-Sign On (SSO) Verfahren unterstützt. SSO hat den Vorteil, dass der Authentifizierungsvorgang nur einmalig durchgeführt werden muss und den Services in weiterer Folge ein Access-Token für die Kommunikation bereitgestellt wird. Dabei empfiehlt es sich, standardisierte Protokolle wie beispielsweise OAuth2, OpenId Connect oder JSON Web Tokens (JWT) zu verwenden. (Newman, 2015) Wolff (2015) empfiehlt, die Autorisierung und die damit einhergehende Prüfung der Zugriffsrechte in den Microservices selbst durchzuführen, um notwendige Änderungen an der Zugriffslogik schnell umsetzen zu können und nicht von einer zentralen Lösung abhängig zu sein.

2.4 Zusammenfassung

Zu Beginn dieses Kapitels wurden verschiedene Begriffe der Softwareentwicklung definiert und abgegrenzt. Hervorzuheben sind dabei die Begriffe „Monolith“ und „Service-orientierte Architekturen“, welche wesentliche Begrifflichkeiten für das Thema dieser Arbeit darstellen.

Ein Monolith beinhaltet in der Regel verschiedene Module, Komponenten und Services und fasst diese zu einem ausführbaren Programm zusammen. Änderungen an einzelnen Teilen des Systems bedingen eine vollständige Neuerstellung der Applikation, welche anschließend alle notwendigen Phasen für die Inbetriebnahme als Ganzes durchlaufen muss. Diese Vorgehensweise kann sich auf die Flexibilität und Kosten negativ auswirken.

SOA gilt als ein Basisprinzip für verteilte Systeme, welches zu flexiblen Software-Architekturen führen soll und dabei auf die Wiederverwendbarkeit von Diensten fokussiert. Grundlage dafür sind verschiedene Designprinzipien, welche notwendige Eigenschaften von Services beschreiben und auch als Ausgangsbasis für Microservices-Architekturen dient. In der Literatur findet sich keine eindeutige Abgrenzung zwischen SOA und Microservices. Nach Wolff (2015) zielen Microservices eher auf den Einsatz in einzelnen Applikationen ab, wohingegen SOA das ganze Unternehmen betrachtet und dabei das Zusammenspiel der verschiedenen in der IT eingesetzten Systeme in den Vordergrund stellt.

Im weiteren Verlauf dieses Kapitels wurde der Microservices-Ansatz diskutiert und dessen Vor- und Nachteile beschrieben. Dies wird in den folgenden Absätzen zusammengefasst.

Das Microservices-Paradigma stützt sich neben SOA auf weitere Prinzipien der Softwareentwicklung wie beispielsweise Domain-Driven Design und *Continuous Delivery*. Eine Anwendung wird aus einer Vielzahl an voneinander unabhängigen Services zusammengestellt, welche über definierte Schnittstellen miteinander kommunizieren. Jedes Service sollte eigenständig lauffähig sein, gesondert bereitgestellt und parallel in mehreren Instanzen ausgeführt werden können. Weiters sollte ein Service nur jeweils eine fachliche Aktivität abbilden und alle dafür benötigten Ressourcen umfassen. Beim Entwurf von Microservices sollte nach dem *Design for Failure* Prinzip vorgegangen werden, damit bei Ausfällen eines oder mehrerer Services das Gesamtsystem nicht beeinträchtigt wird.

Aus der Eigenständigkeit der einzelnen Services ergeben sich verschiedene Vorteile. Services können unabhängig voneinander entwickelt, bereitgestellt und betrieben werden, woraus sich eine freie Technologiewahl ergibt und somit für jeden Anwendungsfall die am besten geeignete Technologie verwendet werden kann. Der eigenständige Betrieb der Services kann zu einer Kostenreduktion führen, weil dadurch Teile eines Systems gezielt skaliert und somit weniger Ressourcen benötigt werden. Weiters verteilt sich die fachliche Komplexität einer Anwendung auf die einzelnen Services, womit diese in der Regel leichter zu verstehen sind, sich dadurch die Änderbarkeit und Austauschbarkeit einzelner Teile eines Systems erhöhen und es seltener zu einer Architekturerosion kommt. Ein Microservice sollte in der Verantwortlichkeit von einem einzelnen Team sein und vollständig von diesem entwickelt sowie betrieben werden, um einfachere Kommunikationsstrukturen zu schaffen und ein besseres Verständnis zwischen fachlichen und technischen Teams herbeizuführen.

Nachteile von Microservices ergeben sich vor allem dadurch, dass es sich bei diesem Architekturansatz um ein verteiltes System handelt. Die Kommunikation über ein Netzwerk ist fehleranfällig und langsamer als ein lokaler Methodenaufruf innerhalb des gleichen Prozesses. Services müssen mit dieser Fehleranfälligkeit umgehen können und sollten die Kommunikation untereinander möglichst geringhalten. Weiters kann die freie Wahl einer Technologie zu einer unerwünschten Technologievielfalt innerhalb eines Systems führen und es werden zusätzlich Werkzeuge benötigt, um das Zusammenspiel der einzelnen Dienste zu koordinieren und überwachen. Die vollständige Abbildung einer bestimmten Fachlichkeit in einem Service kann neben Umstellungen im Datenmanagement auch organisatorische Änderungen zur Folge haben. Die fachlich benötigten Daten sollten innerhalb eines Service gespeichert werden, was in weiterer Folge dazu führen kann, dass Daten redundant gehalten werden müssen und ein verteiltes Datenmanagement benötigt wird, welches die Komplexität des Gesamtsystems in der Regel erhöht.

Im Anschluss wurden Microservices-Architekturen untersucht, bei welchen in einer Makro-Architektur das Zusammenspiel zwischen Services geregelt und jedem Service innerhalb der eigenen Mikro-Architektur freie Wahl hinsichtlich der verwendeten Technologien und Architekturmuster ermöglicht werden sollten. Die geeignete Größe eines Microservice ist nicht fest definiert und wird in der Regel als zu groß angesehen, wenn die Entwicklung nicht mehr in einem einzelnen Team möglich ist. Services sollten jedoch nicht zu klein gestaltet werden, da dies sonst zu verteilten Transaktionen führen, den betrieblichen Aufwand steigern sowie die Kommunikation zwischen den Services erhöhen kann. Neben diesen Faktoren hat vor allem die fachliche Trennung des Systems einen wesentlichen Einfluss auf die tatsächliche Größe eines Service.

Abschließend wurden Querschnittsthemen betrachtet, zu welchen unter anderem das betriebliche Monitoring oder ein gemeinsames Sicherheitskonzept zählen und in einem auf Microservices basierten System berücksichtigt werden müssen. Hervorzuheben ist, dass diese vor allem die technische Komplexität des Gesamtsystems erhöhen und bei der Einführung von Microservices nicht außer Acht gelassen werden sollen.

3 ERWEITERUNG UND MODERNISIERUNG VON LEGACY-SYSTEMEN

“If you find yourself in a hole, stop digging.”
Denis Healy (1988)

Das eingehende Zitat wird als “Law of Holes” bezeichnet und soll in Bezug auf die Entwicklung einer Software ausdrücken, dass wenn diese nur mehr mit hohem Aufwand und tiefgreifenden Änderungen angepasst und erweitert werden kann, man damit aufhören und stattdessen alternative Lösungen suchen sollte, um ein System um neue Funktionalitäten zu erweitern. (Richardson & Smith, 2016)

Aus diesem Grund werden im vorliegenden Kapitel Möglichkeiten zur Erweiterung und Modernisierung von monolithischen Altsystemen durch die Einführung von Microservices untersucht. Einleitend wird der Prozess der Softwareevolution sowie Methoden des Software-Reengineerings diskutiert und deren Anwendbarkeit auf Microservices betrachtet. Anschließend werden Strategien für eine schrittweise Ablöse von monolithischen Systemen durch Microservices vorgestellt. Im weiteren Verlauf dieses Kapitels werden Vorgehensweisen beschrieben, die sich für die Auffindung von Schnittpunkten in bestehenden Systemen eignen. Dabei werden neben der fachlichen Aufteilung eines Systems auch Wege beschrieben, die eine Trennung auf Ebene der Datenbank sowie der Benutzerschnittstelle ermöglichen.

3.1 Softwareevolution

Unternehmen investieren in der Regel viel Geld in Software-Systeme und sind heute oftmals abhängig von der Funktionstüchtigkeit dieser. Damit ein Software-System auf Dauer nützlich bleibt, sollte es regelmäßig gewartet und weiterentwickelt werden, um auf Änderungen im Unternehmen reagieren und neue Anforderungen erfüllen zu können. Außerdem müssen Fehlverhalten einer Anwendung korrigiert und gegebenenfalls Anpassungen aufgrund geänderter Hardware und Software durchgeführt werden. Aus diesen Gründen ist die Softwareentwicklung Teil des gesamten Lebenszyklus eines Systems. (Sommerville, 2012)

Der Lebenszyklus eines Software-Systems teilt sich nach Sneed (2012) in drei Phasen, der *Entstehungsphase*, in welcher die Software entwickelt und in Betrieb genommen wird, der *Evolutionsphase*, in welcher das System gewartet und weiterentwickelt wird, sowie der *Erlösungsphase*, in welcher das System nicht mehr benötigt und ausgemustert wird.

In der Evolutionsphase – in welcher sich die Software im Betrieb befindet – können Fehler auftreten, neue Anforderungen entstehen oder die Umgebung der Software könnte sich ändern (Sommerville, 2012). Die Software sollte regelmäßig gewartet und weiterentwickelt werden, um nicht zu altern und somit weiterhin für den vorgesehenen Zweck eingesetzt werden zu können (Balzert, 2011).

3.1.1 Softwarewartung

Die Aktivitäten in der Evolutionsphase können in Wartungs- und Pflegetätigkeiten gruppiert werden. Wartungsaktivitäten befassen sich mit der Stabilisierung des Systems, der Korrektur von auftretenden Fehlern sowie der Verbesserung der Leistungsfähigkeit der Software. Zu den Pflegeaktivitäten zählen notwendige Anpassungen an eine neue Systemumgebung sowie die Erweiterung des Systems um neue Funktionalitäten. (Balzert, 2011)

Im weiteren Verlauf dieser Arbeit werden die zuvor genannten Pflegeaktivitäten näher betrachtet. Da sich diese Unterteilung in Wartung und Pflege jedoch nicht durchgängig in der Literatur findet, wird im weiteren Verlauf dieser Arbeit darauf verzichtet und allgemein der Begriff Wartung verwendet.

Für gewöhnlich kann davon ausgegangen werden, dass der Aufwand für die Tätigkeiten in der Evolutionsphase größer als der Entwicklungsaufwand ist (Vogel et al., 2009) und mit steigender Lebensdauer und Umfang eines Systems zunimmt (Balzert, 2011). Nach Sommerville (2012) fallen zwei Drittel der Kosten für Wartungstätigkeiten und lediglich ein Drittel für die Neuentwicklung von Systemen an. 65% des benötigten Wartungsaufwands fallen dabei in der Regel für das Hinzufügen neuer und Modifizieren bestehender Funktionalitäten an, der restliche Aufwand verteilt sich recht gleichmäßig auf die Behebung von Fehlern und Anpassungen des Systems auf eine geänderte Umgebung (Sommerville, 2012). Diese Verteilung des Wartungsaufwands wird in nachfolgender Abbildung 3-1 dargestellt.

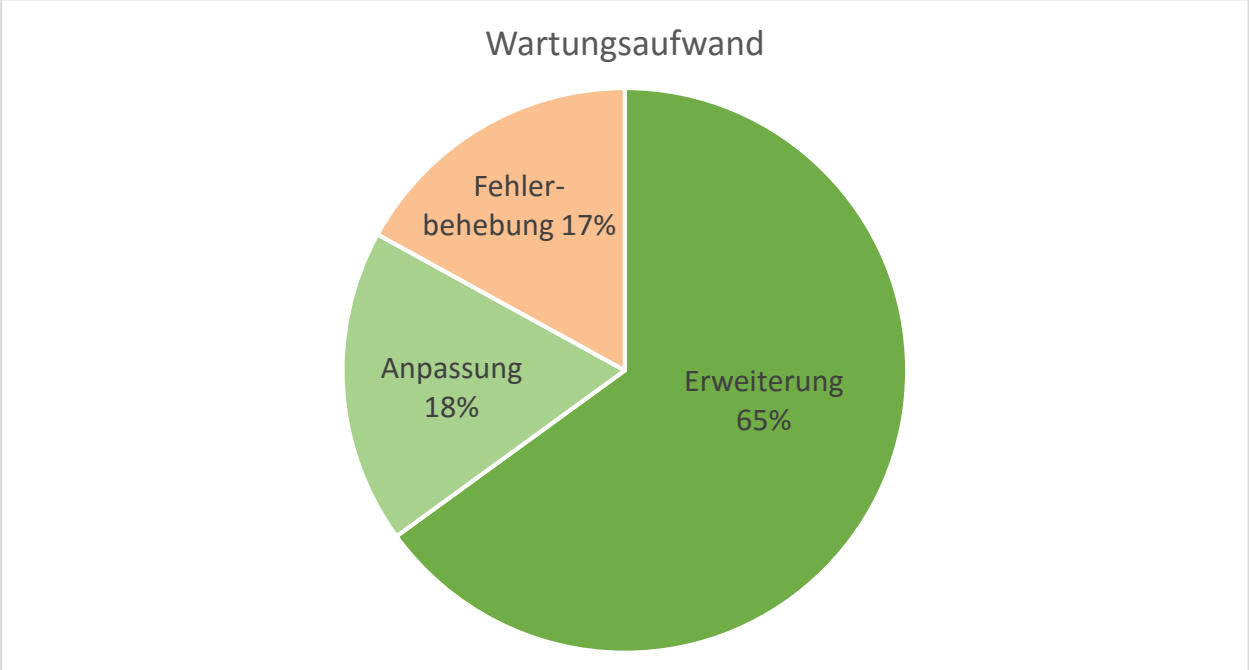

Abbildung 3-1: Verteilung des Wartungsaufwands (in Anlehnung an Sommerville, 2012)

Erfahrungsgemäß können die Kosten für die Wartung reduziert werden, wenn bei der Entwicklung eines Systems darauf geachtet wird, ein wartbares System zu entwerfen (Sommerville, 2012). Je älter jedoch ein System wird, desto aufwändiger gestaltet sich in der Regel der Wartungsaufwand, was in weiterer Folge zu einer zusätzlichen Steigerung der Kosten führen kann (Balzert, 2011).

Um dem entgegenzuwirken, sollte eine Strategie entwickelt werden, wie mit dem Altsystem in Zukunft umgegangen wird, um damit die Frage zu beantworten, ob ein System weiterhin gewartet oder durch ein neues Software-System ersetzt werden soll (Balzert, 2011).

Aus der Literatur lassen sich verschiedene Strategien ableiten, um Altsysteme auf eine technologisch aktuelle Basis zu überführen. Für die Erneuerung beziehungsweise Modernisierung eines Systems gibt es nach Masak (2006) grundsätzlich die zwei Ansätze Big-Bang-Migration und inkrementelle Vorgehensweise, beide werden in der nachfolgenden Tabelle 3-1 gegenübergestellt:

	Big-Bang	**inkrementell**
Geeignet für	Nicht zerlegbare Systeme	Zerlegbare Systeme
Risiko	Sehr hoch	Kontrollierbar
Risikofall	Gesamtprojekt	Teilprojekt
Ziel	Sofort	Inkrementell
Planbarkeit	Termin	Optimistisch
Gesamtdauer	Niedriger	Höher
Kosten	Niedriger	Höher

Tabelle 3-1: Gegenüberstellung Big-Bang und inkrementelle Migration (Masak, 2006)

Da bei einer inkrementellen Vorgehensweise gewöhnlicherweise ein geringeres Risiko besteht, ist diese der Big-Bang-Migration zu bevorzugen. Handelt es sich bei dem zu migrierenden Altsystem jedoch um ein nicht zerlegbares System, ist eine inkrementelle Ablöse nicht möglich. (Masak, 2006)

Nach Sneed und Sneed (2003) bieten sich für eine Big-Bang-Migration unter anderem folgende Alternativen an:

- *Neuentwicklung:*

  Ein bestehendes Software-System zu einem festgelegten Zeitpunkt vollständig durch eine neue Anwendung zu ersetzen, wird als Neuentwicklung bezeichnet. Dabei gilt es zu beachten, dass eine vollständige Neuentwicklung in der Regel eine lange Zeit in Anspruch nimmt, kostenintensiv ist und daher ein hohes Risiko darstellen kann (Byars, 2013; Sneed & Sneed, 2003). In weiterer Folge kann dies dazu führen, dass ein Unternehmen während der Dauer der Entwicklung nur begrenzt handlungsfähig ist, da bis zur Fertigstellung des neuen Systems kaum auf Änderungen im Unternehmen reagiert werden kann (Joel, 2000). Dies findet sich auch bei Sommerville (2012), welcher ebenfalls hervorstreicht, dass eine Neuentwicklung zu geschäftskritischen Verzögerungen führen kann und bei alternativen Wegen der Überführung einer Software normalerweise ein verringertes Risiko besteht.

- *Kaufen:*

  Durch den Kauf einer Standard-Software ist ebenfalls eine Ablöse eines Altsystems möglich. Hervorzuheben ist, dass die zugekauften Standard-Softwarekomponenten in der Regel allgemeine Prozessabläufe abbilden und möglicherweise hohe Aufwände anfallen

können, um die Software an die Anforderungen und Prozesse eines Unternehmens entsprechend anzupassen. (Sneed & Sneed, 2003)

Formen der inkrementellen Vorgehensweise werden im nächsten Abschnitt diskutiert und dabei verschiedene Migrationsstrategien betrachtet. Grundsätzlich sind für die Entscheidung einer geeigneten Alternative zwei Faktoren ausschlaggebend, die Wirtschaftlichkeit der Überführung des Altsystems sowie dessen verbleibende Einsatzfähigkeit. (Balzert, 2011)

3.1.2 Migrationsstrategien

Nach Masak (2006) spricht man von einer Migration, wenn sich grundlegende Bestandteile eines Systems wie beispielsweise Hardware, das zugrunde liegende Betriebssystem oder die Architektur der Software selbst ändern. Werden im Zuge der Wartung das System selbst oder Teile davon in eine neue technische Umgebung überführt, wird dies als Migration bezeichnet.

Dabei werden Funktionalitäten des Altsystems in das Zielsystem transformiert, wodurch eine nachträgliche Prüfung der Migration in Form von Regressionstests ermöglicht wird (Gimnich & Winter, 2005). Eine Migration sollte dabei schrittweise erfolgen und Funktionalitäten der Reihe nach überführt werden, um dadurch kurzfristige Rückmeldungen über den Erfolg der Transformation zu erhalten (Starke, 2015). Diese Vorgehensweise zur Überführung von Funktionalitäten in eine neue Umgebung eignet sich auch für die Einführung von Microservices (Wolff, 2015).

Laut Gimnich und Winter (2005) ergeben sich aus der zu migrierenden Legacy-Anwendung nicht-funktionale Anforderungen an das Zielsystem, da diese in der Regel im Altsystem nicht ausreichend erfüllt werden und daher Auslöser für eine Migration sein können.

Die Migration eines Software-Systems betrifft gewöhnlicherweise nicht nur die Software selbst, sondern kann sich auch auf die Umgebung der Software – wie zum Beispiel die verwendete Hardware oder das der Software zugrundeliegende Betriebssystem – auswirken beziehungsweise von dieser ausgelöst werden. Hinsichtlich der Software selbst kann sich eine Migration auf die Daten und Datenstrukturen, die ausführbare Programmlogik oder die Benutzerschnittstelle auswirken und bedingen sich in der Regel gegenseitig. (Gimnich & Winter, 2005)

Um eine Migration durchzuführen, bieten sich die drei Migrationsstrategien Kapselung, Konvertierung und Reengineering an, welche nachfolgend erläutert werden.

Kapselung

Bei der Kapselung – welche auch als „Wrapping“ bezeichnet wird – werden neue Schnittstellen in bestehende Systeme oder Systemteile eingeführt, um diese dadurch zu umschließen und deren Funktionalitäten für neue Systeme zur Verfügung zu stellen (Vogel et al., 2009). Das Altsystem wird dabei nicht verändert und behält den vorhandenen Zustand (Masak, 2006). Diese Schnittstellen müssen so gestaltet werden, dass der bestehende Code in einer anderen Umgebung verwendet werden kann (Sneed, 2003).

Ein weit verbreiteter Anwendungsfall für diese Vorgehensweise ist die Einbindung eines Legacy-Systems in eine Web-Applikation (Sneed, 2003). Dabei wird am Server der Zugriff auf die Funktionen der Altanwendung mittels der eingeführten Schnittstellen ausgeführt und dem User eine neue, webbasierte Oberfläche zur Verfügung gestellt. Dieser Ansatz wird auch als „Refronting“ bezeichnet. (Masak, 2006)

Die Kapselung eignet sich vor allem für die schrittweise Ablöse eines Systems, da dadurch auf einfache Weise eine Integration eines Altsystems beziehungsweise Teile davon in ein moderneres System ermöglicht wird (Vogel et al., 2009). In der Regel kann davon ausgegangen werden, dass der bestehende Code des Legacy-Systems ausreichend getestet ist und damit ein funktionierender Baustein integriert wird (Masak, 2006). Kapselung kann auch als Zwischenschritt oder Übergangslösung bei der Modernisierung eines Systems verwendet werden, um bestehende Komponenten in ein neues System einzubinden (Bisbal et al., 1999).

Konvertierung

Als Konvertierung wird der Ansatz bezeichnet, den Quellcode eines Systems an eine neue Umgebung anzupassen ohne dabei dessen Funktionalität zu ändern. Eine Konvertierung muss beispielsweise durchgeführt werden, wenn sich das der Software zugrundeliegende Betriebssystem ändert und diese Änderungen nicht rückwärtskompatibel sind. Weitere Einsatzzwecke der Konvertierung sind unter anderem die Überführung des Quellcodes in eine neue Programmiersprache oder notwendige Anpassungen einer Software an ein neues Datenbanksystem. Dabei werden gewöhnlicherweise Werkzeuge eingesetzt, die diese Konvertierung möglichst automatisiert durchführen. (Sneed, 2003)

Ein System muss vor und nach der Konvertierung die gleichen Ergebnisse liefern und sollte daher im Anschluss ausführlich getestet werden, um sicherzustellen, dass sich die Programmfunktionalität nicht geändert hat. (Gimnich & Winter, 2005)

Reengineering

Das Reengineering zielt darauf ab, ein System unter Beibehaltung der fachlichen Funktionalitäten ganz oder teilweise neu zu entwickeln. Dabei soll die Struktur einer Software verbessert und dadurch die Wartbarkeit dieser erhöht werden. (Masak, 2006)

Diese Strategie findet eine weite Verbreitung bei der Erweiterung und Modernisierung von Legacy-Systemen und wird im nachfolgenden Abschnitt im Detail diskutiert.

3.1.3 Software-Reengineering

Der Begriff Software-Reengineering beschreibt alle Tätigkeiten, die im Zuge der Erweiterung und Modernisierung einer Software ausgeführt werden (Vogel et al., 2009). Dabei handelt es sich um die zuvor diskutierten Wartungstätigkeiten, welche ein System um neue Funktionalitäten erweitern oder bestehende Funktionalitäten anpassen (Balzert, 2011).

Beim Reengineering wird das Ziel verfolgt, die Komplexität des Systems so zu verringern, dass es weiterhin genutzt werden kann und die Kosten für weitere Anpassungen im Rahmen bleiben (Demeyer, Ducasse, & Nierstrasz, 2009).

Um ein System im Zuge des Reengineerings entsprechend optimieren und erweitern zu können, muss dessen Funktionsweise zuvor verstanden werden. Da bei Legacy-Systemen oftmals das Wissen der ursprünglichen Entwickler fehlt – da diese beispielsweise nicht mehr im Unternehmen tätig sind – müssen Wege gefunden werden, um den Aufbau und die Funktionsweise des Systems zu verstehen (Vogel et al., 2009). Diese Tätigkeiten werden als Reverse Engineering bezeichnet und sind für gewöhnlich die Grundlage für die weiteren Schritte im Reengineering einer Software (Demeyer et al., 2009).

Demeyer et al. (2009) beschreiben diesen Prozess des Reverse Engineerings und dessen Bedeutung für das Reengineering einer Software wie folgt:

> *"Basically you can analyze the source code, run the system, and interview users and developers to build a model of the legacy system. Then you must determine what are the obstacles to further progress, and fix them. This is the essence of reengineering, which seeks to transform a legacy system into the system you would have built if you had the luxury of hindsight and could have known all the new requirements that you know today."* (Demeyer et al., 2009, S. 3)

Wurde ein System im Zuge des Reverse Engineerings analysiert und dokumentiert, wird in der nächsten Phase – der Restrukturierung – die Struktur des Systems geändert und an das gewünschte Zielsystem angepasst (Starke, 2015). Die Restrukturierung erfolgt dabei jeweils auf der gleichen Abstraktionsebene und transformiert eine Repräsentation des Altsystems in die gewünschte Repräsentation des geplanten neuen Systems (Masak, 2006).

Abschließend werden im Zuge des Forward Engineerings die gewünschten Änderungen implementiert. Beim Forward Engineering wird dabei analog zu einer Neuimplementierung vorgegangen. (Masak, 2006; Starke, 2015)

Der zuvor erwähnte dreistufige Prozess des Reengineerings wird in Abbildung 3-2 dargestellt und zeigt diesen in Form der Reengineeringpyramide nach Byrne (Masak, 2006).

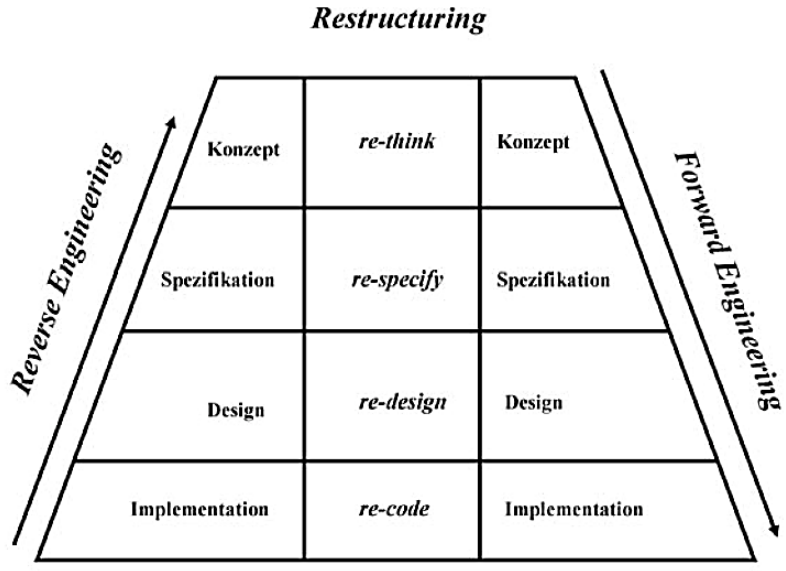

Abbildung 3-2: Reengineeringpyramide nach Byrne (Masak, 2006)

Vogel et al. (2009) sowie Starke (2015) weisen in diesem Zusammenhang darauf hin, dass Werkzeuge des Reengineerings nicht nur für Legacy-Systeme verwendet werden können. Diese eignen sich grundsätzlich für jede Software, um durch inkrementelle Restrukturierungsmaßnahmen die Architektur eines Systems zu verbessern.

Um Software systematisch unter Berücksichtigung des Kosten-Nutzen-Verhältnisses zu verbessern, soll nach Starke (2016) ein iterativer Ansatz gewählt werden, welcher sich aus drei Phasen zusammensetzt und in jeder Phase Informationen über zu optimierende Teile eines Systems sammelt.

In der ersten Phase erfolgt eine Analyse des Systems, aus welcher eine Liste mit auftretenden Problemen („Issue List“) hervorgeht. Anschließend werden in der zweiten Phase die gefundenen Probleme einzeln bewertet und dabei jeweils die durch das Problem verursachten Kosten betrachtet. In der dritten und letzten Phase einer Iteration werden Maßnahmen – welche die Ursachen der Probleme lösen sollen – abgeleitet, in einer Liste („Improvement Backlog“) gesammelt und ebenfalls die Kosten für deren Durchführung ermittelt. Die Ergebnisse einer Iteration sollen anschließend herangezogen werden, um eine Priorisierung herbeizuführen und die auszuführenden Tätigkeiten zu planen. Diese Vorgehensweise kann je nach Bedarf wiederholt werden und soll eine Kosten-Nutzen-Analyse der durchzuführenden Änderungen ermöglichen (Starke, 2016)

Die Zusammenhänge zwischen auftretenden Problemen, Maßnahmen zur Behebung sowie die dadurch verursachten Kosten werden in Abbildung 3-3 dargestellt.

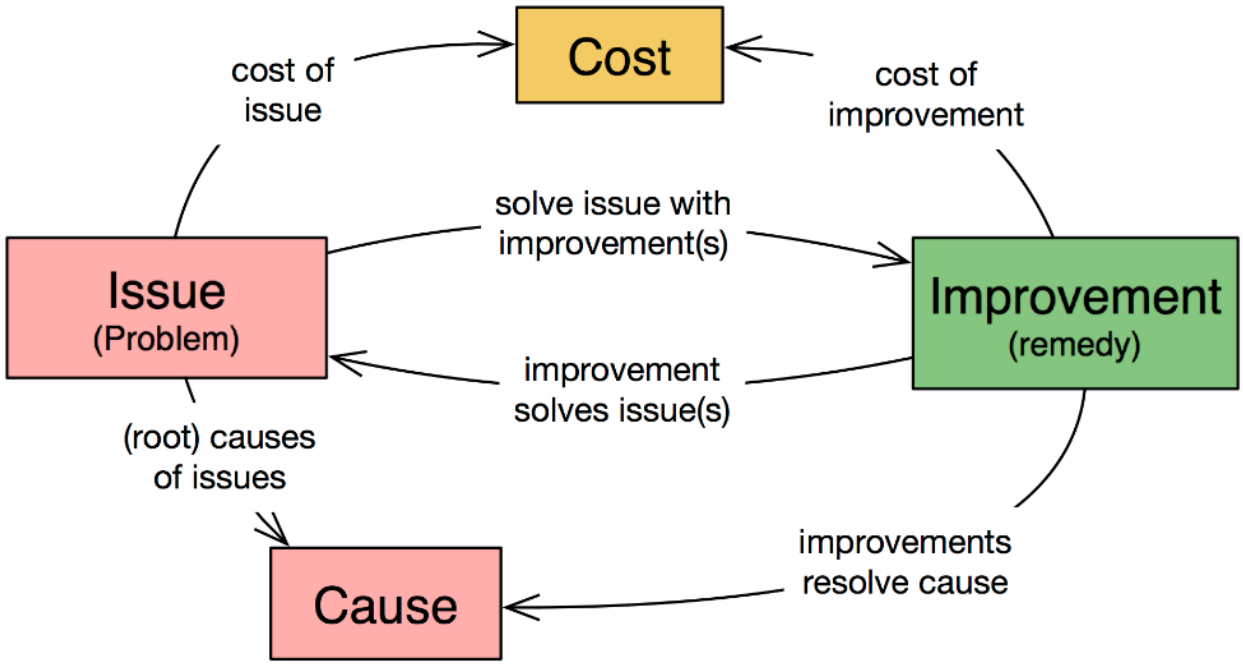

Abbildung 3-3: Zusammenhänge zwischen Problemen, Maßnahmen und Kosten (Starke, 2016)

Dieses Modell zur systematischen Verbesserung einer Software wird als *Architecture Improvement Method*[3] bezeichnet und steht als Open Source zur Verfügung.

3.2 Schrittweise Erweiterung und Ablöse von Altsystemen

Um im Zuge der im vorhergehenden Kapitel betrachteten Softwareevolution ein System zu optimieren und dieses schrittweise hin zum gewünschten Zielsystem zu transformieren, haben sich in der Praxis verschiedene Vorgehensweisen etabliert, welche in diesem Abschnitt vorgestellt werden.

Ausschlaggebend für die Wahl einer geeigneten Methode ist die Architektur des Legacy-Systems (Bisbal et al., 1999) und ob das System – wie bereits in Abschnitt 3.1.1 diskutiert – zerlegbar ist oder nicht. Auch ohne die internen Strukturen eines Systems zu kennen, können Microservices dessen Funktionalitäten erweitern und ändern, indem sie beispielsweise an der Benutzerschnittstelle des Systems angesiedelt werden (Wolff, 2015).

Nachfolgend werden in der Literatur häufig diskutierte Strategien und Methoden für die Transformation beschrieben.

3.2.1 Chicken-Little-Strategie

Die *Chicken-Little-Strategie* ermöglicht eine schrittweise Ablöse eines Altsystems und beschreibt eine aus mehreren Schritten bestehende Vorgehensweise, um ein System als Ganzes oder Teile davon inkrementell abzulösen und in eine modernere Umgebung zu überführen. Während der Migration werden das abzulösende sowie neue System parallel betrieben, die Kommunikation zwischen beiden Systemen wird mittels Gateways gesteuert. (Bisbal et al., 1999)

Zu Beginn wird mit Hilfe von Reverse-Engineering Techniken das Altsystem analysiert und strukturiert. Dabei werden – soweit möglich – Abhängigkeiten zwischen einzelnen Modulen entfernt und durch klar definierte Schnittstellen ersetzt. Als nächste Schritte werden die Schnittstellen des Zielsystems definiert, das Zielsystem sowie dessen Datenbank erstellt und diese in der Zielumgebung installiert. (Bisbal et al., 1999)

Anschließend werden Gateways entwickelt, welche je nach gewählter Vorgehensweise die Kommunikation des Zielsystems mit der Datenbank des Altsystems oder des Altsystems mit der Datenbank des Zielsystems ermöglichen. Dabei werden nach Kaps (2016) folgende drei Vorgehensweisen unterschieden:

- *Database-First*: Bei der *Database-First*-Strategie wird die Datenbasis des Altsystems übernommen und in die Zieldatenbank überführt. Anschließend muss ein *Forward-Gateway* erstellt werden, welches Anfragen aus dem Altsystem transformiert und an die Zieldatenbank weiterleitet.

[3] Architecture Improvement Method: http://aim42.org/

- *Database-Last*: Die *Database-Last*-Vorgehensweise setzt den Fokus auf die Entwicklung des Zielsystems unter Verwendung der Datenbasis der zu migrierenden Anwendung. Bei dieser Methode muss ein *Reverse-Gateway* entwickelt werden, welches die Anfragen des Zielsystems in die benötigte Form des Altsystems transformiert.
- *Composite-Database*: Die beiden zuvor genannten Methoden können kombiniert und gemeinsam genutzt werden, dieser Ansatz wird als *Composite-Database*-Strategie bezeichnet. Dabei werden die Datenbank-Systeme des Alt- und Neusystems parallel betrieben; erst nach abgeschlossener Migration kann das Legacy-System und dessen Datenbank vom Betrieb genommen werden.

Nachfolgende Abbildung 3-4 zeigt im linken Bereich einen möglichen Aufbau des *Database-First*- und im rechten Bereich den des *Database-Last*-Ansatzes mit den jeweils verwendeten Gateways.

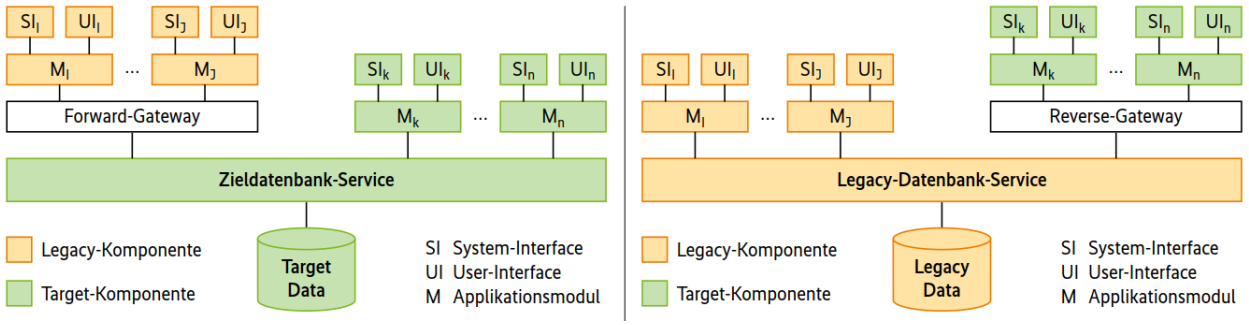

Abbildung 3-4: Database-First vs. Database-Last (Kaps, 2016)

Nach der Implementierung und Integration der Gateways werden in weiterer Folge die Funktionalitäten des Altsystems schrittweise in die Zielumgebung überführt und können anschließend im Altsystem deaktiviert werden. Wurden alle Module des Altsystems in die neue Umgebung transformiert, kann dieses vollständig deaktiviert und ausgemustert werden. (Kaps, 2016)

Bei der *Chicken-Little*-Strategie gilt es zu beachten, dass die Erstellung der benötigten Gateways sehr aufwendig sein kann und man abhängig von der gewählten Vorgehensweise umfangreiche Anpassungen vornehmen muss.

Beim *Database-First*-Ansatz werden klar definierte Schnittstellen zwischen dem Legacy-System und dessen Datenbank benötigt, damit das *Forward-Gateway* entsprechend platziert und Anfragen an die Zieldatenbank umleiten kann. Die Nichtverfügbarkeit dieser Schnittstellen kann dazu führen, dass eine Integration eines *Forward-Gateways* nur mittels tiefgreifenden Änderungen am Altsystem durchgeführt werden und zu hohen Aufwänden führen kann. Aus diesem Grund ist in der Regel der *Database-Last*-Ansatz zu bevorzugen, da meist entsprechende Werkzeuge verfügbar sind, welche die Transformation in das Datenformat des Altsystems automatisiert durchführen können. Hierbei gilt es zu beachten, dass die Transformationen der Anfragen lange dauern und sich somit auf die Performance des Zielsystems auswirken können. (Kaps, 2016)

Da bei der *Composite-Database*-Strategie sowohl das alte als auch neue Datenbank-System während der Migration im Einsatz bleiben, müssen Wege gefunden werden, um die Daten

synchron zu halten und deren Integrität und Konsistenz sicherzustellen. Weiters wird eine zusätzliche Komponente benötigt, welche die Koordination übernimmt und je nach Anfrage diese an die Gateways oder direkt zur jeweiligen Datenbank weiterleitet. (Bisbal et al., 1999)

Nachfolgende Abbildung 3-5 zeigt eine mögliche Architektur dieses Ansatzes.

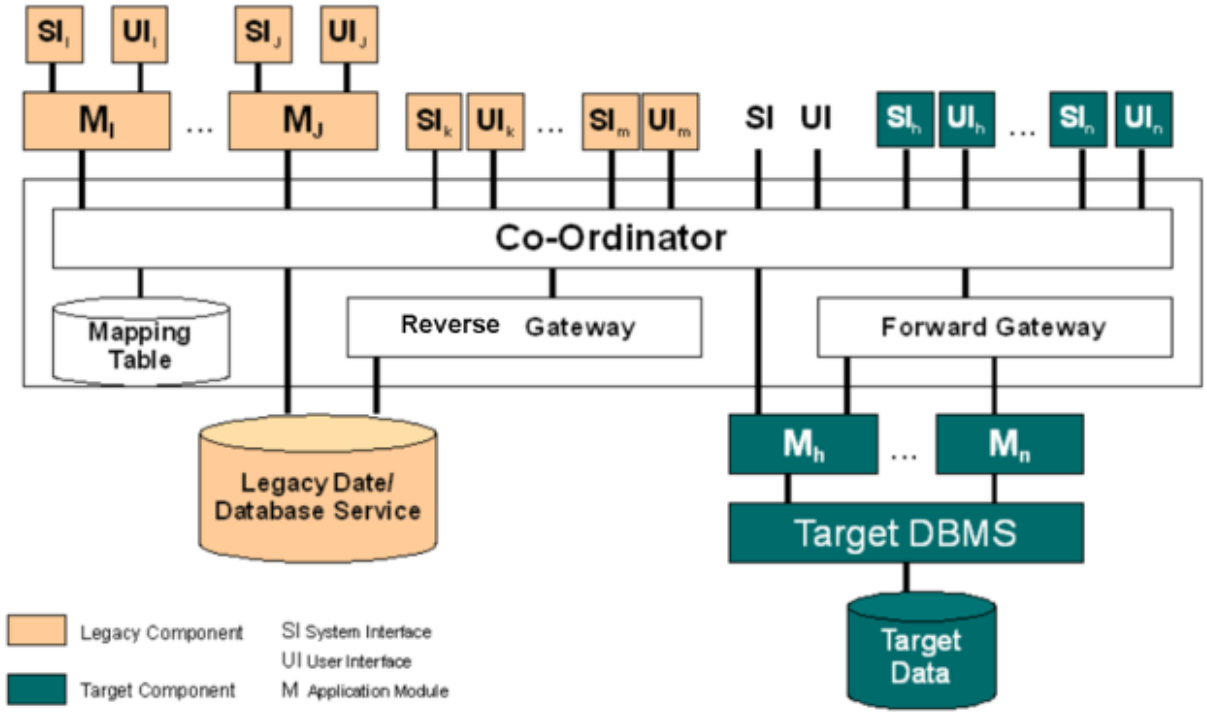

Abbildung 3-5: System-Aufbau für die Migration nach der Composite-Database-Strategie (Starke, 2017)

Der parallele Einsatz der Datenbank-Systeme kann dazu führen, dass bei der *Composite-Database*-Strategie verteilte Transaktionen durchgeführt werden müssen, welche – wie bereits in Kapitel 2.3.3 diskutiert – die Komplexität des Gesamtsystems erheblich steigern können. (Bisbal et al., 1999)

3.2.2 Butterfly-Methode

Eine Vorgehensweise für die reine Datenmigration zwischen Alt- und Zielsystem stellt die *Butterfly*-Methode dar. Dabei wird davon ausgegangen, dass während der Migration keine Zusammenarbeit zwischen den beiden Systemen notwendig ist und daher – im Gegensatz zu der eingangs diskutierten *Chicken-Little*-Strategie – kein Gateway benötigt wird. (Kaps, 2016) Bisbal et al. (1999) streichen dabei hervor, dass eine funktionierende Transformation der Datenbasis die Grundlage für eine erfolgreiche Migration eines Systems in eine neue Zielumgebung ist. Diese Methode zielt darauf ab, die Migration der Datenbasis in das Zielsystem schrittweise durchzuführen, um dabei den Betrieb des Legacy-Systems nicht zu beeinflussen und somit Produktionsausfälle zu vermeiden. (Kaps, 2016)

Bei diesem Ansatz wird zu Beginn die Datenbank des Altsystems für schreibende Zugriffe gesperrt und ein *Data Access Allocator* (DAA) eingesetzt, welcher zwischen Legacy-System und Legacy-Datenbank positioniert wird und Anfragen an die Datenbank steuert. Wird vom Altsystem

eine Datenmanipulation ausgelöst, speichert der DAA diese Änderung in einem temporären Speicher. Benötigt das Altsystem lesenden Zugriff auf Daten, ruft der DAA nicht-manipulierte Daten direkt aus der Legacy-Datenbank und manipulierte aus dem temporären Speicher ab. (Bisbal et al., 1999)

Während das Altsystem sich weiterhin im Betrieb befindet, werden vom sogenannten *Chrysaliser* die Daten aus der Legacy-Datenbank geladen, transformiert und in der Zieldatenbank gespeichert. Wurde die Legacy-Datenbank vollständig überführt, wird der bisher eingesetzte, temporäre Speicher (TS1) ebenfalls in den Lesemodus versetzt und ein neuer temporärer Speicher (TS2) erstellt. Der *Chrysaliser* wiederholt den Transformationsvorgang für TS1 und überführt alle Daten in das Zielsystem. Nach der Fertigstellung wird wieder ein neuer temporärer Speicher (TS3) erstellt und mit der Überführung von TS2 begonnen. Dieser Vorgang wird solange wiederholt, bis die Dauer einer Transformation so kurz ist, dass in dieser Zeit keine neuen Datenmanipulationen im zuletzt erstellten temporären Speicher stattgefunden haben. Sobald alle Daten überführt wurden, kann das Legacy-System deaktiviert und das Zielsystem in Betrieb genommen werden. (Bisbal et al., 1999)

Nachfolgende Abbildung 3-6 stellt diese Vorgehensweise grafisch dar.

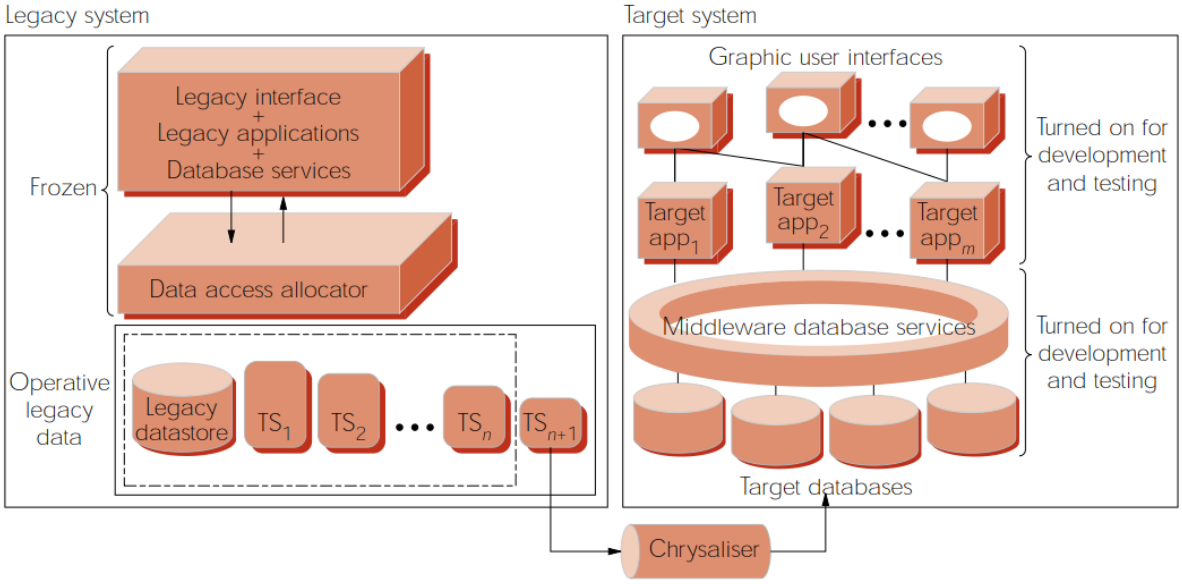

Abbildung 3-6: Butterfly-Methode (Bisbal et al., 1999)

Nach Kaps (2016) eignet sich diese Methode besonders für große Datenmengen, welche in der Regel zu lang andauernden Migrationsprozessen führen können. Neben der Vermeidung von Produktionsausfällen bietet diese Methode noch weitere Vorteile:

- Die Migration kann bei Problemen jederzeit abgebrochen und die in den temporären Datenspeichern enthaltenen Daten in die Datenbank des Altsystems zurückgespielt werden.
- Während der Migration kann das neu entwickelte Zielsystem mit den bereits migrierten Daten intensiv getestet werden.

Ähnlich wie bei den zuvor betrachteten Gateways der *Chicken-Little*-Strategie kann auch bei dieser Methode die Entwicklung des *Chrysalisers* sehr aufwendig werden, da dieser gewöhnlicherweise komplexe Transformationsregeln abbilden muss. Weiters sollte berücksichtigt werden, dass je nach Größe des Datenbestandes möglicherweise zusätzliche Hardware benötigt wird, um den speicherintensiven Migrationsprozess durchführen zu können. (Kaps, 2016)

3.2.3 Split Backend and Frontend

Im Gegensatz zu den bisher vorgestellten Methoden, welche den Fokus auf die Transformation der Daten in das Zielsystem legen, fokussieren sich die nachfolgenden Methoden auf die zu überführende Programmlogik eines Legacy-Systems.

Die Methode *Split Backend and Frontend* eignet sich für monolithische Systeme, welche auf einer Schichtenarchitektur basieren und bereits eine grundsätzliche Trennung zwischen Benutzerschnittstelle und Programmlogik aufweisen. Bei dieser Vorgehensweise wird die Benutzerschnittstelle in eine eigenständige Frontend-Applikation ausgegliedert und kommuniziert in weiterer Folge über eine Schnittstelle mit der Backend-Applikation, welche die Programmlogik beinhaltet. (Richardson & Smith, 2016)

Diese Methode eignet sich als initialer Schritt, der getätigt werden kann, um ein monolithisches System auf relativ einfachem Wege in mehrere Services aufzuteilen. Durch die Einführung einer geeigneten Schnittstelle in der Backend-Applikation, kann diese gesondert getestet werden und begünstigt in weiterer Folge die Überführung in eine Microservices-Architektur, da diese Schnittstelle von zukünftigen Microservices wiederverwendet werden kann. (Richardson & Smith, 2016)

3.2.4 Change by Split

Die Strategie *Change by Split* kann eingesetzt werden, um ein großes System in kleinere aufzuteilen und diese für unterschiedliche Nutzergruppen zu optimieren. Bei dieser Vorgehensweise wird die Code- und Datenbasis eines Systems kopiert, um anschließend Optimierungsarbeiten für bestimmte Nutzergruppen an den einzelnen Kopien durchzuführen. (Starke, 2016)

Zu Beginn werden bei dieser Methode Benutzergruppen identifiziert, welche unterschiedliche Anforderungen an das System haben und für jede dieser Benutzergruppen eine eigene Kopie des Systems erzeugt. Anschließend werden die Anforderungen der einzelnen Gruppen gegenübergestellt und dabei Funktionalitäten ermittelt, welche gemeinsam verwendet oder lediglich von einer Gruppe benötigt werden. Die gemeinsam genutzten Komponenten werden in eigene Module ausgelagert und diese in die jeweiligen Kopien des Systems integriert. Überflüssiger Code kann somit entfernt werden, wodurch die einzelnen Systeme kleiner werden und im Optimalfall nur mehr Code für diese Funktionalitäten beinhalten, welche nur von der jeweiligen Benutzergruppe benötigt wird. (Starke, 2017)

3.2.5 Branch by Abstraction

Wie in Kapitel 2.2.3 beschrieben, ist ein Vorteil von Microservices die Möglichkeit, Änderungen schnell in Produktion zu bringen und neue Funktionalitäten zeitnah bereitzustellen, wofür in der Regel eine *Continuous-Delivery-Pipeline* eingesetzt wird. Bei der Verwendung des *Continuous Delivery* Ansatzes sollte sich der Quellcode grundsätzlich nach jeder Änderung in einem Zustand befinden, in welchem die Applikation gebaut und bereitgestellt werden kann. Um dies auch bei größeren Erweiterungen, deren Entwicklung für gewöhnlich einen längeren Zeitraum dauern kann, sicherzustellen, kann das *Branch by Abstraction* Muster angewendet werden. (Fowler, 2014)

Bei dieser Vorgehensweise wird eine alte Komponente durch eine neue ersetzt. Damit während der Entwicklung der neuen Komponente die Anwendung lauffähig bleibt und die alte Komponente weiterhin verwendet werden kann, wird vor den Eingangspunkten der alten und neuen Komponente eine weitere Komponente eingebaut, der sogenannte *Abstraction Layer*. Dieser delegiert vorerst alle Anfragen an die alte Komponente. Sobald die neue Komponente fertig entwickelt und getestet wurde, wird der *Abstraction Layer* entsprechend angepasst, um Anfragen an die neue Komponente zu delegieren. (Fowler, 2014)

Um im Fehlerfall auf die alte Komponente zurückwechseln zu können, empfiehlt es sich eine weitere – als *Toggle* bezeichnete – Komponente einzubauen, welche ein Umschalten zwischen der alten und neuen Komponente zur Laufzeit ermöglicht (Smith, 2013).

Dieses Muster wird in Abbildung 3-7 dargestellt.

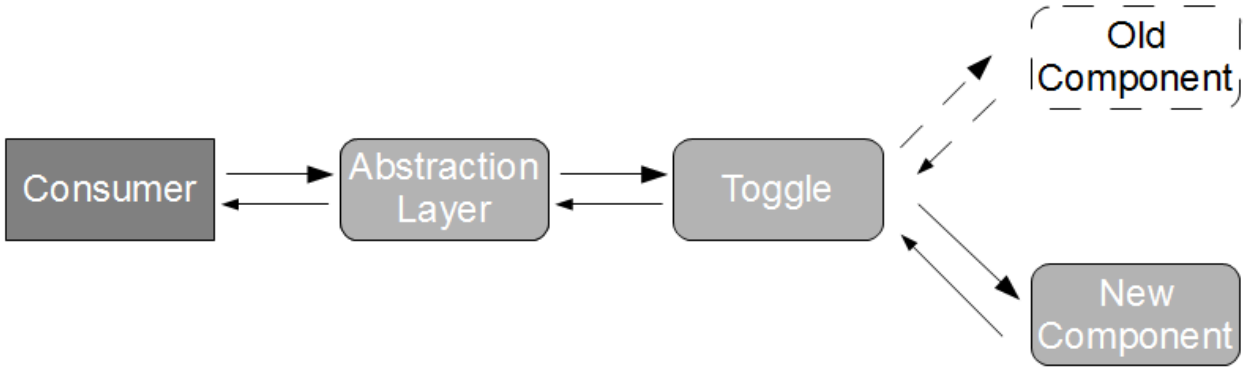

Abbildung 3-7: Branch by Abstraction (Smith, 2013)

Das Muster *Verify Branch By Abstraction* ist eine Erweiterung der soeben beschriebenen Methode. Dabei wird zwischen *Toggle* und den beiden Komponenten ein weiterer Baustein hinzugefügt. Diese sogenannte *Verify*-Komponente leitet eine Abfrage an beide Komponenten weiter und vergleicht anschließend die Ergebnisse miteinander. Weicht das Resultat der neuen Komponente vom Ergebnis der alten ab, werden nachträgliche Anfragen automatisch an die alte Komponente geleitet. Dieser Mechanismus ermöglicht eine automatisierte Prüfung der neuen Implementierung und eignet sich, um Schritt für Schritt eine Komponente eines Systems durch eine neue auszutauschen. (Smith, 2013)

3.2.6 Strangler Application

Bei der als *Strangler Application* oder *Legacy Strangulation* bezeichneten Methode werden neue Services in der Umgebung eines monolithischen Systems eingeführt, um dessen Funktionalitäten zu erweitern oder bestimmte Teile zu ersetzen. Dabei wird rund um den bestehenden Monolith ein neues System aus einzelnen Services erstellt, in welchem der Monolith im Laufe der Zeit weniger Aufgaben zu erledigen hat, selbst zu einem von vielen Services werden kann oder irgendwann nicht mehr benötigt wird und somit ausgemustert werden kann. (Fowler, 2004b) Nach Hammant (2013) eignet sich diese Vorgehensweise als relativ sichere Variante, um eine Ablöse eines Systems in mehreren Phasen durchzuführen und durch eine bessere geeignete Lösung zu ersetzen.

Den Kern dieser Methode bildet eine in den meisten Fällen neu einzuführende Komponente, welche dem System vorgeschaltet wird und die Steuerung der Anfragen übernimmt (Newman, 2015). Bei dieser Komponente kann es sich um einen Request Router (Richardson & Smith, 2016), Dispatcher (Santis, Florez, Nguyen, & Rosa, 2016) oder Proxy (Hughes, 2013) handeln. Da diese Komponente einem System vorgeschaltet werden muss, eignet sich diese Vorgehensweise besonders für Client-Server Architekturen (Richardson & Smith, 2016).

Nachfolgende Abbildung 3-8 zeigt dieses Prinzip eines vorgeschalteten Service beispielhaft anhand eines Legacy-Systems, welches um ein eigenständiges Service erweitert und dadurch ein Modul des Legacy-Systems obsolet gemacht wurde. Eingehende Anfragen werden von einem Proxy verarbeitet, ist eine Anfrage für das neue Service bestimmt, wird diese direkt an das Service weitergeleitet.

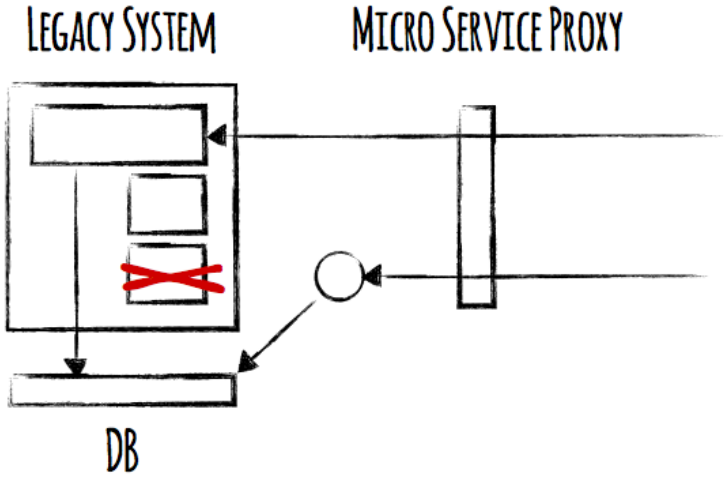

Abbildung 3-8: Microservice Proxy (Hughes, 2013)

Nach Wolff (2015) können für das Verteilen der Anfragen die *Enterprise Integration Patterns* herangezogen werden, welche verschiedene Muster beinhalten, um Nachrichten entsprechend zu verarbeiten. Wolff (2015) beschreibt diese wie folgt:

- *Message Router*: Ein *Message Router* verteilt Nachrichten an die einzelnen Services und kann somit gezielt Anfragen an ein Legacy-System zu einem anderen Service umleiten.

- *Content Based Router*: Der *Content Based Router* betrachtet den Inhalt einer Nachricht und entscheidet aufgrund dessen, wohin eine Nachricht weitergeleitet wird. Dies ermöglicht die gezielte Weiterleitung von Nachrichten an ein Service, wenn diese einen speziellen Inhalt aufweisen.
- *Message Filter*: Ein *Message Filter* kann verwendet werden, um eingehende Nachrichten zu filtern und nur relevante Nachrichten an ein Service weiterzuleiten.
- *Message Translator*: Um Nachrichten in das von einem neuen Service geforderte Format zu übersetzen, kann ein *Message Translator* verwendet werden. Dies ermöglicht es, Anfragen an ein Legacy-System so zu transformieren, dass diese von einem Service verarbeitet werden können und somit nicht mehr an das Legacy-System verarbeitet werden müssen.
- *Content Enricher* / *Content Filter*: Benötigt ein Service Daten, die in der Legacy-Nachricht nicht enthalten sind, kann ein *Content Enricher* die Nachricht um die benötigten Informationen ergänzen. Umgekehrt kann ein *Content Filter* eingesetzt werden, um unnötige Informationen auszufiltern.

Zusätzlich zum Message/Request Router werden bei der Methode *Strangler Application* Komponenten mit klar definierten Schnittstellen eingeführt, mittels derer eine Kommunikation zwischen dem Altsystem und den neu erstellten Services ermöglichen wird (Santis et al., 2016). Der Quellcode der für diese Komponenten erstellt werden muss, wird als „Glue code“ bezeichnet und soll lediglich dazu dienen, verschiedene Elemente miteinander zu verbinden (Clements et al., 2010).

In der nachfolgenden Abbildung 3-9 wird die Verwendung solcher Komponenten skizziert und die dadurch ermöglichte direkte Kommunikation zwischen den Systemen dargestellt.

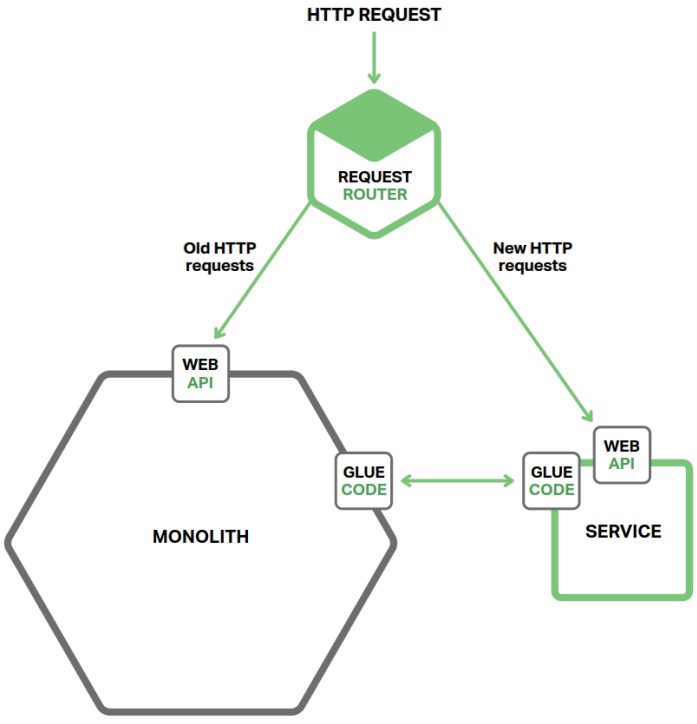

Abbildung 3-9: Architektur einer Strangler Application Vorgehensweise (Richardson & Smith, 2016)

Ein alternativer Begriff für diese Komponenten ist der A*nticorruption Layer*, welcher aus dem Domain-Driven Design stammt und dafür sorgt, die interne Repräsentation eines Systems nicht nach außen zu geben und dessen Domainmodell auf ein von extern verfügbares Modell übersetzt. Diese Komponenten werden vorwiegend eingesetzt, damit ein Service auf benötigte Daten des Monolithen zugreifen kann. (Richardson & Smith, 2016)

Zu den weiteren Möglichkeiten, auf Daten der monolithischen Applikation zuzugreifen, zählen Richardson und Smith (2016) den direkten Zugriff auf die Datenbank oder die Integration dieser in die eigene Datenbank eines Service. Beim direkten Zugriff muss beachtet werden, dass dadurch weiterhin eine Kopplung zwischen Service und Monolithen besteht und das Service auf Änderungen an der Datenbank reagieren muss (Wolff, 2015). Werden Daten des Monolithen in die Datenbank des Service integriert, muss sichergestellt werden, dass diese mit der monolithischen Datenbank regelmäßig synchronisiert wird (Richardson & Smith, 2016).

3.2.7 Extract Services

Ergänzend zu der im vorherigen Abschnitt vorgestellten *Strangler Application* Methode können neue, eigenständige Services durch Extrahieren bestehender Module aus dem Monolithen erstellt werden. Jedes losgelöste Modul verkleinert die monolithische Applikation und kann dazu führen, dass diese entweder abgelöst werden kann oder selbst zu einem Service wird. (Richardson & Smith, 2016)

Kann ein Modul nicht vollständig aus dem Legacy-System abgelöst werden, da es beispielsweise weiterhin benötigte interne Abhängigkeiten gibt, kann auch eine Duplizierung des Quellcodes Sinn machen, um dadurch die Aufteilung des Altsystems zu initiieren. Wird dieser Vorgang wiederholt durchgeführt und immer mehr Funktionalitäten aus dem Legacy-System ausgelagert, können sich dadurch die internen Abhängigkeiten auflösen und das interne Modul kann zu einem späteren Zeitpunkt entfernt werden. (Hughes, 2013)

Zu Beginn sollten solche Module aus dem Monolithen extrahiert werden, welche den größten Nutzen erzeugen (Richardson & Smith, 2016) oder am häufigsten geändert werden müssen (Balakrushnan et al., 2016). Auch jene Module sollten prioritär behandelt werden, die einen erhöhten Ressourcenbedarf aufweisen und gesondert skaliert werden sollten (Richardson & Smith, 2016).

Wie bei der *Strangler Application* Vorgehensweise sollte diese Methode schrittweise durchgeführt werden und auf mehrere Phasen aufgeteilt werden. Dabei gilt es zu beachten, dass nach jeder Phase umfangreiche Tests des gesamten Systems notwendig sind, um eine erfolgreiche Extraktion des Moduls sicherzustellen. (Hammant, 2013)

Wolff (2015) weist darauf hin, dass bei Microservices-Architekturen eine fachliche Trennung zwischen den Services angestrebt wird und dies bei dieser Vorgehensweise berücksichtigt werden sollte. Daher gilt es zu beachten, dass beim Loslösen von Funktionalitäten aus einem Monolithen, passende Schnittpunkte gefunden werden, um den Vorgaben der jeweiligen Microservices-Architektur zu entsprechen. (Wolff, 2015)

3.3 Schnittpunkte finden

Wie in den vorangegangenen Kapiteln erläutert, ist ein wesentlicher Punkt bei der Einführung von Microservices die Ermittlung von passenden Schnittpunkten, um große Systeme in kleinere Teile aufzuteilen. Dabei können die im vorherigen Abschnitt diskutierten Methoden eingesetzt werden, um eine Teilung einer monolithischen Software iterativ durchzuführen.

Damit die Aufteilung aber zu eigenständigen, voneinander möglichst unabhängigen Microservices führt, sollten dafür Werkzeuge gewählt werden, welche eine fachliche Trennung des Systems unterstützen und ermöglichen (Martincevic, 2016; Wolff, 2015). Grundsätzlich gilt es unabhängig vom gewählten Ansatz Wege zu finden, um Microservices zu erstellen, deren Implementierung, Anpassbarkeit und Auslieferbarkeit unabhängig von anderen Services sind und somit autonom, von eigenverantwortlichen Teams betrieben werden können (Martincevic, 2016).

Dieses Kapitel betrachtet zu Beginn die Trennung eines Systems nach fachlichen Aspekten. In weiterer Folge werden technische Aspekte diskutiert, welche ebenfalls zu berücksichtigen sind und Auswirkungen auf die zu erstellenden Microservices haben können.

3.3.1 Fachliche Trennung mittels Domain-Driven Design

Software sollte vorwiegend dem Zweck dienen, das Lösen von Problemen der Fachdomäne zu erleichtern (Martincevic, 2016). Daher ist das genaue Verständnis über die jeweilige fachliche Domäne eine Voraussetzung für die Erstellung einer guten Software (Starke, 2015).

Diese fachliche Ausrichtung sollte auch bei der Modellierung eines Systems stets im Vordergrund stehen und sich nicht auf technische Aspekte fokussieren. Dabei ist es wichtig, sowohl Fachexperten als auch Softwareentwickler in die Systemmodellierung einzubeziehen, um dadurch ein gemeinsames Verständnis für die Fachdomäne zu erhalten. (Starke, 2015) Nur durch ein ausführliches Wissen über die Funktionsweise der Domäne ist es in der Regel möglich, diese durch eine Software entsprechend unterstützen zu können (Martincevic, 2016).

Ein vorhandenes Legacy-System kann für das Verständnis der Fachdomäne von Vorteil sein, da in diesem die Funktionalitäten in der Regel fachlich klar abgegrenzt sind und somit als Ausgangspunkt für die Modellierung einer Microservices-Architektur und die Aufteilung in fachliche Services dienen kann. Dabei gilt es jedoch zu beachten, dass innerhalb des Legacy-Systems vorhandene Module meist nach technischen Aspekten getrennt sind und kein fachlicher Schnitt vorherrscht – eine direkte Überführung von Modulen der Altanwendung in Microservices ist daher in der Regel nicht zu empfehlen und stattdessen eine Änderung des fachlichen Schnitts zu bevorzugen. (Wolff, 2015) Die Aufteilung sollte dabei so erfolgen, dass Services jeweils eine fachliche Geschlossenheit aufweisen (Röwekamp & Limburg, 2016).

Ein Ansatz, um eine fachliche Modellierung durchzuführen – welcher eine weite Verbreitung im Umfeld von Microservices aufweist – wird als Domain-Driven Design (DDD) bezeichnet und wurde von Evans (2004) geprägt. Dabei handelt es sich um eine Sammlung von Mustern zur Modellierung eines Systems (Vernon, 2016) unter Verwendung der Sprache der fachlichen Anwendungsdomäne (Starke, 2015).

Um das notwendige Verständnis der Fachdomäne zu erhalten und darauf aufbauend eine entsprechende Software zu entwickeln, setzt DDD auf zwei wesentliche Elemente: dem *Bounded Context* und der *Ubiquitous Language* (Vernon, 2016).

Die *Ubiquitous Language* stellt eine gemeinsame und unmissverständliche Sprache dar, welche auf der Fachdomäne basiert. Diese Sprache sollte von allen Beteiligten verwendet werden und sich auch bei der Entwicklung der Software im Quellcode wiederfinden. Dringt man weiter in die Fachdomäne ein und sammelt dabei neue Erkenntnisse, soll sich dies auch in der *Ubiquitous Language* widerspiegeln und diese entsprechend erweitert oder geändert werden. (Evans, 2004)

Der *Bounded Context* stellt eine semantische Kontextgrenze dar, innerhalb welcher jede Komponente eines Models eine bestimmte Bedeutung und kontextspezifische Funktionalitäten auszuführen hat. Jeder *Bounded Context* definiert seine eigene *Ubiquitous Language* und beschreibt damit die von ihm abgebildete Fachdomäne. Ein komplexes System, welches in der Regel aus einer Vielzahl von *Bounded Contexts* besteht, kann somit mehrere *Ubiquitous Languages* aufweisen, wodurch ein Begriff innerhalb des Systems verschiedene Bedeutungen einnehmen kann. Ein *Bounded Context* sollte dabei in der Zuständigkeit von einem einzelnen Team sein, welches für die Entwicklung verantwortlich ist. (Vernon, 2016) Jeder *Bounded Context* sollte eine explizite Schnittstelle aufweisen, welche das Modell für Zugriffe von außen zur Verfügung stellt und somit die Kommunikation mit anderen *Bounded Contexts* ermöglicht (Newman, 2015).

Dass bei der Verwendung von DDD und der Modellierung der Domänenmodelle mit *Bounded Contexts* ein Begriff verschiedene Bedeutungen haben kann, beschreibt Wolff (2015) wie folgt:

> *„Mit dem Einsatz von Bounded Contexts akzeptieren Sie, dass Dinge, die ähnlich aussehen und den gleichen Namen tragen, in verschiedenen Kontexten unterschiedliche Bedeutung haben. Das ist kein »Problem« der Implementierung, sondern eine harte Realität des Lebens.“* (Wolff, 2015, S. 50)

Das Zusammenspiel der *Bounded Contexts* wird beim DDD als strategisches Design bezeichnet. Mit Hilfe einer *Context Map*, welche die *Bounded Contexts* und deren Verbindungen zueinander beinhaltet, kann der Aufbau eines Systems dargestellt werden. (Wolff, 2015)

Um die Modellierung der Fachdomäne innerhalb eines *Bounded Contexts* zu ermöglichen, beinhaltet DDD verschiedene Bausteine, deren Beziehungen zueinander in Abbildung 3-10 abgebildet werden.

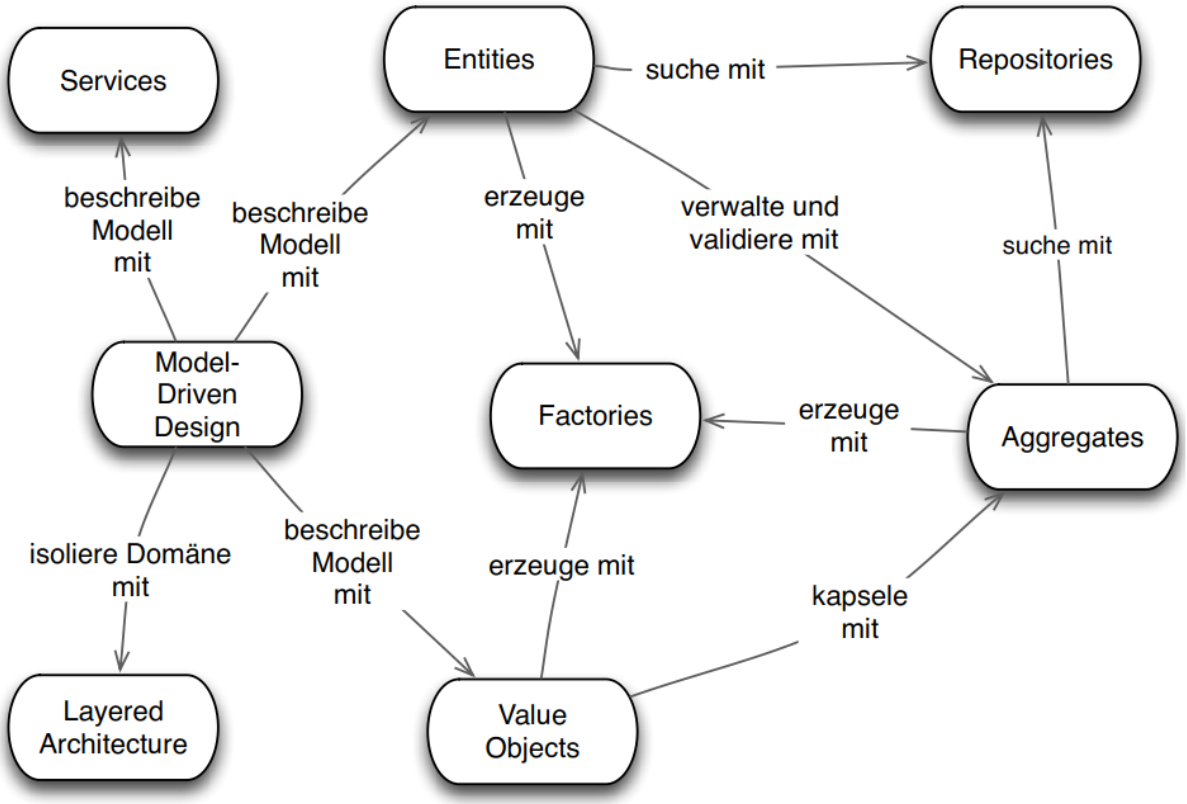

Abbildung 3-10: Muster des Domain-Driven Design (Starke, 2015)

Als Basisbausteine gelten *Entities*, *Value Objects* und *Services*. *Entities* stellen die Kernobjekte der Fachdomäne dar, verfügen über einen eindeutigen, unveränderlichen Schlüssel und haben einen klar definierten Lebenszyklus. Im Gegensatz dazu haben *Value Objects* keine eindeutige Identität und werden eingesetzt, um den Zustand anderer Objekte zu beschreiben. Fachliche Abläufe werden mit *Services* modelliert, welche zustandslose Operationen ausführen und als Parameter *Entities* und *Value Objects* entgegennehmen. (Starke, 2015) Um *Entities* und *Value Objects* zu einem Domänenobjekt zusammenzusetzen, können *Aggregates* verwendet werden (Wolff, 2015).

In diesem Abschnitt wurden die grundlegenden Elemente des DDD vorgestellt, für eine weiterführende Beschreibung dieser Herangehensweise wird an dieser Stelle auf entsprechende Literatur verwiesen: Evans (2004) und Vernon (2016) beschäftigen sich ausführlich damit.

3.3.2 DDD im Kontext von Microservices und Legacy-Systemen

Der *Bounded Context* Ansatz eignet sich für die Erstellung von Microservices, um die geforderte fachliche Geschlossenheit der Services zu erreichen (Röwekamp & Limburg, 2016). Über die expliziten Schnittstellen kann die Kommunikation zwischen Microservices stattfinden und durch klar definierte Grenzen eine lose Kopplung zwischen den Services erreicht werden (Newman, 2015).

In der Regel sollte ein *Bounded Context* durch einen Microservice abgedeckt werden. Ist dieser jedoch zu umfangreich für einen Service oder sollten beispielsweise einzelne Bereiche innerhalb

eines *Bounded Context* gesondert skalierbar sein, kann auch die Aufteilung auf mehrere Microservices sinnvoll sein. Andererseits sollte ein Microservice nicht mehrere *Bounded Contexts* umfassen, da dadurch die unabhängige Entwicklung eines Microservices beeinträchtigt werden kann. (Wolff, 2015)

Hinsichtlich der Kommunikation zwischen Services finden sich weitere Elemente im DDD, welche in einer Microservices-Architektur von Vorteil sein können (Santis et al., 2016).

Das bereits in Abschnitt 3.2.4 erwähnte Muster des *Anticorruption Layers* (ACL) kann verwendet werden, um eine Transformation des internen Modells durchzuführen und dabei nur gewisse Informationen in einem für externe Services aufbereitetem Modell zur Verfügung zu stellen (Röwekamp & Limburg, 2016). *Anticorrpution Layer* eignen sich besonders, um eine Integration eines Legacy-Systems durchzuführen und dadurch das eigene Domänenmodell abzusichern (Vernon, 2016).

Der *Shared Kernel* Ansatz ermöglicht die Definition von gemeinsamen Domänenobjekten damit diese in mehreren Microservices genutzt werden können (Röwekamp & Limburg, 2016), wobei allerdings zu beachten ist, dass dies zu einer engen Kopplung zwischen den Services führen kann (Newman, 2015). Beim *Conformist* Muster nutzt der aufrufende Service das zur Verfügung gestellte Domänenmodell, hingegen wird beim *Customer/Supplier* Ansatz dem Aufrufer ein seinen Anforderungen entsprechendes Modell zur Verfügung gestellt (Wolff, 2015).

Wird im Zuge einer Microservices-Architektur eine asynchrone, durch Ereignisse gesteuerte Kommunikation angestrebt, kann dies mit Hilfe von *Domain Events* modelliert werden (Vernon, 2016).

Um DDD im Umfeld von Legacy-Systeme einzuführen und dadurch eine Ablöse oder Erweiterung dieser zu initiieren, empfehlen sich nach Evans (2013) verschiedene Strategien, welche nachfolgend auszugsweise erläutert werden.

Bubble Context

Die Strategie *Bubble Context* kann verwendet werden, um neue fachliche Anforderungen an ein Legacy-System mit Hilfe des DDD-Ansatzes zu implementieren und eine klare Trennung zwischen der neuen Funktionalität und dem Altsystem herbeizuführen. Die neu abzubildende Funktionalität wird in einem *Bounded Context* definiert, der Zugriff auf benötigte Daten des Legacy-Systems erfolgt mittels eines ACL. (Evans, 2013)

Der ACL dient dabei als Repository für den neuen Kontext und übersetzt für diesen die Daten des Altsystems in das benötigte Format. Die Daten werden dabei nicht zwischengespeichert, sondern werden direkt über das Legacy-System geladen. Eine Koordinator-Komponente steuert diesen Ablauf und lädt die benötigten Daten, leitet diese für die Transformation an verschiedene Translatoren weiter und gibt ein passendes Objekt an das aufrufende Service des *Bubble Contexts* zurück. Je nach Anwendungsfall können über einen ACL auch mehrere Legacy-Systeme angebunden werden. (Evans, 2013)

Ein beispielhafter Systemaufbau eines *Bubble Context* Ansatzes wird in Abbildung 3-11 skizziert.

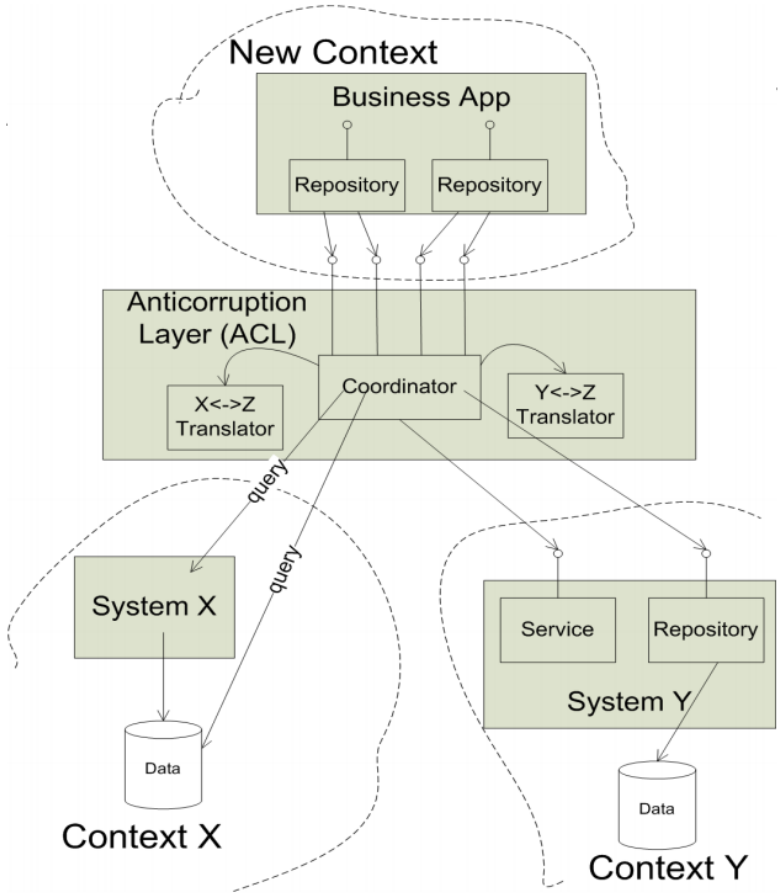

Abbildung 3-11: Bubble Context mit Anticorruption Layer (Evans, 2013)

Dieser Ansatz eignet sich, wenn nur eine geringe Menge an Daten vom Legacy-System benötigt wird, man auf eine Synchronisierung von Datenbanken verzichten möchte und die auszuführenden Operationen nicht performancekritisch sind. Es gilt jedoch zu beachten, dass diese Vorgehensweise zu einer starken Kopplung zwischen den Systemen führen kann, da für jede Abfrage an den ACL ein direkter Zugriff auf das Legacy-System erfolgen muss. Da der *Bubble Context* keine eigene Datenbank beinhaltet, müssen auch etwaige, für den neuen Kontext benötigte Erweiterungen an der Datenbasis im Legacy-System durchgeführt werden. Daher sollte in der Regel eine der nachfolgend beschriebenen Strategien bevorzugt verwendet werden, um mittels DDD eine neue fachliche Anforderung umzusetzen. (Evans, 2013)

Autonomous Bubble

Im Gegensatz zu dem soeben beschriebenen Ansatz *Bubble Context* werden bei der Strategie *Autonomous Bubble* Daten nicht direkt aus dem Legacy-System geladen, sondern vom ACL in eine Datenbank innerhalb des neu modellierten *Bounded Contexts* übertragen. Der ACL lädt die benötigten Daten in einem bestimmten Zyklus aus der Legacy-Datenbank, transformiert diese mit Hilfe der Translatoren in das vom *Bounded Context* erwartete Format und speichert die Datensätze in dessen Datenbank. Diese Methode führt zu einer losen Kopplung zwischen den Systemen, da das Legacy-System nur für die Synchronisierung verfügbar sein muss. Da die

Aktivitäten des ACL im Hintergrund ausgeführt werden und lediglich eine Aktualisierung der Datenbank erfolgt, ist der neue Kontext von Ausfällen des Legacy-Systems nicht direkt betroffen. (Evans, 2013)

Nach Evans (2013) kann dieser Ansatz in weiterer Folge mit den Mustern von ereignisgesteuerten Architekturen erweitert werden. Dabei reagiert der ACL auf auftretende Ereignisse im Legacy-System, überführt diese in *Domain Events* und speichert sie in der Nachrichten-Queue des neuen Kontexts.

Nachfolgende Abbildung 3-12 zeigt diesen Ansatz einer *Autonomous Bubble* unter Verwendung von Nachrichten-Queues.

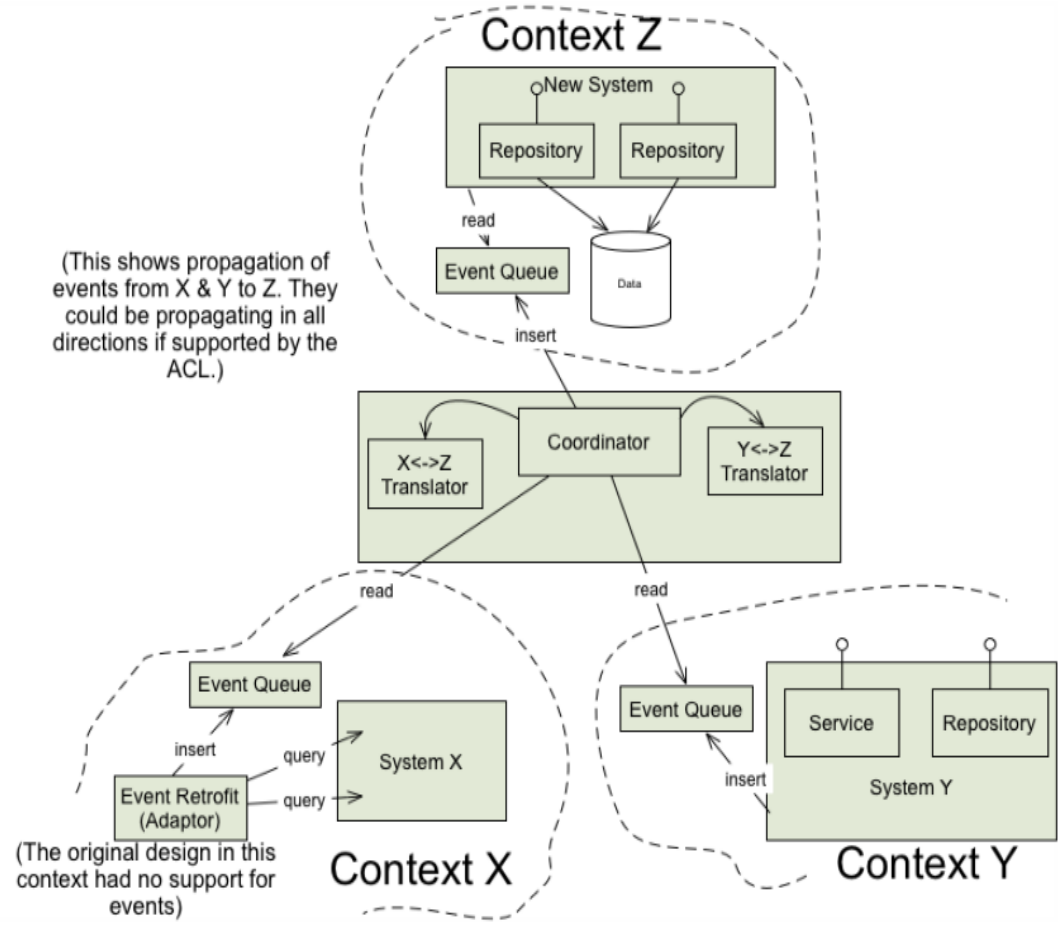

Abbildung 3-12: Autonomous Bubble mit Domain Events (Evans, 2013)

3.3.3 Dezentrales Datenmanagement

Wie bereits in Kapitel 2 über Microservices-Architekturen diskutiert, wirkt sich die Einführung von Microservices gewöhnlicherweise auch auf die Speicherung der Daten aus. Jedes Microservice sollte möglichst autonom und losgelöst von anderen Services agieren können, definierte Grenzen nicht überschreiten und lediglich den vorgesehen fachlichen Kontext betrachten. Die dafür benötigten Daten sollten ebenfalls Teil des jeweiligen Microservice sein und nicht in einer

gemeinsamen Datenbank verwaltet werden. Diese Punkte gilt es beim Zerteilen einer monolithischen Applikation zu berücksichtigen und werden in diesem Abschnitt diskutiert.

Nach Fowler und Lewis (2015) sollten Microservices dem Ansatz der transaktionslosen Kommunikation folgen und auf verteilte Transaktionen möglichst verzichten, da ein funktionierendes Transaktionsmanagement zu den schwierigsten Aufgaben bei verteilten System zählt (Richards, 2016).

ACID und verteilte Transaktionen

Eine Transaktion fasst mehrere, gemeinsam auszuführende Änderungen zusammen (Richards, 2016) und sollte den ACID-Eigenschaften folgen, welche nach Vogel et al. (2009) wie folgt definiert sind:

- ***A**tomicity*: Eine Transaktion muss als Ganzes abgearbeitet werden und kann als unteilbare Einheit betrachtet werden. Tritt ein Fehler vor Abschluss der Transaktion auf, müssen alle im Zuge der Transaktion bereits durchgeführten Aktionen wieder rückgängig gemacht werden.
- ***C**onsistency*: Das System muss sich nach Durchführung der Transaktion in einem konsistenten Zustand befinden.
- ***I**solation*: Transaktionen dürfen sich nicht gegenseitig beeinflussen.
- ***D**urability*: Alle Änderungen, die im Zuge der Transaktionen durchgeführt wurden, müssen nach Fertigstellung dieser dauerhaft gespeichert sein und dürfen auch durch einen Systemausfall nicht verloren gehen.

Verteilte Transaktionen, welche sich auf mindestens zwei getrennte Systeme auswirken, können mit Hilfe des *Two-Phase Commit Protocol* durchgeführt werden (Vogel et al., 2009) und sollen dafür Sorge tragen, dass sich alle beteiligten Systeme nach Abschluss einer Transaktion in einem konsistenten Zustand befinden (Newman, 2015). Dabei werden pro System eigenständige Transaktionen ausgeführt, welche mittels eines übergeordneten Transaktionsmanagers gesteuert werden. Jedes System prüft im ersten Schritt, ob die Transaktion durchgeführt werden kann und übermittelt das Ergebnis an den Transaktionsmanager. Erhält dieser von allen Systemen eine positive Rückmeldung, werden die Transaktionen bestätigt und durchgeführt. Eine einzelne negative Rückmeldung eines Systems ist ausreichend, damit der Transaktionsmanager den Vorgang abbricht und keine der untergeordneten Transaktionen bestätigt. (Newman, 2015)

Diese Vorgehensweise ist in der Regel fehleranfällig, da sowohl ein Ausfall des Transaktionsmanagers als auch eines einzelnen Systems dazu führen kann, dass die Systeme auf die Bestätigung warten müssen und sich gegenseitig blockieren können. Weiters muss jedes an einer verteilten Transaktion beteiligte System dafür Sorge tragen, dass nach Erhalt der Transaktionsbestätigung diese auch wirklich durchgeführt wird und eine entsprechende Fehlerbehandlung aufweist. (Newman, 2015)

CAP-Theorem und BASE-Prinzip

Als Alternative zu verteilten Transaktionen finden sich im Umfeld von verteilten Microservices-Architekturen andere Ansätze, um die soeben erläuterten Nachteile zu behandeln. Dabei werden

neben der Konsistenz eines Systems dessen Verfügbarkeit und Ausfallstoleranz betrachtet, um dadurch Anforderungen, die an verteilte System gestellt werden – wie beispielsweise Replikation von Daten, Skalierbarkeit, niedrige Reaktionszeiten – erfüllen zu können (Edlich, Friedland, Hampe, Brauer, & Brückner, 2011). Diese drei Faktoren werden im Zuge des CAP-Theorems wie folgt beschrieben:

- ***C**onsistency*: Ein verteiltes System erreicht nach Abschluss einer Transaktion einen konsistenten Zustand, indem die durchgeführten Änderungen auf alle Knoten repliziert wurden und nachfolgende Leseoperationen, unabhängig vom jeweiligen Knoten, die geänderten Werte zurückliefern (Edlich et al., 2011).
- ***A**vailability*: Ein System wird als verfügbar eingestuft, wenn es auf jede Anfrage eine Antwort an den Aufrufer zurückliefert (Newman, 2015) und diese Antwort in einer akzeptablen Reaktionszeit erfolgt (Edlich et al., 2011). Die Reaktionszeit kann sich dabei von System zu System unterscheiden und sollte zumindest so gewählt werden, dass sie keinen direkten Einfluss auf die Geschäftsentwicklung hat (Edlich et al., 2011).
- ***P**artition Tolerance*: Die Partitionstoleranz beschreibt das Verhalten eines Systems, wenn ein oder mehrere Knoten ausfallen. Dabei darf der Ausfall eines Knotens nicht zu einem Gesamtausfall des Systems führen, sodass Anfragen weiterhin von anderen, verfügbaren Knoten beantwortet werden. (Starke, 2015)

Das CAP-Theorem nach Brewer (2000) sagt dabei aus, dass nur jeweils zwei dieser drei Faktoren zu einem Zeitpunkt erreicht werden können und dies beim Aufbau eines verteilten Systems berücksichtigt werden sollte (Starke, 2015).

Wird ein hoch verfügbares und ausfalltolerantes System benötigt, müssen dafür Abstriche bei der Konsistenz gemacht werden, da beim Ausfall von Knoten Änderungen nicht auf diese repliziert werden können und diese dann vorübergehend einen inkonsistenten Zustand aufweisen. Um Konsistenz und Partitionstoleranz zu erreichen, sind hingegen Einschränkungen bei der Verfügbarkeit hinzunehmen, da erst nach einer vollständigen Replikation auf alle Knoten das System wieder Anfragen korrekt beantworten kann. Als dritte Alternative können bei der Partitionstoleranz Abstriche akzeptiert werden, um dadurch ein verfügbares und konsistentes System zu gewährleisten. Dabei muss jedoch berücksichtigt werden, dass sich die zuletzt genannte Variante lediglich für nicht-verteilte Systeme eignet, da bei Netzwerkausfällen in einem verteilten System nur einer der beiden Zustände erreicht werden kann. (Newman, 2015)

Als Gegensatz zu ACID hat sich für verteilte Systeme das BASE-Prinzip („basically available, soft state, eventually consistent“) als alternatives Konsistenzmodell etabliert, welches die Verfügbarkeit eines Systems in den Vordergrund stellt und Konsistenz dieser unterordnet (Starke, 2015). Edlich et al. (2011) beschreiben dies wie folgt:

> *„Wo ACID einen pessimistischen Ansatz bei der Konsistenz verfolgt, ist BASE ein optimistischer Ansatz, bei dem Konsistenz als ein Übergangsprozess zu sehen ist und kein fester Zustand nach einer Transaktion. Daraus entsteht ein*

> *völlig neuartiges Verständnis von Konsistenz: Eventually Consistency.“* (Edlich et al., 2011, S. 33–34)

Eventual Consistency bedeutet, dass ein System nicht direkt nach Abschluss einer Transaktion, sondern erst nach einer Phase der Inkonsistenz einen konsistenten Zustand erreicht. Die Dauer dieser Phase wird dabei unter anderem von der Anzahl der Knoten und der Last des Systems beeinflusst. (Edlich et al., 2011) Während dieser Übergangsphase hin zu einem konsistenten Zustand befindet sich das System in einem „soft transition state“ (Richards, 2016).

Der BASE-Ansatz findet in auf Microservices basierten Systemen eine weite Verbreitung, um dadurch eine möglichst hohe Verfügbarkeit der einzelnen Services zu erreichen (Röwekamp & Limburg, 2016). Ausschlaggebend für die Wahl eines geeigneten Ansatzes ist die abzubildende Fachlichkeit und ob diese nach dem Prinzip der *Eventual Consistency* ausgeführt werden beziehungsweise ob ein Geschäftsprozess so verändert werden kann, dass dieser Ansatz in Frage kommt (Flohre, 2015; Röwekamp & Limburg, 2016).

Transaction boundaries identifizieren

Bei der Modellierung des Systems und dem Auffinden von geeigneten Schnittpunkten, sollten die soeben diskutierten Ansätze berücksichtigt und darauf Rücksicht genommen werden, dass über die Grenzen der einzelnen Microservices hinweg die Konsistenz der Daten kaum zugesichert werden kann (Wolff, 2015). Newman (2015) weist in diesem Zusammenhang darauf hin, dass hierbei aber keine allgemeine Entscheidung für einen der Ansätze getätigt werden muss, sondern diese auch in Kombination verwendet werden können. Dabei kann beispielsweise innerhalb eines Microservice die Konsistenz nach dem ACID-Paradigma sichergestellt und für das Gesamtsystem der Ansatz der *Eventual Consistency* gewählt werden (Newman, 2015).

Grundsätzlich sollte versucht werden, die Grenzen von Microservices so zu gestalten, dass verteilte Transaktionen nicht benötigt werden. Dafür sollte bei der Modellierung klar definiert werden, welche Daten innerhalb einer Transaktion gespeichert werden müssen (Wolff, 2015). Dabei sollte stets die abzubildende Fachdomäne berücksichtigt und ermittelt werden, welche Domänenobjekte tatsächlich zusammenhängend gespeichert werden müssen, um Geschäftsregeln nicht zu verletzen. Daraus ergeben sich Grenzen für einzelne Transaktionen innerhalb eines Kontexts, welche als *Transaction boundaries* bezeichnet werden können. Diese Grenzen sollten möglichst klein gestaltet werden, um eine Skalierbarkeit von Services zu fördern. (Posta, 2016)

Um diese Grenzen zu modellieren und Schnittpunkte auf Ebene der Datenhaltung zu finden, eignet sich ebenfalls die in den vorangegangenen Abschnitten diskutierte Vorgehensweise des DDD. *Aggregates* werden für den Zusammenschluss von Domänenobjekten eingesetzt (Flohre, 2015), sollten möglichst klein gehalten und immer innerhalb einer einzelnen Transaktion verarbeitet werden (Vernon, 2016). Ein *Aggregate* sollte immer innerhalb eines Microservice verarbeitet werden und dessen Grenzen – den *Bounded Context* – nicht überschreiten, um verteilte Transaktionen zu vermeiden oder keine aufwendige Kompensationslogik ausführen zu müssen (Flohre, 2015).

Kommunikation mittels Events

Auch wenn ein sauberer fachlicher Schnitt erfolgt, alle benötigten Domänenobjekte innerhalb eines *Bounded Contexts* gespeichert werden und Services somit unabhängig voneinander agieren können, werden in der Regel Mechanismen benötigt, um anderen Services Daten zur Verfügung zu stellen beziehungsweise diese über Änderungen zu informieren.

Posta (2016) empfiehlt dafür den Einsatz von Events, um über die Grenzen von *Transaction boundaries* oder *Bounded Contexts* zu kommunizieren. Dabei werden nach erfolgreichem Abschluss einer Transaktion Events in einer Nachrichtenwarteschlange erstellt, welche von anderen Services bei Bedarf abonniert werden kann und diese somit über Änderungen informiert werden (Flohre, 2015). Diese Events sind unveränderbar und enthalten in der Regel die veränderten Daten, wodurch jedes Service beim Erhalt eines Events selbst entscheiden kann, welche Daten für den jeweiligen Kontext relevant sind und diese entsprechend verarbeiten (Posta, 2016).

Ein Problem bei dieser Vorgehensweise ist, dass für die Speicherung der Änderungen in der Datenbank und dem anschließenden Speichern des Events in der Nachrichtenwarteschlange, zwei Transaktionen benötigt werden, da hier in der Regel unterschiedliche Systeme zum Einsatz kommen (Posta, 2016). Um diesem Fall entgegenzutreten, eignen sich Mechanismen wie beispielsweise *Event Sourcing*, bei welchen Änderungen an Objekten nicht direkt in einer Datenbank durchgeführt, sondern die Ereignisse selbst in einem *Event Store* gespeichert werden und aus diesen die Änderungen an Domänenobjekten rekonstruiert werden können (Wolff, 2015).

Die nachfolgende Abbildung 3-13 zeigt eine beispielhafte *Event Sourcing* Architektur.

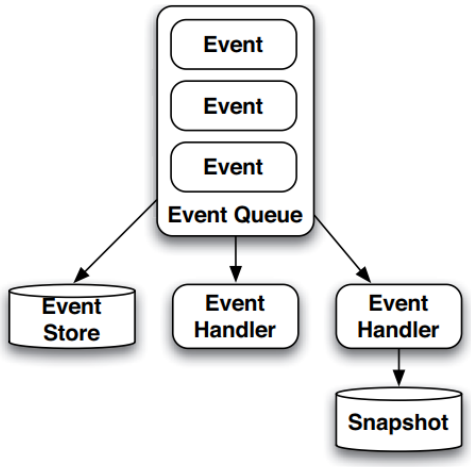

Abbildung 3-13: Event Sourcing (Wolff, 2015)

Ein auftretendes Ereignis wird in der Nachrichtenwarteschlange (*Event Queue*) abgelegt und kann somit von beliebigen *Event Handler* verarbeitet werden. Alle Events werden im *Event Store* abgelegt und dürfen im Nachhinein nicht mehr veränderbar sein. Der *Event Store* ermöglicht die Rekonstruktion aller aufgetretenen Änderungen, woraus sich der Zustand eines Objektes zu einem gewünschten Zeitpunkt ableiten lässt. Um den aktuellen Zustand von Objekten einfach abfragen zu können, empfiehlt sich der Einsatz einer „Snapshot“-Datenbank. Dabei wird ein *Event Handler* eingesetzt, welcher die Nachrichtenwarteschlange abonniert und beim Erhalt

neuer Events die damit verbundenen Änderungen an Objekten in der „Snapshot“-Datenbank abbildet. (Wolff, 2015)

Dieser Ansatz hat den Vorteil, dass lediglich das Speichern des Ereignisses in der Nachrichtenwarteschlange in einer Transaktion durchgeführt werden muss und kein Zwei-Phasen Commit oder ähnliche Ansätze benötigt werden (Posta, 2016).

Datenbankstrukturen aufteilen

Nachdem Schnittpunkte gefunden und die technischen Ansätze für ein verteiltes Datenmanagement gewählt wurden, müssen in der Regel bestehende Datenbankstrukturen der monolithischen Systeme aufgeteilt und Daten in die neue Struktur migriert werden.

Nach Santis et al. (2016) haben sich drei Muster etabliert, welche im Zuge der Aufteilung einer monolithischen Anwendung in Microservices angewendet werden sollten, um neben den Änderungen am Quellcode die notwendigen Änderungen an der Datenbank durchzuführen:

- *Änderungen in kleinen Schritten durchführen*:

  Um Fehlerquellen zu minimieren beziehungsweise im Fehlerfall schnell reagieren zu können, sollten umfangreiche Änderungen an der Datenbank stets aufgeteilt und in kleinen Paketen durchgeführt werden. Die Änderungen an der Datenbank sollten dabei mit Änderungen am Quellcode abgestimmt und zusammen mit diesen bereitgestellt werden. (Santis et al., 2016)

- *Änderungsskripte versionieren*:

  Alle für die Umstrukturierung benötigten Datenbankskripte sollten zusammen mit dem Quellcode der Applikation in einem gemeinsamen Versionsverwaltungssystem gespeichert werden. Dabei soll eine einheitliche Struktur festgelegt werden, wie die entsprechenden Dateien benannt und abgespeichert werden müssen. (Santis et al., 2016)

- *Änderungen automatisiert durchführen*:

  Wenn die durchzuführenden Datenbankänderungen mit dem Quellcode abgestimmt wurden und gemeinsam bereitgestellt werden können, sollte dieser Vorgang automatisiert ausgeführt werden. Dabei sollte bei der Automatisierung auf die *Continuous-Delivery-Pipeline* der Microservices zurückgegriffen werden und diese auch für Datenbankänderungen genutzt werden. (Santis et al., 2016)

Wie bestehende Datenbankstrukturen geteilt und Beziehungen zwischen Tabellen gelöst werden können, wird im Zuge dieser Arbeit nicht weiter im Detail diskutiert und an dieser Stelle auf entsprechende Literatur verwiesen: Ambler und Sadalage (2006) beschreiben dafür verschiedene strategische Ansätze, die sich für diesen Zweck eignen.

3.3.4 Benutzerschnittstelle

Um aus einem System, dass aus einer Vielzahl von Microservices besteht, ein für den Nutzer zugängliches Gesamtsystem zu erstellen, erfolgt in der Regel die Integration der einzelnen Services über eine gemeinsame Benutzeroberfläche (Wolff, 2015). Dabei stößt man beim Thema

Benutzerschnittstelle – in weiterer Folge auch als User Interface (UI) bezeichnet – auf eine gewisse Uneinigkeit in der Literatur, ob dieses Teil eines Microservices sein oder als eigenständige Anwendung entwickelt werden sollte.

Nach Wolff (2015) sollte das UI als Teil des Microservices bereitgestellt werden, da diese die Fachlichkeit, die ein Microservice implementiert, dem Benutzer zugänglich macht. Da lediglich ein einzelnes Team für ein Microservice zuständig sein sollte, kann dieses alle benötigten Änderungen an den Funktionalitäten – unabhängig ob diese das UI, die Logik oder die Datenbank betreffen – durchführen. Grundsätzlich muss das UI aber nicht in einem Microservice integriert werden, wenn beispielsweise die Logik eines Microservice von verschiedenen Benutzerschnittstellen benötigt wird. Einen alternativen Ansatz zu Microservices mit integriertem UI stellen Self-Contained Systems (SCS) dar, bei welchen die Benutzerschnittstelle als eigenständiger Service entwickelt wird und dieser auf verschiedene Microservices zugreift. (Wolff, 2015)

Unabhängig vom gewählten Ansatz sollte darauf geachtet werden, dass das UI vom selben Team entwickelt wird, welches auch für die jeweilige Fachlichkeit zuständig ist und diese ebenfalls in den Microservice implementiert. Wird das UI von einem anderen Team entwickelt, kann dies zu ungewollten Abhängigkeiten zwischen den Teams führen. (Wolff, 2015)

Um eine gemeinsame Benutzerschnittstelle für ein System bestehend aus mehreren Microservices zu gestalten, bieten sich verschiedene Strategien an.

Das UI kann wie eingangs beschrieben als Teil des Microservice bereitgestellt werden und somit den fachlichen Use-Case des Microservices abbilden (Röwekamp & Limburg, 2016). Dabei kann jeder Microservice eine eigene Single-Page-Application (SPA) bereitstellen, welche innerhalb einer HTML-Seite die gesamte Logik des Microservices mit Javascript implementiert. Der Wechsel zwischen den einzelnen SPA kann mit einfachen Links realisiert werden, wobei hier zu berücksichtigen ist, dass beim Aufruf eine neue SPA geladen werden muss und dies unter Umständen lange dauern kann. Statt einer SPA können auch einfache HTML-basierte Oberflächen pro Microservice bereitgestellt werden, welche die benötigten Funktionalitäten beinhalten. (Wolff, 2015) Da in diesem Fall jedes Microservice seine eigene Benutzerschnittstelle bereitstellt, kann dies zu Unterschieden in der Darstellung der gesamten Benutzeroberfläche führen. Um dem entgegenzuwirken, sollten Regeln definiert werden, die jedes UI einhalten muss sowie Design-Ressourcen bereitgestellt werden, die zu einem einheitlichen Aussehen führen. (Röwekamp & Limburg, 2016) Weiters sollte eine einheitliche Lösung zur Authentifizierung und Autorisierung der Benutzer gewählt werden, da ansonsten beim Aufruf einer anderen SPA ein erneutes Anmelden notwendig sein kann (Wolff, 2015).

Als Alternative für den soeben genannten Ansatz beschreibt Wolff (2015) den Einsatz einer SPA für alle Microservices. Dabei sollte eine Technologie gewählt werden, welche eine Modularisierung der SPA ermöglicht, um pro Microservice ein eigenes Modul entwickeln zu können. Jedoch gilt es zu beachten, dass die Module sich gegenseitig aufrufen können und somit ungewollte Abhängigkeiten zwischen den Microservices entstehen. Weiters muss die SPA als eigenes Artefakt bereitgestellt werden, was zu einer erhöhten Kommunikation zwischen den

einzelnen Teams führen kann und somit wichtige Vorteile von Microservices-Architekturen verloren gehen können.

Eine weitere Strategie, um eine gemeinsame Benutzeroberfläche bereitzustellen, ist der Zusammenbau des UI aus HTML-Fragmenten, die von den Microservices geliefert werden. Dabei kann ein Frontend-Server eingesetzt werden, welcher die Kommunikation mit den Services übernimmt, das UI zusammenstellt und dieses dem Benutzer zur Verfügung stellt. (Newman, 2015) Ein Vorteil dieser Variante ist das einheitliche Aussehen der Benutzerschnittstelle, da die einzelnen Fragmente lediglich HTML-Code liefern und der Frontend-Server die Design-Ressourcen hinzufügen kann (Wolff, 2015).

Newman (2015) bezeichnet diesen Ansatz als *UI Fragment Composition*, welcher in Abbildung 3-14 beispielhaft dargestellt wird.

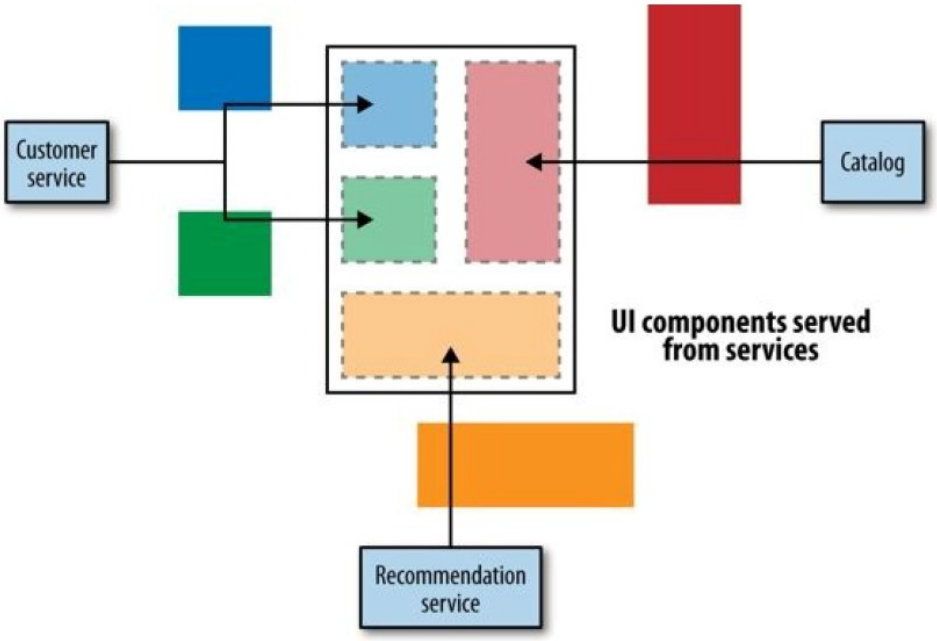

Abbildung 3-14: UI Fragment Composition (Newman, 2015)

Die bisher genannten Möglichkeiten eignen sich für webbasierte Applikationen, für welche lediglich ein Browser als Client benötigt wird. Müssen jedoch weitere Clients bedient werden und Benutzerschnittstellen für beispielsweise mobile Endgeräte oder Desktop-Anwendungen bereitgestellt werden, können die zuvor genannten Vorgehensweisen nur bedingt eingesetzt werden.

Für diese Anwendungsfälle eignet sich der Ansatz *Backend-for-Frontends*, bei welchen ein eigenes Microservice eingesetzt wird, um die Kommunikation zwischen Client und anderen Microservices zu zentralisieren. Dieses Microservice wird auch als API-Gateway bezeichnet und sollte selbst keine Fachlichkeit implementieren. (Newman, 2015) Da in der Regel ein eigenes Team für die Erstellung von mobilen Clients beauftragt ist, sollte das API-Gateway auch von diesem Team erstellt werden (Wolff, 2015).

Auch auf Ebene der UI ist eine Integration eines Legacy-Systems möglich. Eine Möglichkeit dafür ist die Einbindung der von Microservices gelieferten HTML-Fragmente in die bestehende

Benutzerschnittstelle. Weiters kann der zuvor bei SPA beschriebene Ansatz verwendet werden, um auf Basis von Links zwischen der Legacy-Anwendung und neuen UIs zu wechseln. (Newman, 2015)

3.4 Zusammenfassung

In diesem Kapitel wurden Möglichkeiten zur Ablöse, Erweiterung und Modernisierung von monolithischen Legacy-Systemen untersucht.

Eingangs wurde der Lebenszyklus von Software-Systemen betrachtet, welcher sich aus drei Phasen zusammensetzt und zu einem wesentlichen Teil aus der Evolutionsphase besteht, in welcher ein System gewartet und weiterentwickelt wird. Diese Phase verursacht in der Regel über die Lebensdauer einer Software zwei Drittel der gesamten Kosten, wobei der Großteil davon gewöhnlicherweise für die Erweiterung des Systems um neue Funktionalitäten aufgewendet wird. Daher sollte bereits beim Entwurf eines Systems die Änderbarkeit und Erweiterbarkeit berücksichtigt werden, um die in der Evolutionsphase anfallenden Wartungskosten zu reduzieren.

Eine schrittweise Erweiterung eines Systems hat in der Regel ein geringeres Risiko und sollte somit gegenüber einer Neuentwicklung bevorzugt werden. Dabei können die Migrationsstrategien Kapselung, Konvertierung und Reengineering eingesetzt werden, wobei sich vor allem Letzteres für eine inkrementelle Vorgehensweise eignet.

Zur Durchführung der schrittweisen Ablöse oder Erweiterungen eines Systems finden sich in der Literatur verschiedene Ansätze. Die Strategien *Chicken-Little* und *Butterfly* legen den Fokus auf die Überführung und Anpassung der Daten an die Strukturen des Zielsystems. Bei den Methoden *Split Backend and Frontend* und *Change by Split* wird die Programmlogik in den Vordergrund gestellt und zeigen, wie ein System grundsätzlich in mehrere Bestandteile geteilt werden kann. Der Ansatz *Branch by Abstraction* zeigt eine Vorgehensweise, um eine Komponente innerhalb eines Systems zu ersetzen, ohne dabei die Funktionsfähigkeit der Anwendung zu beeinträchtigen. Die *Strangler Application* und *Extract Services* Methoden eignen sich besonders für die Einführung von Microservices und nützen einen vorgeschalteten Proxy, um Anfragen an ein Legacy-System auf ein Service umzuleiten.

Nachdem geeignete Strategien für die Durchführung einer Migration gefunden wurden, müssen passende Schnittpunkte ermittelt werden, um ein monolithisches System aufzuteilen und mittels Microservices zu erweitern. Diese Aufteilung sollte sich an der fachlichen Domäne orientieren und nicht auf technische Aspekte fokussieren. Ein geeignetes Werkzeug dafür ist das Domain-Driven Design, welches entsprechende Muster und Bausteine für die Modellierung der Fachdomäne beinhaltet.

Um DDD im Umfeld von Legacy-Systemen einzusetzen und diese durch eigenständige Services zu erweitern, wurden im weiteren Verlauf dieses Kapitels die Ansätze *Bubble Context* und *Autonomous Bubble* vorgestellt.

Abschließend wurden mögliche Auswirkungen der Aufteilung eines Systems auf das Datenmanagement und die Benutzerschnittstelle diskutiert.

Die Einführung von eigenständigen Microservices bedingt für gewöhnlich ein dezentrales Datenmanagement und kann zu verteilten Transaktionen führen, welche jedoch fehleranfällig und schwer handhabbar sein können. Um dem entgegenzuwirken, sollte in einem verteilten System das BASE-Prinzip zum Einsatz kommen und eine ereignisgesteuerte Kommunikation durchgeführt werden. Bei diesen Ansätzen wird der Verfügbarkeit eines Systems ein höherer Stellenwert als der Konsistenz der Daten zugeordnet. Welcher Ansatz beim Datenmanagement gewählt werden kann, ist abhängig von der Fachlichkeit und ob diese nach dem *Eventual Consistency* Prinzip abgebildet werden kann.

4 EMPIRISCHER TEIL

In diesem Kapitel wird die empirische Vorgehensweise dieser Arbeit behandelt, welche sich aus drei Teilen zusammensetzt. Zu Beginn werden basierend auf den Erkenntnissen der Literaturrecherche Hypothesen gebildet und diese operationalisiert. Anschließend wird im zweiten Teil die Ausarbeitung des Fragebogens und Durchführung der Befragung erläutert, um in weiterer Folge dessen Ergebnisse auszuwerten und zu interpretieren.

Im abschließenden Teil wird die Durchführung der Marktbetrachtung beschrieben und diskutiert. Bei dieser werden Best-Practice-Ansätze für die Einführung von Microservices eruiert und diese in weiterer Folge den Ergebnissen der Befragung gegenübergestellt. Dadurch soll geprüft werden, ob sich in der Praxis Ansätze finden, die den Erwartungen der befragten Personen entsprechen.

4.1 Hypothesen

Ausgehend von den Erkenntnissen aus der in den vorherigen Kapiteln durchgeführten Literaturrecherche, werden in diesem Abschnitt Annahmen hinsichtlich der Erwartungshaltung an Microservices in Form von Hypothesen gebildet. Anschließend werden diese Hypothesen operationalisiert und aus den relevanten theoretischen Begriffen messbare Merkmale und Merkmalsausprägungen abgeleitet. Haben Merkmale mindestens zwei Ausprägungen, werden diese als Variable bezeichnet und können je nach Art nach qualitativen und quantitativen Merkmalen differenziert werden (Brüsemeister, 2008).

Porst (2014) weist darauf hin, dass bei der Formulierung von Fragen und Antworten diese Variablen berücksichtigt werden müssen, um dadurch valide und zuverlässige Informationen erfassen zu können.

Nachfolgend werden Hypothesenpaare bestehend aus jeweils einer Alternativhypothese (H1) und einer komplementären Nullhypothese (H0) aufgestellt, davon Variablen abgeleitet und beschrieben sowie die zugehörigen Fragen definiert. Der vollständige Fragebogen findet sich im Anhang A, wobei sich die bei den jeweiligen Fragen angegebene Nummerierung im angehängten Fragebogen wiederfindet.

4.1.1 Hypothese 1

Zur Erweiterung und Modernisierung von monolithischen Legacy-Systeme finden sich verschiedene Ansätze im Zuge der Softwareevolution. Richardson und Smith (2016) bezeichnen Microservices als eine dafür geeignete Alternative.

Microservices sollen dabei helfen, die Komplexität eines Systems beherrschbar zu machen und dieses nach fachlichen Aspekten zu teilen. Damit einhergehend entsteht eine verteilte Systemarchitektur, welche für gewöhnlich zu einer Erhöhung der technischen Komplexität des Gesamtsystems führt. (Fowler & Lewis, 2015)

Mit der nachfolgenden Hypothese soll geprüft werden, ob die Komplexität von monolithischen Systemen einer der Beweggründe für die Einführung von Microservices ist.

H1: *Je höher die Komplexität eines monolithischen Altsystems ist, desto höher ist die Bereitschaft, dass bei einer Modernisierung des Systems Microservices verwendet werden.*

H0: *Die Komplexität eines monolithischen Altsystems hat keinen Einfluss auf die Bereitschaft, bei einer Modernisierung des Systems Microservices zu verwenden.*

Bei dieser Hypothese wurden die Variablen „Komplexität“ und „Bereitschaft Microservices einzusetzen“ identifiziert, welche im Zuge nachfolgender Fragen gemessen werden sollen:

- Wenn Sie monolithische Softwarelösungen in Ihrem Arbeitsumfeld betrachten, wie oft treten Probleme hinsichtlich einer hohen Komplexität auf? (#2.5)
- Planen Sie den Microservices-Ansatz bei einem Ihrer nächsten Projekte einzusetzen? (#12)
- Ist eine hohe Komplexität einer der Hauptgründe, um ein monolithisches System abzulösen/zu modernisieren? (#20.4)
- Planen Sie aktuell eine Ablöse/Modernisierung einer bestehenden Software-Applikation? (#21)

Die erste Frage wird mittels einer Ordinalskala bestehend aus den Kategorien „oft“, „gelegentlich“, „selten“ und „nie“ bewertet, die zweite und dritte Frage werden mittels einer binären Variable operationalisiert und können jeweils mit „Ja“ oder „Nein“ beantwortet werden. Diese beiden Fragen sollen zur Messung der Variable „Komplexität“ herangezogen werden.

Die Variable „Bereitschaft Microservices einzusetzen“ soll im Zuge der letzten angeführten Frage gemessen werden, wobei diese auf einer Nominalskala bestehend aus nachfolgenden Werten beantwortet werden soll:

- „Ja, Microservices werden eingesetzt“
- „Ja, Einsatz von Microservices wird evaluiert“
- „Ja, Microservices werden nicht eingesetzt“
- „Nein, aktuell nicht geplant“
- „Nicht bekannt“

4.1.2 Hypothese 2

Wie in Kapitel 3.1 ausführlich diskutiert, wird ein großer Teil der Aufwände und Kosten während der Lebensdauer einer Software durch die Wartung des Systems verursacht. Die folgende Hypothese soll prüfen, ob dadurch eine Erhöhung der Bereitschaft zur Einführung von Microservices gegeben ist:

H1: *Je mehr die Wartbarkeit eines monolithischen Altsystems eingeschränkt ist und je höhere Kosten dabei verursacht werden, desto höher ist die Bereitschaft, bei einer Modernisierung das Microservices-Architekturparadigma zu verwenden.*

H0: *Die Wartbarkeit eines monolithischen Altsystems und die dabei verursachten Kosten haben keine Auswirkung auf die Bereitschaft, bei einer Modernisierung das Microservices-Architekturparadigma zu verwenden.*

Von dieser Hypothese wurden die Variablen „eingeschränkte Wartbarkeit“ und „Wartungskosten“ abgeleitet, deren Messung mittels folgender Fragen durchgeführt und jeweils mit „Ja“ oder „Nein“ beantwortet werden sollen:

- Ist eine eingeschränkte Wartbarkeit einer der Hauptgründe, um ein monolithisches System abzulösen/zu modernisieren? (#20.2)
- Sind hohe Wartungs-/Betriebskosten eine der Hauptgründe, um ein monolithisches System abzulösen/zu modernisieren? (#20.3)

4.1.3 Hypothese 3

Eine verteilte Systemarchitektur wie dies beispielsweise bei Microservices der Fall ist, führt für gewöhnlich zu einer Erhöhung der technischen Komplexität des Systems. Erfahrene Teams mit den benötigten Kompetenzen trauen sich in der Regel eher zu, diese erhöhte Komplexität zu beherrschen und tendieren daher eher dazu, neue Technologien und Architekturansätze einzuführen. (Fowler & Lewis, 2015)

Mit der nachfolgenden Hypothese soll überprüft werden, ob die Entscheidung für einen Einsatz von Microservices mit dem Erfahrungslevel einer Person in Verbindung gebracht werden kann.

H1: *Je mehr Erfahrung eine Person mit verteilten Systemen hat, desto eher werden Microservices bei zukünftigen Projekten eingesetzt.*

H0: *Unabhängig von der Erfahrung einer Person mit verteilten Systemen, wird der Einsatz von Microservices bei zukünftigen Projekten beabsichtigt.*

Bei dieser Hypothese wurde die Variable „Erfahrung mit verteilten Systemen“ identifiziert, welche im Zuge folgender Fragen gemessen wird:

- Sind Sie mit Service-orientierten Architekturen / verteilten Systemen vertraut? (#3.1)
- Sind Sie mit dem Thema Microservices vertraut? (#7)
- Haben Sie bereits praktische Erfahrungen mit Microservices? (#8)

Die Bewertung der ersten dieser drei Fragen erfolgt anhand einer Nominalskala bestehend aus drei Kategorien: „Ja, in der Praxis eingesetzt“, „Ja, theoretische Kenntnisse“ sowie „Nein“ werden als Antwortmöglichkeiten angeboten. Die zweite Frage kann mit „Ja“ oder „Nein“ beantwortet werden, die Operationalisierung der dritten Frage erfolgt mittels folgenden Antwortmöglichkeiten:

- Ja, Microservices sind produktiv im Einsatz
- Ja, Microservices in Entwicklung

- Nein, Evaluierung läuft
- Nein, keine Erfahrung mit Microservices
- Kein Interesse an Microservices

4.1.4 Hypothese 4

Microservices ermöglichen es, einzelne Bestandteile eines Systems unabhängig von anderen auszutauschen, um dadurch schnell auf geänderte Anforderungen reagieren zu können. Um diese Reaktionszeit zu verbessern, sollten Teams unabhängig arbeiten können und die Verantwortung für die Entwicklung, den Betrieb und die Qualitätssicherung von Services übertragen bekommen. Dafür werden neben einer technischen Infrastruktur, die einen hohen Automatisierungsgrad für die Bereitstellung von Services aufweist, vor allem organisatorische Strukturen benötigt, welche die Zusammenstellung solcher Teams ermöglichen und das Unternehmen dazu bereit ist, die Verantwortung an diese zu übertragen. (Wolff, 2015)

Folgende Annahme geht davon aus, dass Unternehmen dazu bereit sind, organisatorische Änderungen durchzuführen, um in Folge mittels Microservices eine Verbesserung der Reaktionszeit auf geänderte Marktbedingungen erzielen zu können:

H1: *Je länger die Bereitstellung eines monolithischen Systems dauert, desto eher sind Unternehmen dazu bereit, die für den Einsatz von Microservices notwendigen organisatorischen Änderungen durchzuführen, um eine Verbesserung der Reaktionszeit auf geänderte Marktanforderungen herbeizuführen.*

H0: *Unternehmen sind nicht dazu bereit, die für den Einsatz von Microservices notwendigen organisatorischen Änderungen durchzuführen, um eine schnellere Bereitstellung eines monolithischen Systems herbeizuführen.*

Aus dieser Hypothese wurden die Variablen „Bereitschaft organisatorische Änderungen durchzuführen“ und „Verbesserung der Reaktionszeit“ abgeleitet. Die Messung der ersten Variable erfolgt anhand folgender Fragen, welche jeweils mit „Ja“ oder „Nein“ beantwortet werden können:

- Erfolgt in Ihrem Arbeitsumfeld eine organisatorische Trennung von Entwicklung und Betrieb? (#16)
- Besteht in Ihrem Arbeitsumfeld die Bereitschaft organisatorische Änderungen durchzuführen, um neue technologische Ansätze einzuführen? (#18)
- Der Microservices-Ansatz ermöglicht die Umsetzung selbstständiger, voneinander unabhängiger Services. Würden Sie die Verantwortung für Entwicklung, Betrieb und Qualitätssicherung dieser Services an einzelne Teams übertragen? (#19)
- Treffen Sie in ihrem Arbeitsumfeld organisatorische Entscheidungen? (#26)

Die zweite Variable „Verbesserung der Reaktionszeit“ wird mit diesen Fragen bewertet:

- Wenn Sie monolithische Softwarelösungen in Ihrem Arbeitsumfeld betrachten, wie oft treten Probleme hinsichtlich langen Deployment-Zyklen auf? (#2.7)
- In welchen Bereichen erwarten Sie sich die meisten Vorteile durch Microservices? Verbesserung der Reaktionszeit auf geänderte Marktbedingungen (#9.6)

Die erste Frage kann mittels der Antwortkategorien „oft“, „gelegentlich“, „selten“ und „nie“ bewertet werden, die zweite mit einer „Ja“ oder „Nein“ Auswahl.

4.1.5 Hypothese 5

Aus der theoretischen Ausarbeitung kann abgeleitet werden, dass bei der Verwendung von Microservices ein dezentrales Datenmanagement eingeführt werden sollte, da ansonsten Vorteile von Microservices wie beispielsweise unabhängige Entwicklung und Betrieb verloren gehen. Dies bedeutet, dass auch eine Modernisierung eines Legacy-Systems mittels Microservices zu einem verteilten System mit getrennter Datenhaltung führen sollte und dadurch möglicherweise Daten redundant gespeichert werden müssen.

Anhand des nachfolgenden Hypothesenpaares soll ermitteln werden, ob in der Praxis eine Bereitschaft zur Einführung eines dezentralen Datenmanagements gegeben ist oder ob dadurch eine Einführung von Microservices eher abgelehnt wird.

H1: *Je öfter die Verfügbarkeit eines monolithischen Altsystems eingeschränkt ist, desto eher ist die Bereitschaft zur Einführung eines dezentralen Datenmanagements gegeben.*

H0: *Eine eingeschränkte Verfügbarkeit eines monolithischen Altsystems hat keine Auswirkung auf die Bereitschaft zur Einführung eines dezentralen Datenmanagements.*

Bei dieser Hypothese wurden die Variablen „Verfügbarkeit“ und „Bereitschaft zur Einführung eines dezentralen Datenmanagements“ festgelegt. Für die Bewertung dieser Variablen werden folgende Fragen herangezogen:

- Wenn Sie monolithische Softwarelösungen in Ihrem Arbeitsumfeld betrachten, wie oft treten Probleme hinsichtlich eingeschränkter Verfügbarkeit auf? (#2.1)
- Ist in Ihrem Arbeitsumfeld die Bereitschaft gegeben, dezentrales Datenmanagement einzuführen und möglicherweise redundante Daten zu speichern? (#5)
- Ist in Ihrem Arbeitsumfeld die Bereitschaft gegeben, Abstriche bei der Konsistenz von Daten hinzunehmen, um dadurch die Verfügbarkeit von Systemen zu erhöhen? (#6)

Bei der ersten Frage wird eine Skala bestehend aus den Werten „oft“, „gelegentlich“, „selten“ und „nie“ verwendet, die Beantwortung der beiden weiteren Fragen erfolgt jeweils anhand einer vierteiligen Intervallskala mit den Werten „Ja“, „Eher ja“, „Eher nein“ und „Nein“.

4.2 Befragung

Neben der Prüfung der zuvor festgelegten Hypothesen soll die Befragung vor allem Erkenntnisse über aktuelle Probleme bei monolithischen Softwarelösungen und die an Microservices gestellten Erwartungen liefern sowie die Bereitschaft zur Veränderung in den jeweiligen Unternehmen untersuchen. Um eine größere Population bei der Stichprobenziehung zu erreichen, wurde für die Befragung ein quantitativer Forschungsansatz gewählt und ein standardisierter Fragebogen einzelnen Experteninterviews vorgezogen.

Diese Form der schriftlichen Befragung hat unter anderem die Vorteile, dass dafür ein geringerer Personal- und Zeitaufwand notwendig ist, die befragten Personen nicht durch das Verhalten des Interviewers beeinflusst werden und sich mehr Zeit nehmen können, die Fragen durchzudenken und zu beantworten. Als Nachteile gelten vor allem die fehlende Kontrolle bei der Durchführung und dass es bei den Befragten zu Verständnisproblemen bei der Beantwortung kommen kann. (Raithel, 2008)

Nach Porst (2014) soll ein Fragebogen auf zuvor erarbeiteten, theoretischen Konzepten basieren und einen expliziten Zusammenhang zwischen den Fragen, der Theorie und den aufgestellten Hypothesen herstellen. Die Fragen müssen so gewählt werden, dass diese auf die zuvor durch Operationalisierung ermittelten Variablen verweisen (Brüsemeister, 2008).

Nachfolgend wird der Aufbau und Inhalt des Fragebogens im Detail beschrieben, welcher ausgehend von den im vorherigen Abschnitt definierten Fragen gestaltet wurde.

4.2.1 Aufbau und Inhalt

Porst (2014) weist in seinem Buch darauf hin, dass der Aufbau eines Fragebogens entscheidend für die Qualität der Ergebnisse sein kann und Fragen so gestaltet werden sollen, dass diese die interviewten Personen nicht überfordern. Weiters ist auch das Layout des Fragebogens entscheidend; der Fragebogen sollte unter anderem einheitlich dargestellt und klar strukturiert sein sowie eine gute Lesbarkeit aufweisen.

Bei den Fragen kann zwischen drei Arten differenziert werden: offene, halboffene sowie geschlossene Fragen. Bei der geschlossenen Frage werden fixe Antwortmöglichkeiten vorgegeben, aus denen die Befragten wählen müssen. Offene Fragen erfordern eine Beantwortung in Form einer selbst formulierten Antwort. Halboffene Frage sind eine Mischung aus den beiden zuvor genannten und beinhalten neben vorgegebenen Antworten die Möglichkeit, eine eigene Antwort zu formulieren. (Raithel, 2008)

Bei der Ausarbeitung des vorliegenden Fragebogens wurden alle drei genannten Arten von Fragen eingesetzt, wobei es sich bei der Mehrheit um geschlossene beziehungsweise halboffene Fragen handelt. Offene Fragen wurden gezielt eingesetzt, um der interviewten Person die Möglichkeit zu bieten, detaillierte Sichtweisen zu einem Thema preiszugeben. Diese sollen in weiterer Folge dazu dienen, zusätzliche Erkenntnisse über die Erwartungshaltung an Microservices zu gewinnen.

Der ausgearbeitete Fragebogen findet sich im Anhang dieser Arbeit. Insgesamt werden den teilnehmenden Personen 32 Fragen gestellt, welche nach Themen geordnet in folgende sieben Gruppen eingeteilt wurden:

- Allgemeine Informationen
- Software Allgemein
- Erwartungshaltung Microservices
- Betrieb / Infrastruktur
- Organisation
- Legacy-Anwendungen
- Abschluss

Zu Beginn werden in der ersten Fragengruppe „Allgemeine Informationen“ Daten zu den teilnehmenden Personen ermittelt. Diese sind die jeweilige berufliche Position, der Erfahrungslevel in der Softwareentwicklung sowie die Größe des Unternehmens beziffert anhand der Anzahl an Angestellten.

Die zweite Gruppe beinhaltet sechs Fragen zu allgemeinen Themen der Softwareentwicklung wie beispielsweise den subjektiven Stellenwert verschiedener Qualitätsmerkmale und Probleme, die bei monolithischen Systemen auftreten. Zur Bewertung der Qualität einer Software kann dabei nach Sneed (2003) auf die im ISO-9126 Standard definierten Merkmale zurückgegriffen werden. Weiters werden Fragen zu verteilten Systemen und dezentralem Datenmanagement gestellt sowie die Bereitschaft hinterfragt, ob *Eventual Consistency* im jeweiligen Arbeitsumfeld eingesetzt werden kann.

Die dritte Gruppe umfasst Fragen, welche neben dem Erfahrungslevel vor allem die Erwartungshaltung an Microservices sowie den Stellenwert verschiedener Vor- und Nachteile dieses Architekturansatzes ermitteln soll. Die nächsten beiden Gruppen behandeln die Themen „Betrieb / Infrastruktur“ sowie „Organisation“. Dabei werden neben dem Grad der Automatisierung und dem Einsatz verschiedener Technologien die Auswirkungen auf die Organisation hinterfragt.

Die Gruppe „Legacy-Anwendungen“ beinhaltet Fragen, welche aktuelle Probleme von monolithischen Systemen sowie die Bereitschaft zur Ablöse oder Erweiterung dieser Systeme erheben soll. Weiters werden in dieser sowie in der nachfolgenden letzten Gruppe offene Fragen gestellt, aus welchen Rückschlüsse auf die Vollständigkeit der theoretischen Betrachtung gezogen werden sollen. Anhand deren Beantwortung werden zusätzliche Erkenntnisse über alternative Möglichkeiten zur Ablöse oder Erweiterung monolithischer Systeme beziehungsweise ein Informationsgewinn über den Bekanntheitsgrad der im Zuge der theoretischen Ausarbeitung vorgestellten Strategien erwartet. Diese offenen Fragen sind allesamt als optional definiert und müssen daher von den Teilnehmenden nicht zwingend befüllt werden.

4.2.2 Durchführung

Der Fragebogen wurde mittels „LimeSurvey“[4], eine unter einer Open Source Lizenz zur Verfügung stehende Umfrage-Software, erstellt und online zur Verfügung gestellt.

In Anlehnung an Raithel (2008) wurde ein Pretest durchgeführt, bei welchem der Fragebogen vorab an ausgewählte Personen übermittelt wurde, um dadurch unter anderem die Verständlichkeit der Fragestellungen, deren Reihenfolge sowie die Benutzbarkeit des eingesetzten Bewertungsinstruments zu prüfen. Die aus diesem Pretest eingegangen Rückmeldungen wurden bewertet und in den vorliegenden Fragebogen eingearbeitet.

Die Umfrage wurde im Zeitraum vom 24.02.2017 bis einschließlich 03.03.2017 durchgeführt. Als Zielgruppe wurden Personen definiert, die in der Softwareentwicklung tätig sind, entweder als Entwicklerin oder Entwickler, in der Software-Architektur oder im Management. Die Teilnehmerinnen und Teilnehmer wurden gezielt ausgesucht, wobei darauf geachtet wurde, Personen aus Unternehmen unterschiedlicher Größe und von verschiedenen beruflichen Positionen in der Softwareentwicklung auszuwählen. Ein Link zur Umfrage wurde per E-Mail übermittelt und durch eine persönliche Anrede zur Teilnahme motiviert. Weiters wurden die Personen dazu eingeladen, den übermittelten Link an Personen in ihrem Arbeitsumfeld weiterzuleiten, welche Interesse an der Softwareentwicklung zeigen.

4.2.3 Auswertung

In diesem Abschnitt wird die Auswertung der Umfrage beschrieben und die Ergebnisse präsentiert, die Interpretation der Resultate erfolgt im nachfolgenden Abschnitt. Die gesammelten Ergebnisse in tabellarischer Form finden sich im Anhang dieser Arbeit.

Die Auswertung der Fragen erfolgt nach einem deskriptiven Verfahren, wobei zur Beschreibung der erhobenen Daten in Anlehnung an Raithel (2008) die Häufigkeitsverteilung angewendet wird.

Antworten auf offene Fragen werden einer computergestützten Inhaltsanalyse unterzogen. Dabei werden in einem ersten Schritt die Ergebnisse gesichtet und bei Bedarf gefiltert. Anschließend erfolgt eine Kategorienbildung, wobei sowohl auf die Theorie als auch die erhaltenen Antworttexte selbst zurückgegriffen wird. Diese Kategorien bilden die Grundlage für die in Folge durchzuführende Codierung der Ergebnisse, bei der Textstellen den entsprechenden Kategorien zugeordnet werden. Abschließend wird im Zuge einer Häufigkeitsanalyse festgestellt, wie oft eine Kategorie von den teilnehmenden Personen berücksichtigt wurde, um daraus Rückschlüsse auf die vorliegende Fragestellung dieser Arbeit ziehen zu können. (Züll & Mohler, 2001)

An der Umfrage nahmen im genannten Zeitraum gesamt 81 Personen teil, wobei 66 Personen den Fragebogen vollständig ausgefüllt haben. Wie bei Raithel (2008) beschrieben, sollte vor der Durchführung der Bewertung eine Filterung der Ergebnisse stattfinden, bei welcher unvollständig ausgefüllte Fragebögen oder Antworten in sehr schlechter Ausfüllqualität bereinigt werden. Diese

[4] https://www.limesurvey.org/

Vorgehensweise wurde auf die vorliegenden Ergebnisse angewendet, wobei 15 unvollständig ausgefüllte Rücksendungen aus der Ergebnismenge entfernt wurden. Bei der Analyse hinsichtlich Qualität der Antworten konnten keine Fragebögen identifiziert werden, welche erkennbare Ausfüllmuster oder eine unsachliche Beantwortung der offenen Fragen aufweisen. Dadurch mussten keine weiteren Antworten bereinigt werden, womit sich eine Stichprobengröße von n = 66 ergibt.

Die Befragten verteilen sich auf die Zielgruppen, dem Erfahrungslevel in der Softwareentwicklung sowie der Unternehmensgröße wie in Tabelle 4-1 ersichtlich.

Merkmale	Antworten	Anzahl	Prozent
Berufsgruppe	Softwareentwicklung	31	47,0%
	Software-Architektur	20	30,3%
	IT-Management	13	19,7%
	Sonstige	2	3,0%
Erfahrungslevel in der Softwareentwicklung	weniger als 3 Jahre	5	7,6%
	3 bis 10 Jahre	27	40,9%
	mehr als 10 Jahre	34	51,5%
Anzahl Mitarbeiterinnen und Mitarbeiter im Unternehmen	< 10	10	15,2%
	10 bis 99	33	50,0%
	100 bis 499	9	13,6%
	> 500	14	21,2%

Tabelle 4-1: Merkmale der teilnehmenden Personen

Weiters haben 52 Personen (78,8%) angegeben, technische Entscheidungen im jeweiligen Arbeitsumfeld zu treffen. Organisatorische Entscheidungen im Arbeitsumfeld werden hingegen lediglich von 39,4% der Befragten getroffen.

Nachfolgend werden die Ergebnisse der einzelnen Fragegruppen der Reihe nach beschrieben. Wie bereits in den Absätzen zuvor, werden auch die in Folge angeführten Prozentwerte gerundet und in der Regel mit einer Kommastelle dargestellt.

Software Allgemein

Bei der Befragung nach dem Stellenwert ausgewählter Qualitätsmerkmale wurde die Wartbarkeit am wichtigsten bewertet, alle Befragten stuften diese als sehr wichtig (81,8%) oder eher wichtig (18,2%) ein. Weiters bekamen die Verfügbarkeit, Benutzbarkeit und Funktionalität einen hohen Stellenwert zugesprochen. Als am wenigsten wichtig wurden die Qualitätsmerkmale Wiederverwendbarkeit und Austauschbarkeit von den Befragten eingestuft. Dieses Ergebnis wird in Abbildung 4-1 im Detail dargestellt.

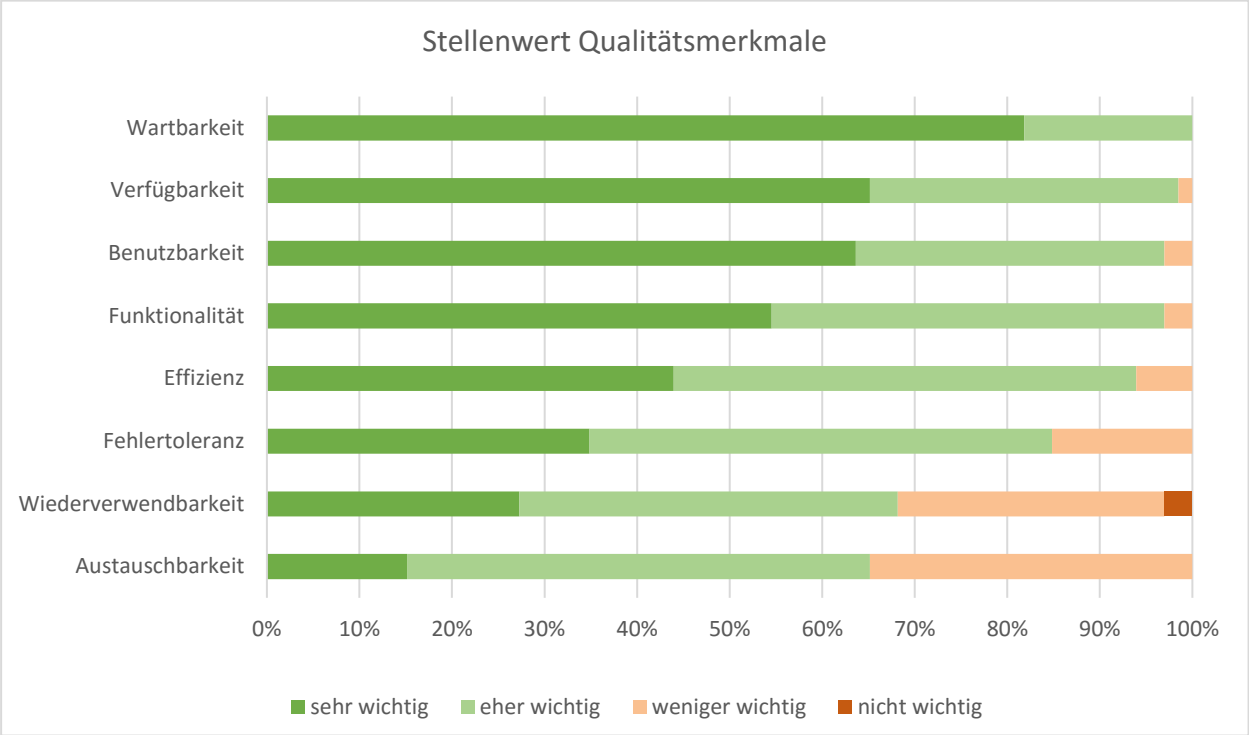

Abbildung 4-1: Stellenwert ausgewählter Qualitätsmerkmale von Software

Häufig auftretende Probleme bei monolithischen Anwendungen sind eine unzureichende Testabdeckung, hohe Komplexität des Systems, eingeschränkte Wartbarkeit und der Einsatz veralteter Technologien. Diese Merkmale wurden von jeweils mehr als 50% der Befragten als oft eintretende Schwierigkeiten beurteilt. Eher selten treten Probleme aufgrund einer eingeschränkten Verfügbarkeit oder hoher Fehlerraten ein. Eine detaillierte Verteilung dieser Antworten findet sich in Abbildung 4-2.

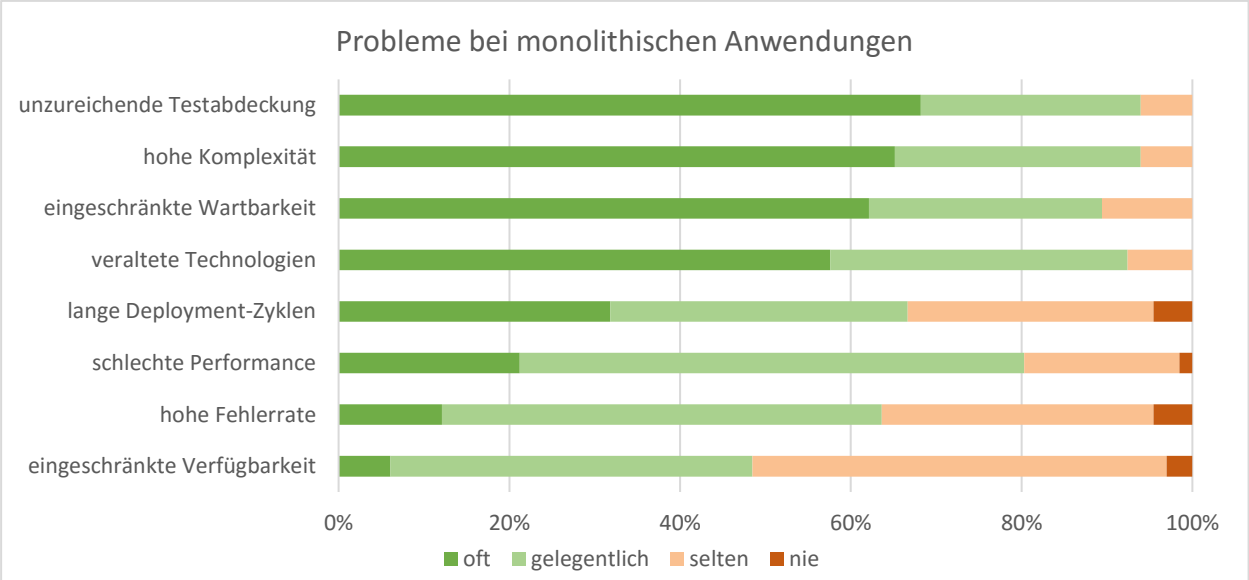

Abbildung 4-2: Häufigkeit von Problemen monolithischer Anwendungen

Weiters wurden die teilnehmenden Personen hinsichtlich ihrer Kenntnisse und Erfahrungen mit verschiedenen aus der Theorie abgeleiteten Begrifflichkeiten rund um verteilte Systeme befragt.

Dabei konnte festgestellt werden, dass verteilte Systeme eine weite Verbreitung haben und die befragten Personen mit diesen vertraut sind. Auch Domain-Driven Design und ereignisgesteuerte Architekturen sind den meisten bekannt und werden auch vermehrt in der Praxis eingesetzt. Weniger verbreitet ist der *Design for Failure* Ansatz und das in Kapitel 3.3.3 diskutierte CAP-Theorem. Nachfolgende Abbildung 4-3 stellt diese Ergebnisse grafisch dar.

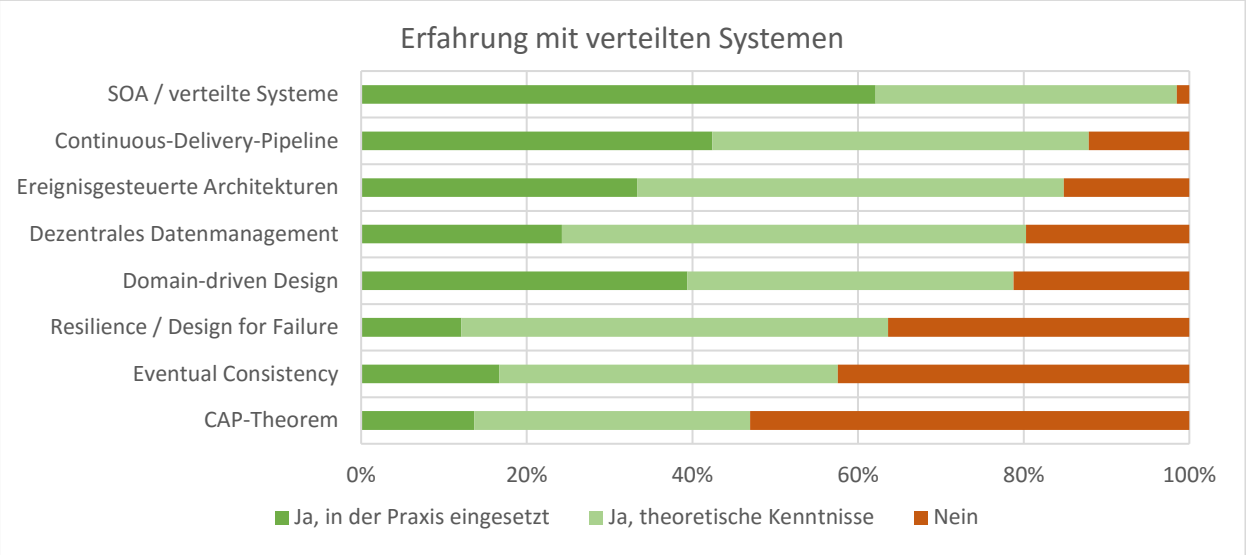

Abbildung 4-3: Kenntnisse und Erfahrungen mit Begrifflichkeiten von verteilten Systemen

Bei 59% der teilnehmenden Personen sind im Arbeitsumfeld verteilte Systeme im Einsatz. Die Frage nach der Bereitschaft zur Einführung eines dezentralen Datenmanagements und dadurch einem möglichen redundanten Speichern von Daten, wurde von mehr als der Hälfte der teilnehmenden Personen mit ja (29%) oder eher ja (32%) beantwortet. Die Bereitschaft, Abstriche bei der Konsistenz von Daten hinzunehmen, um dadurch die Verfügbarkeit eines Systems zu erhöhen, wurde hingegen von lediglich 15% mit ja beantwortet. Die detaillierten Ergebnisse finden sich in der nachfolgenden Abbildung 4-4 sowie Abbildung 4-5.

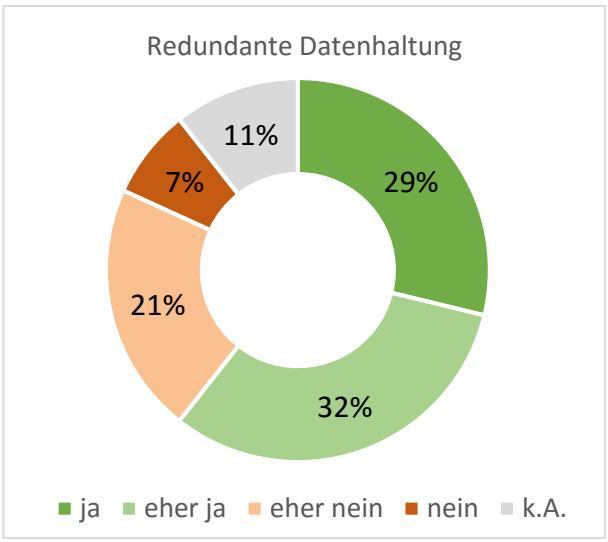

Abbildung 4-4: Bereitschaft für redundante Datenhaltung

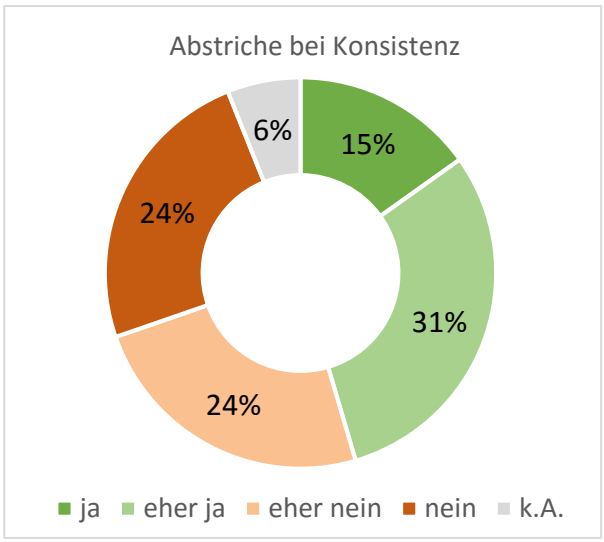

Abbildung 4-5: Bereitschaft, Abstriche bei der Konsistenz von Daten hinzunehmen

Erwartungshaltung Microservices

Im Zuge der dritten Fragegruppe „Erwartungshaltung Microservices“ gaben 54 Personen (81,8%) an, mit Microservices vertraut zu sein und knapp 58% wollen diesen Architekturansatz bei zukünftigen Projekten einsetzen. Produktiv im Einsatz oder bereits in Entwicklung sind Microservices bei jeweils 15 der teilnehmenden Personen (22,8%). Bei 21,2% befinden sich Microservices aktuell in Evaluierung, keine Erfahrung mit Microservices haben 31,8% beziehungsweise 21 Personen, lediglich einmal wurde die Auswahl „kein Interesse an Microservices“ getroffen.

Die erwarteten Vorteile durch den Einsatz von Microservices sind in Tabelle 4-2 ersichtlich. Fünf Personen nützten die Möglichkeit, textuell weitere Vorteile zu beschreiben. Dabei wurde dreimal die Möglichkeit erwähnt, dass Services gesondert skaliert werden können, einmal auf die freie Technologiewahl hingewiesen und einmal wurde angemerkt, dass Microservices aktuell nicht den Erwartungen in der Praxis gerecht werden, da sie die Komplexität erhöhen statt diese zu reduzieren. Diese Antworten werden in der nachfolgenden Tabelle nicht gesondert angeführt.

Vorteile	Anzahl	Prozent
Aufteilung der Komplexität eines Systems	47	71,2%
Unabhängiges Deployment / Betrieb	43	65,2%
Freie Technologiewahl	32	48,5%
Selbstständige, voneinander unabhängige Teams	28	42,4%
Verbesserung der Reaktionszeit auf geänderte Marktbedingungen	17	25,8%
Reduktion der Betriebs-/Wartungskosten	14	21,2%
Entwicklung neuer Geschäftsmodelle	11	16,7%
Erschließung neuer Kundenkreise	8	12,1%
keine Erwartung von Vorteilen	8	12,1%

Tabelle 4-2: Erwartete Vorteile durch den Einsatz von Microservices

Erwartete Nachteile durch Microservices sind vor allem die erhöhte Komplexität und die fehleranfällige Kommunikation zwischen den Services, beide Ausprägungen wurden jeweils von etwa der Hälfte der teilnehmenden Personen als Nachteil markiert. Die gesammelten Ergebnisse finden sich in Tabelle 4-3. Drei der Teilnehmenden haben als weiteren Nachteil angegeben, dass Microservices negative Auswirkungen auf den initialen Aufwand in der Entwicklung haben, da unter anderem zusätzliche Schnittstellen definiert und neue Technologien gelernt werden müssen. Diese Angaben werden in der Ergebnistabelle unter „Erhöhung des initialen Entwicklungsaufwands“ geführt. Weiters wurde einmal angemerkt, dass Microservices aktuell nicht den Erwartungen in der Praxis gerecht werden, da sie die Komplexität erhöhen statt diese zu reduzieren.

Nachteile	Anzahl	Prozent
Erhöhung der Komplexität des Gesamtsystems	33	50,0%
Kommunikation zwischen Services	32	48,5%
Dezentrales Datenmanagement	26	39,4%

Organisatorische Änderungen	26	39,4%
Erhöhung des betrieblichen Aufwands	26	39,4%
Freie Technologiewahl	8	12,1%
keine Erwartung von Nachteilen	8	12,1%
Erhöhung des initialen Entwicklungsaufwands	3	4,6%

Tabelle 4-3: Erwartete Nachteile durch den Einsatz von Microservices

Aus der Inhaltsanalyse der offenen Frage #12 hinsichtlich der Vorteile, die sich die Befragten aus Sicht der Kunden erwarten, kann vor allem die erhöhte Reaktionszeit und die dadurch schnellere Time-to-Market abgeleitet werden, neun Mal wurde dieses Merkmal genannt. Weiters wurden die höhere Verfügbarkeit und bessere Wartbarkeit als Vorteile genannt, von denen Kunden bei der Einführung von Microservices profitieren können. Auch die unabhängige Entwicklung kann Vorteile bringen, da durch die strikte Trennung von Teams weniger Abhängigkeiten bestehen und dadurch der Abstimmungsaufwand reduziert werden kann. Dies könnte vor allem Vorteile haben, wenn interne und externe Entwicklungsteams eingesetzt werden. Tabelle 4-4 beinhaltet die aufbereiteten Ergebnisse dieser Frage.

Erwartungen aus Kundensicht	**Anzahl**	**Prozent**
Kürzere Time-to-Market ▪ schnellere Reaktionszeiten auf geänderte Anforderungen ▪ schnellerer Markeintritt möglich	9	47,4%
Erhöhte Wartbarkeit ▪ Systeme können einfacher und schneller angepasst oder erweitert werden ▪ Austausch einzelner Anwendungsbereiche möglich	5	26,3%
Unabhängige Entwicklung ▪ Weiterentwicklung einzelner fachlicher Module ▪ Entkopplung von Systemen ▪ Reduzierter Kommunikationsaufwand beim Einsatz von internen und externen Entwicklungsteams	4	21,1%
Unabhängiger Betrieb ▪ Höhere Verfügbarkeit (Ausfallsicherheit, getrennte Skalierbarkeit) ▪ Updates haben weniger Einfluss auf laufenden Betrieb	4	21,1%
Schrittweise Ablöse ▪ schrittweise Ablöse von Altsystemen möglich	1	5,3%
Erschließung neuer Geschäftsfelder ▪ Flexible Kombination von Diensten ermöglicht neue Anwendungen zu erstellen	1	5,3%

Tabelle 4-4: Vorteile, die aus Kundensicht für die Einführung von Microservices sprechen könnten

Betrieb / Infrastruktur

Beinahe bei allen an der Umfrage teilnehmenden Personen werden Werkzeuge eingesetzt, um ein (teil-) automatisiertes Bereitstellen der Software zu ermöglichen, lediglich vier Personen gaben an, dass keine Tools im Einsatz sind. Abbildung 4-6 zeigt die Ergebnisse dieser Frage.

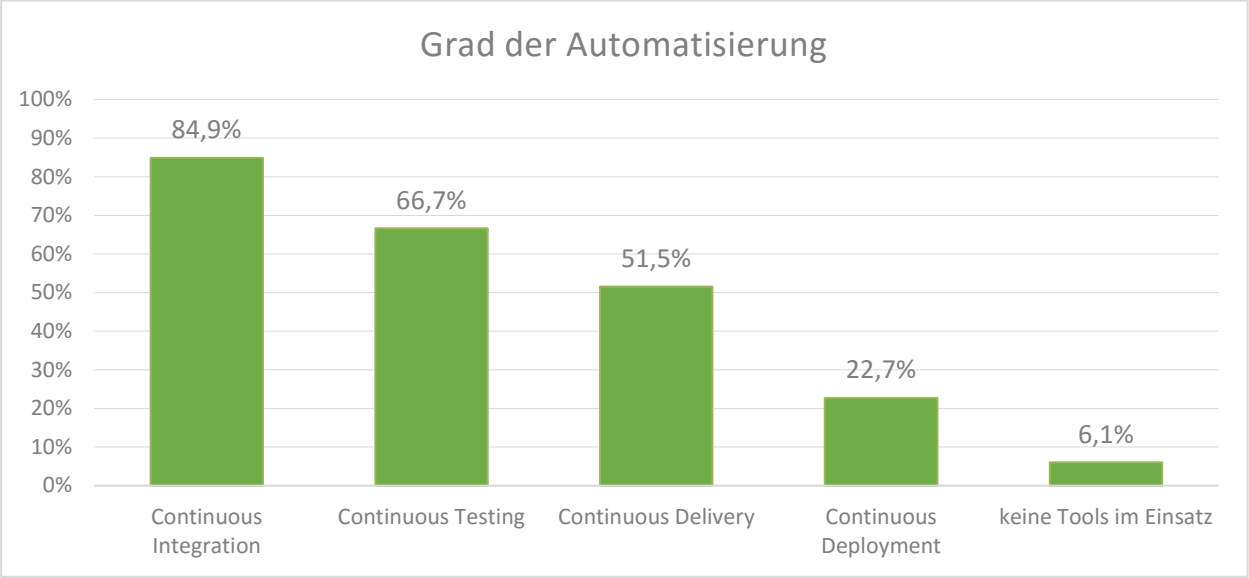

Abbildung 4-6: Einsatz von Werkzeugen, um Software (teil-) automatisiert bereitzustellen

Die Frage nach dem Einsatz von Containertechnologien im Arbeitsumfeld zeigte, dass diese bei der Hälfte der Befragten nicht im Einsatz sind (42,4%) oder dies nicht bekannt ist (7,6%). 20 Personen (30,3%) gaben an, Containertechnologien zu evaluieren und lediglich 19,7% der Befragten setzen diese in der Entwicklung und im Betrieb ein.

Ähnlich verhält es sich bei der Verwendung von cloudbasierten Diensten, ebenfalls die Hälfte gab an, dass keine cloudbasierten Dienste im Einsatz sind (40,9%) oder sie nicht wissen, ob diese eingesetzt werden (9,1%). Private Cloud-Dienste werden von 33,3% und öffentliche von 22,7% der Teilnehmenden eingesetzt. 18,2% verwenden dabei eine Platform-as-a-Service (PaaS) Lösung, 15,2% greifen auf eine Infrastructure-as-a-Service (IaaS) Lösung zurück.

Organisation

Aus den Umfrageergebnissen geht hervor, dass bei der Hälfte der teilnehmenden Personen im Arbeitsumfeld eine Trennung zwischen Entwicklung und Betrieb erfolgt und bei 59,1% auf DevOps-Ansätze zurückgegriffen wird. Die Bereitschaft, für die Einführung neuer technologischer Ansätze organisatorische Änderungen durchzuführen, ist bei 41 Personen (62,1%) gegeben. Das genaue Ergebnis mit einer prozentuellen Verteilung wird Abbildung 4-7 gezeigt.

Noch mehr Zustimmung unter den Befragten herrscht bei der Bereitschaft, die vollständige Verantwortung für Entwicklung, Qualitätssicherung und Betrieb an ein Team zu übertragen. Dieses Ergebnis findet sich in Abbildung 4-8.

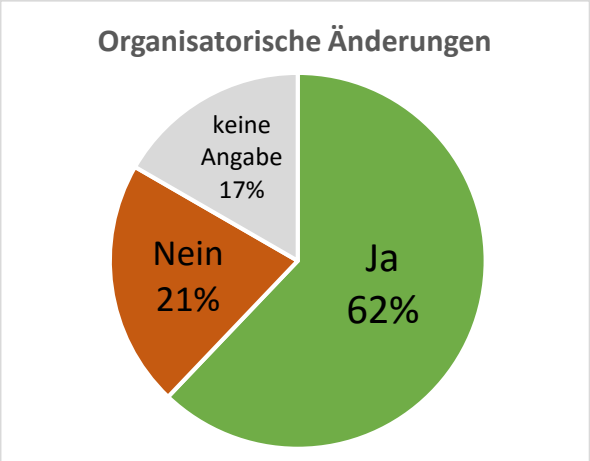

Abbildung 4-7: Bereitschaft, organisatorische Änderungen durchzuführen

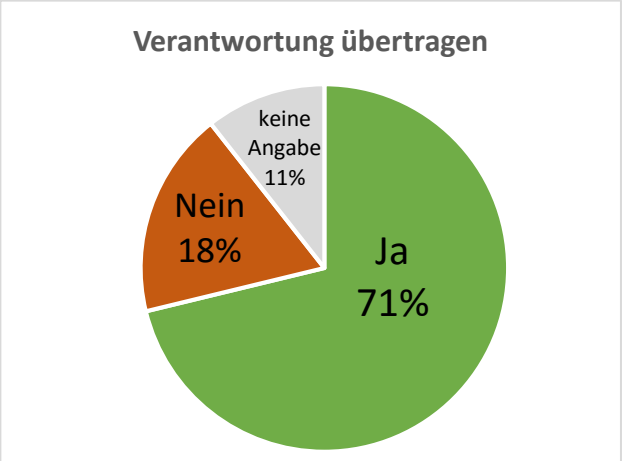

Abbildung 4-8: Bereitschaft, Verantwortung zu übertragen

Legacy-Anwendungen

In der folgenden Tabelle 4-5 wird das Abstimmungsergebnis zur Frage nach Gründen, welche zu einer Ablöse oder Modernisierung von monolithischen Anwendungen führen, präsentiert. Am häufigsten wurde dabei die eingeschränkte Wartbarkeit als Motivation genannt.

Gründe für Ablöse/Modernisierung	Anzahl	Prozent
Eingeschränkte Wartbarkeit	46	69,7%
Veraltete Technologien	44	66,8%
Hohe Wartungs-/Betriebskosten	40	60,6%
Hohe Komplexität des Systems	36	54,6%
Neue Funktionalitäten	26	39,4%

Tabelle 4-5: Gründe für die Ablöse oder Modernisierung von monolithischen Applikationen

42 Personen gaben des Weiteren an, aktuell eine Modernisierung eines Altsystems zu planen. Dafür wollen 11 Personen (26,2%) Microservices einsetzen, 19 (45,2%) evaluieren deren Einsatz und 12 (28,6%) gaben an, dass bei der Modernisierung Microservices nicht berücksichtigt werden. Somit ergibt sich, dass 71,4% dieser Personen das Microservices-Architekturparadigma für eine Modernisierung in Betracht ziehen. In Abbildung 4-9 wird diese Verteilung grafisch dargestellt.

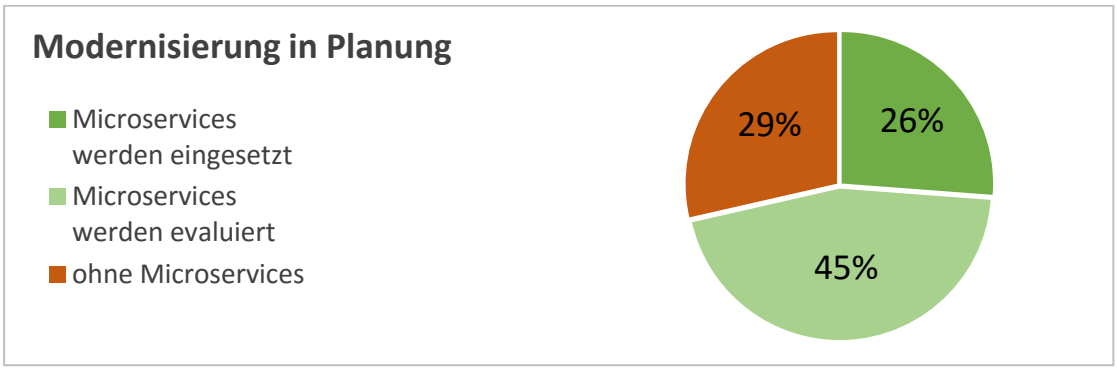

Abbildung 4-9: Einsatz von Microservices bei der Planung von Modernisierungsmaßnahmen von Altsystemen

Optionale offene Fragen

Die Frage #22 „Welche Voraussetzungen muss eine monolithische Anwendung aus Ihrer Sicht erfüllen, um für eine Erweiterung mittels Microservices geeignet zu sein?“ wurde von 25 Personen beantwortet. Die Inhaltsanalyse der Ergebnisse zeigte, dass für diese Befragten die Teilbarkeit eines Systems ausschlaggebend für eine Erweiterung mittels Microservices ist, weiteres findet sich in Tabelle 4-6.

Voraussetzungen für Modernisierung mittels Microservices	Anzahl	Prozent
Teilbarkeit des Systems ▪ System kann nach fachlichen und technischen Aspekten geteilt werden ▪ Modularer Aufbau ist zumindest ansatzweise vorhanden ▪ Verteilte Verarbeitung muss möglich sein	16	64,0%
Organisatorische Aspekte ▪ Organisation muss Vertrauen in IT haben ▪ Bereitschaft, organisatorische Änderungen durchzuführen, ist gegeben ▪ Personal muss hohe technologische und fachliche Kompetenz besitzen ▪ benötigte Ressourcen (Budget, Personal) müssen verfügbar sein	4	16,0%
Komplexität des Systems ▪ Komplexität muss hoch genug sein, damit Aufteilung Sinn macht ▪ Größe der Anwendung ausreichend	2	8,0%
Datenmanagement ▪ Trennung des bestehenden Datenmodells muss möglich sein ▪ Verwendung von fachlichen Schlüsseln anstatt Fremdschlüssel-Beziehungen ist möglich	2	8,0%
Testabdeckung ▪ Funktionalitäten des monolithischen Systems weisen eine ausreichende Testabdeckung auf	1	4,0%

Tabelle 4-6: Voraussetzungen, die eine monolithische Anwendung für die Erweiterung mit Microservices erfüllen soll

Gründe, die gegen den Einsatz von Microservices sprechen, um ein Altsystem zu modernisieren, wurden von 26 Teilnehmenden genannt und werden in nachfolgender Tabelle 4-7 aufbereitet und gesammelt gelistet.

Gründe gegen den Einsatz von Microservices	Anzahl	Prozent
Kosten übersteigen Nutzen ▪ Kosten für Einführung von Microservices sind höher als der zu erwartende Nutzen ▪ Altsystem verursacht nur geringe Wartungskosten	10	38,5%
Erhöhte Komplexität ▪ Erhöhung der technischen Komplexität des Gesamtsystems ▪ Erhöhung der organisatorischen Komplexität	7	26,9%
Monolithisches System ist zu klein ▪ bestehendes System ist zu klein oder weist keine ausreichende fachliche Komplexität auf, um es sinnvoll zu teilen	5	19,2%
Kein Wachstumspotential gegeben ▪ System wird nicht weiterentwickelt ▪ keine oder wenige neue Funktionen werden benötigt	5	19,2%

Auswirkungen auf Betrieb ▪ betriebliche Aufwände steigen ▪ unzureichende Ressourcen ▪ kundenseitige Bereitschaft nicht gegeben	4	15,4%
Unzureichende Kompetenzen des Personals ▪ Personal mit ausreichender Erfahrung fehlt	2	7,7%

Tabelle 4-7: Gründe, die gegen den Einsatz von Microservices sprechen, um ein Altsystem zu modernisieren

Weiters nannten 12 Personen bei Frage #24.1 Strategien für die Ablöse oder Erweiterung von monolithischen Systemen. Unter anderem wurde dabei sechs Mal die in Abschnitt 3.2.6 diskutierte *Strangler Application* Strategie und drei Mal die in Abschnitt 3.2.7 vorgestellte *Extract Services* Vorgehensweise beschrieben.

Abschließend wurden bei der Beantwortung von Frage #27 verschiedene No-Gos von den teilnehmenden Personen beschrieben. Im Zuge der inhaltlichen Analyse der Antworten wurden die genannten No-Gos wie folgt zusammengefasst:

- Der Betrieb kann das System aufgrund erhöhter Komplexität nicht mehr handhaben.
- Microservices sind für die meisten Unternehmen unbrauchbar, da diese organisatorisch und im Betrieb zu schwer zu handhaben sind.
- Einführung von Microservices ohne vorherige Auftrennung der bestehenden Datenbasis.
- Fehlende Skills und Lernbereitschaft im Entwicklungsteam, um den Technologie-Stack zu beherrschen.

4.2.4 Analyse und Interpretation

Die zuvor beschriebenen Ergebnisse werden in diesem Abschnitt näher betrachtet, analysiert und interpretiert.

Aus der Befragung geht hervor, dass veraltete Technologien, eine hohe Komplexität und eingeschränkte Wartbarkeit – welche von den Teilnehmenden auch als wichtigstes Qualitätsmerkmal einer Software deklariert wurde – zu den am häufigsten auftretenden Problemen zählen, mit denen sich Personen bei monolithischen Systemen auseinandersetzen müssen und diese Faktoren in der Regel auch die Hauptgründe sind, welche Modernisierungsmaßnahmen eines Systems auslösen.

Dies spiegelt sich auch bei der Erwartungshaltung an Microservices wider, der Großteil der Befragten erwartet sich durch den Einsatz des Microservices-Architekturparadigmas Vorteile hinsichtlich der Aufteilung der Komplexität eines Systems, der freien Technologiewahl und der Unabhängigkeit in Entwicklung und Betrieb. Das sind alles Faktoren, die nach Wolff (2015) und Newman (2015) zu einer Verbesserung der Wartbarkeit führen können.

Die hohen Erwartungen an Microservices hinsichtlich einer Reduzierung der Komplexität werden jedoch auch kritisch betrachtet. Die Hälfte der Teilnehmenden sehen die Erhöhung der Komplexität des Gesamtsystems als einen der größten Nachteile von Microservices-Architekturen. Weiters wurde darauf hingewiesen, dass neben der Erhöhung der technischen

Komplexität des Gesamtsystems, auch eine Erhöhung der organisatorischen Komplexität zu berücksichtigen sei, die durch eine Vielzahl unabhängiger Teams entstehen würde. Die Komplexität ist nach Meinung der Teilnehmenden auch ein ausschlaggebender Faktor für den Einsatz von Microservices bei der Modernisierung eines Systems, diese sollte sowohl fachlich als auch technisch hoch genug sein, damit eine Teilung des Systems sinnvoll ist.

Des Weiteren ist in den Antworten ersichtlich, dass hinsichtlich der Wartungs- und Betriebskosten eine geteilte Meinung unter den Befragten herrscht. 14 Personen haben angegeben, sich durch Microservices eine Reduktion dieser Kosten zu erwarten, demgegenüber rechnen 26 Personen mit einer Erhöhung des betrieblichen Aufwands. Über 60% haben jedoch auch geantwortet, dass hohe Wartungs- und Betriebskosten für gewöhnlich einer der Hauptgründe für die Ablöse oder Modernisierung eines Legacy-Systems sind, wobei hinsichtlich dem Einsatz von Microservices eine Kosten-Nutzen-Analyse erforderlich sei, welche unter anderem die zu erwartende Erhöhung der betrieblichen Aufwände beachten soll.

Bezüglich organisatorischer Aspekte kann geschlussfolgert werden, dass unter den Teilnehmenden eine grundsätzliche Bereitschaft zur Durchführung von organisatorischen Änderungen und Übertragung der Verantwortung an einzelne Teams besteht. Hinsichtlich der Einführung von Microservices wurde hervorgehoben, dass dafür ein entsprechendes Vertrauen der Organisation in die IT existieren sollte, benötigte Ressourcen zur Verfügung gestellt werden müssen und das eingesetzte Personal eine entsprechende Erfahrung benötigt sowie über hohe technologische als fachliche Kompetenzen verfügen sollte.

Fehlende Erfahrung des Personals ist auch einer der Gründe, die nach Ansicht der Befragten gegen den Einsatz von Microservices sprechen. Eine Betrachtung des angegebenen Erfahrungslevels in der Softwareentwicklung kombiniert mit der Erfahrung bei für Microservices wichtigen Ansätzen zeigte, dass der Großteil der befragten Personen mit zumindest drei Jahren Erfahrung in der Softwareentwicklung keine oder lediglich theoretische Kenntnisse dieser haben. Das Ergebnis dieser Auswertung wird in Tabelle 4-8 präsentiert und lässt darauf schließen, das zum Teil das notwendige Wissen und die Erfahrung fehlen und somit eine hohe Lernbereitschaft seitens des Personals erforderlich ist.

		3 bis 10 Jahre		**mehr als 10 Jahre**	
Erfahrung mit …	**Erfahrungslevel**	**Anzahl**	**%**	**Anzahl**	**%**
Resilience / Design for Failure	praktisch	1	3,7%	7	20,6%
	theoretisch	15	55,6%	19	55,9%
	keine	11	40,7%	8	23,5%
Eventual Consistency	praktisch	4	14,8%	7	20,6%
	theoretisch	9	33,3%	16	47,1%
	keine	14	51,9%	11	32,4%
Ereignisgesteuerte Architekturen	praktisch	7	25,9%	14	41,2%
	theoretisch	14	51,9%	18	52,9%
	keine	6	22,2%	2	5,9%

Tabelle 4-8: Erfahrung mit verschiedenen Methoden und Ansätzen im Microservices Umfeld

Auffallend an den Ergebnissen ist, dass lediglich ein geringer Teil der Befragten die Verbesserung der Reaktionszeit auf geänderte Marktbedingungen, die Erschließung neuer Kundenkreise oder die Entwicklung neuer Geschäftsmodelle als Vorteile durch den Einsatz von Microservices sieht. Auch eine Einschränkung der Ergebnisse auf die 13 Personen, die sich der Berufsgruppe „IT-Management“ zugeordnet haben, zeigte keine wesentlichen Veränderungen zu den im Gesamten ermittelten Werten. Der verhältnismäßig niedrige Wert bezüglich der verbesserten Reaktionszeit könnte jedoch auch auf eine unzureichende Fragestellung hinweisen, da, wie in Tabelle 4-4 ersichtlich, eine verkürzte Time-to-Market bei den Vorteilen aus Kundensicht am häufigsten genannt wurde.

In weiterer Folge lässt sich aus den übermittelten Antworten ableiten, dass sich die Mehrheit der Befragten Vorteile hinsichtlich der unabhängigen Bereitstellung und des Betriebs einzelner Services erwartet, um dadurch Dienste unabhängig voneinander skalieren zu können. Aus der Beantwortung der Fragen zu den Themen Infrastruktur und Betrieb lässt sich jedoch ableiten, dass die, für die Entwicklung und dem Betrieb einer Vielzahl an Services notwendige Infrastruktur sowie der ebenfalls dafür benötigte Automatisierungsgrad zum Teil nicht ausreichend vorhanden ist. Nur etwas mehr als die Hälfte der Befragten hat angegeben, eine *Continuous-Delivery-Pipeline* einzusetzen und knapp 20%, dass für die Bereitstellung der notwendigen Infrastruktur moderne Containertechnologien wie beispielsweise *Docker* oder *Kubernetes* verwendet werden.

Bei der abschließenden Ergebnisanalyse in Bezug auf Unterschiede zwischen den an der Umfrage teilnehmenden Zielgruppen wurden nur wenige nennenswerte Abweichungen zu den im vorherigen Abschnitt beschriebenen Ergebnissen erkannt. Vor allem Personen, die der Berufsgruppe „Software-Architektur“ zugeordnet sind, haben die Kommunikation zwischen Services zu 60% und die Einführung eines dezentralen Datenmanagements zu 55% als Nachteil genannt. Die dabei ermittelten Durchschnittswerte liegen bei 49% beziehungsweise 39%. Weiters ist bei dieser Zielgruppe hervorzuheben, dass lediglich 5% im Vergleich zu durchschnittlich 29% der Befragten angegeben haben, bei einer Modernisierung eines monolithischen Systems das Microservices-Architekturparadigma nicht berücksichtigen zu wollen.

4.3 Prüfung der Hypothesen

In diesem Abschnitt werden die in Kapitel 4.1 aufgestellten Hypothesen geprüft und gegebenenfalls verifiziert oder falsifiziert. Dabei wird auf die Ergebnisse der durchgeführten Befragung zurückgegriffen.

4.3.1 Hypothese 1

Von den 66 an der Umfrage teilnehmenden Personen gaben 43 (65,2%) an, oft mit komplexen Altsystemen zu tun zu haben, davon planen 26 eine Ablöse oder Modernisierung eines dieser Systeme. Sieben dieser Personen wollen dabei Microservices einsetzen, weitere 13 Personen evaluieren deren Einsatz. Demgegenüber gaben sechs Befragte an, Microservices für die Modernisierung eines Altsystems nicht einsetzen zu wollen. Aus diesen Ergebnissen lässt sich

ableiten, dass 76,9% der Personen, welche häufig Probleme mit komplexen Altsystemen haben, Microservices bei der Planung einer Modernisierung berücksichtigen und somit ein Wert über der durchschnittlichen Verteilung von 71,4% erreicht wurde.

In weiterer Folge hat die Auswertung gezeigt, dass von den 19 Personen (28,8%), welche gelegentlich mit einer erhöhten Komplexität von monolithischen Systemen konfrontiert sind, zumindest 14 eines dieser Altsysteme modernisieren wollen. Von diesen 14 Befragten wollen vier Microservices einsetzen und sechs befinden sich in der Phase der Evaluierung.

Nur lediglich vier Personen haben angegeben, sich selten mit komplexen Legacy-Systemen beschäftigen zu müssen. Von diesen Befragten planen zwei eine Modernisierung eines Systems, der Einsatz von Microservices ist bei diesen Fällen aber nicht vorgesehen.

Die soeben beschriebenen Ergebnisse werden in folgender Tabelle 4-9 gesammelt aufgelistet.

		Modernisierung Altsystem				
Probleme mit Altsystemen aufgrund hoher Komplexität		**in Planung**	**Microservices**			**ohne Microservices**
			werden eingesetzt	werden evaluiert	**mit Microservices**	
oft	43	26	7	13	20 (76,9%)	6 (23,1%)
gelegentlich	19	14	4	6	10 (71,4%)	4 (28,6%)
selten	4	2	0	0	0 (0,0%)	2 (100,0%)
Gesamt	**66**	**42**	11	19	**30 (71,4%)**	**12 (28,6%)**

Tabelle 4-9: Probleme mit Altsystemen aufgrund hoher Komplexität

Weiters haben 36 Befragte (54,6%) angegeben, dass eine hohe Komplexität von Altsystemen einer der Hauptgründe für eine Modernisierung ist. Von diesen planen 20 Personen aktuell eine solche Modernisierung, wobei vier Personen (20,0%) Microservices einsetzen wollen und diese von weiteren 12 Personen (60,0%) in Evaluierung sind. Somit ergibt sich, dass 80,0% der Befragten, für welche die Komplexität eines monolithischen Altsystems einer der Hauptgründe für eine Modernisierung ist und eine solche gerade planen, Microservices in Betracht ziehen und lediglich 20,0% gegen deren Einsatz sind. Die aus dieser Analyse hervorgegangen Daten werden in Tabelle 4-10 präsentiert.

		Modernisierung Altsystem				
Komplexität als Grund für Modernisierung		**in Planung**	**Microservices**			**ohne Microservices**
			werden eingesetzt	werden evaluiert	**mit Microservices**	
Hohe Komplexität	36	20	4	12	16 (80,0%)	4 (20,0%)

Tabelle 4-10: Komplexität als Grund für eine Modernisierung

Aus diesen Ergebnissen kann abgeleitet werden, dass der Großteil der Befragten, welche oft oder gelegentlich mit einer hohen Komplexität von Altsystemen in Berührung kommen und eine Modernisierung dieser planen, entweder Microservices einsetzen wollen oder deren Verwendung zumindest in Betracht ziehen. Daraus lässt sich schließen, dass die vorliegende Nullhypothese

„*Die Komplexität eines monolithischen Altsystems hat keinen Einfluss auf die Bereitschaft, bei einer Modernisierung des Systems Microservices zu verwenden.“* falsifiziert und die Alternativhypothese als richtig angenommen werden kann, da die Bereitschaft zur Verwendung von Microservices von der Komplexität des Altsystems beeinflusst wird.

4.3.2 Hypothese 2

Ähnlich wie bei der Komplexität verhält es sich auch mit der Wartbarkeit von Altsystemen und den dadurch entstehenden Kosten. 46 der teilnehmenden Personen haben angegeben, dass eine eingeschränkte Wartbarkeit einer der Hauptgründe für Modernisierungsmaßnahmen ist, bei 40 Personen finden sich in den Ergebnissen hohe Wartungskosten von Altsystemen als Beweggrund. Tabelle 4-11 zeigt diese Ergebnisse im Detail.

		Modernisierung Altsystem				
Gründe für Modernisierung		**in Planung**	**Microservices**			**ohne Microservices**
			werden eingesetzt	werden evaluiert	**mit Microservices**	
Eingeschränkte Wartbarkeit	46	29	9	13	22 (75,9%)	7 (24,1%)
Hohe Wartungskosten	40	27	6	14	20 (74,1%)	7 (25,9%)

Tabelle 4-11: Wartbarkeit und Kosten als Gründe für eine Modernisierung

Auch bei der Betrachtung der Wartbarkeit und den Wartungskosten zeigt sich, dass die Bereitschaft für die Berücksichtigung von Microservices für eine Modernisierung eines Altsystems höher ist, als der ermittelte Durchschnittswert von 71,4%. Somit kann aufgrund dieser Ergebnisse die H0 ebenfalls falsifiziert und die Alternativhypothese „*Je mehr die Wartbarkeit eines monolithischen Altsystems eingeschränkt ist und je höhere Kosten dabei verursacht werden, desto höher ist die Bereitschaft, bei einer Modernisierung das Microservices-Architekturparadigma zu verwenden“* verifiziert und als gültig angenommen werden.

4.3.3 Hypothese 3

Bei der Beantwortung der Umfrage haben beinahe alle Personen (98,5%) angegeben, theoretische oder praktische Erfahrungen mit verteilten Systemen aufzuweisen. Weiters haben 81,8% Kenntnisse des Microservices-Architekturparadigmas und 45,5% der Befragten gaben an, praktische Erfahrungen mit Microservices zu haben.

Von den 66 an der Umfrage beteiligten Personen will mehr als die Hälfte (57,6%) Microservices bei zukünftigen Projekten einsetzen. Wie in Tabelle 4-12 ersichtlich, erhöht sich dieser Wert, je mehr Erfahrung eine Person mit verteilten Systemen beziehungsweise Microservices hat und erreicht 80,0% bei Befragten, die mit Microservices bereits praktische Erfahrungen haben und diese entweder produktiv einsetzen oder gerade entwickeln.

	Anzahl	%	Anzahl	%
Erfahrungslevel der Befragten	**Ja**		**Nein**	
Erfahrung mit verteilten Systemen	65	98,5%	1	1,5%
Kenntnisse von Microservices	54	81,8%	12	18,2%
Praktische Erfahrungen mit Microservices	30	45,5%	36	54,5%
Einsatz von Microservices bei zukünftigen Projekten	**Ja**		**Nein**	
Alle Befragten	38	57,6%	28	42,4%
Befragte mit Erfahrung mit verteilten Systemen	38	58,5%	27	41,5%
Befragte mit Kenntnissen von Microservices	36	66,7%	18	33,3%
Befragte mit praktischer Erfahrung mit Microservices	24	80,0%	6	20,0%

Tabelle 4-12: Erfahrung mit verteilten Systemen und Bereitschaft, Microservices bei zukünftigen Projekten einzusetzen

Somit kann auf Basis der vorliegenden Erkenntnisse die Nullhypothese „*Unabhängig von der Erfahrung einer Person mit verteilten Systemen, wird der Einsatz von Microservices bei zukünftigen Projekten beabsichtigt.“* falsifiziert werden, da davon auszugehen ist, dass die Entscheidung für den Einsatz von Microservices davon abhängt, ob eine Person Erfahrung mit verteilten Systemen aufweist.

4.3.4 Hypothese 4

Genau die Hälfte der teilnehmenden Personen hat angegeben, dass in ihrem Arbeitsumfeld eine organisatorische Trennung zwischen Entwicklung und Betrieb erfolgt. Daraus lässt sich schließen, das unabhängige Teams in diesem Umfeld eher nicht zusammengestellt werden können und dafür Änderungen an der organisatorischen Struktur notwendig sein könnten.

Von diesen 33 Personen haben 23 (69,7%) oft oder gelegentlich Probleme mit einer langen Dauer für die Bereitstellung von Software und ebenso viele sind dazu bereit, organisatorische Änderungen für den Einsatz neuer Technologien durchzuführen. Weiters haben 24 (72,7%) dieser Befragten geantwortet, dass sie die Verantwortung an unabhängige Teams übertragen wollen, 18 (54,5%) wollen Microservices bei zukünftigen Projekten einsetzen und 10 (30,3%) erwarten sich durch Microservices Vorteile hinsichtlich einer Verbesserung der Reaktionszeit.

Durch eine weitere Filterung der Ergebnisse konnten neun Personen ermittelt werden, welche angegeben haben, sowohl organisatorische Entscheidungen in ihrem Arbeitsumfeld zu treffen als auch Probleme mit langen Deployment-Zyklen zu haben. Von diesen Personen sind sieben (77,8%) dazu bereit, organisatorische Änderungen durchzuführen, sechs (66,7%) wollen Microservices einsetzen und fünf (55,6%) haben sich dazu bereit erklärt, die Verantwortung für Entwicklung, Betrieb und Qualitätssicherung an eigenständige Teams zu übertragen. Jedoch haben lediglich zwei (22,2%) dieser Personen die verbesserte Reaktionszeit als einen der wesentlichen Vorteile von Microservices ausgewählt.

Die soeben diskutierten Ergebnisse lassen zwar Tendenzen hinsichtlich der Bereitschaft, organisatorische Änderungen durchführen zu wollen, erkennen, es lässt sich aber nicht eindeutig feststellen, ob die zugrundeliegende Motivation die Verbesserung der Reaktionszeit ist. Aus dem

Fragebogen geht daher nicht eindeutig hervor, ob die Hypothese „*Je länger die Bereitstellung eines monolithischen Systems dauert, desto eher sind Unternehmen dazu bereit, die für den Einsatz von Microservices notwendigen organisatorischen Änderungen durchzuführen, um eine Verbesserung der Reaktionszeit auf geänderte Marktanforderungen herbeizuführen.*“ zutrifft. Daher lässt sich aufgrund der vorliegenden Ergebnisse diese Hypothese nicht bestätigen.

4.3.5 Hypothese 5

Wie in Abbildung 4-2 ersichtlich, haben im Zuge der Befragung weniger als die Hälfte der teilnehmenden Personen (48,5%) angegeben, oft oder zumindest gelegentlich mit einer eingeschränkten Verfügbarkeit von monolithischen Systemen konfrontiert zu sein. Die Frage nach der Bereitschaft, dezentrales Datenmanagement einzuführen und Daten dadurch möglicherweise redundant speichern zu müssen, wurde von 40 Personen mit ja oder eher ja beantwortet. Demgegenüber sind nur 30 der Personen dazu bereit, Abstriche bei der Konsistenz hinzunehmen, um dadurch die Verfügbarkeit zu erhöhen.

Aus der detaillierten Analyse dieser Ergebnisse geht hervor, dass die Bereitschaft der Befragten zunimmt, je häufiger sie Probleme mit einer eingeschränkten Verfügbarkeit haben. Dies zeigt sich in Tabelle 4-13, welche die eingeschränkte Verfügbarkeit der Bereitschaft, dezentrales Datenmanagement einzuführen, gegenüberstellt.

	Bereitschaft, dezentrales Datenmanagement einzuführen					
Eingeschränkte Verfügbarkeit	**ja**	**eher ja**	**%**	**eher nein**	**nein**	**%**
oft	1	2	75,0%	1	0	25,0%
gelegentlich	11	7	72,0%	6	1	28,0%
selten	7	11	64,3%	7	3	35,7%
nie	0	1	50,0%	0	1	50,0%
Gesamt	19	21	67,8%	14	5	32,2%

Tabelle 4-13: Bereitschaft, dezentrales Datenmanagement einzuführen, in Bezug auf eine eingeschränkte Verfügbarkeit

Diese Tendenz lässt sich in etwas abgeschwächter Form auch in Tabelle 4-14 erkennen, welche die Ergebnisse der Prüfung beinhaltet, ob Personen dazu bereit sind, Abstriche bei der Konsistenz von Daten hinzunehmen, um dadurch Vorteile in Hinblick auf die Verfügbarkeit zu erhalten.

	Abstriche bei Konsistenz, um Verfügbarkeit zu erhöhen					
Eingeschränkte Verfügbarkeit	**ja**	**eher ja**	**%**	**eher nein**	**nein**	**%**
oft	0	3	75,0%	0	1	25,0%
gelegentlich	7	8	57,7%	4	7	42,3%
selten	3	9	40,0%	11	7	60,0%
nie	0	0	0,0%	1	1	100,0%
Gesamt	10	20	48,4%	16	16	51,6%

Tabelle 4-14: Bereitschaft, Abstriche bei der Konsistenz zu machen, um die Verfügbarkeit eines Systems zu erhöhen

Die Gegenüberstellung dieser Ergebnisse zeigte, dass zwischen der Verfügbarkeit und der Bereitschaft, dezentrales Datenmanagement einzuführen, ein Zusammenhang besteht. Je häufiger Personen mit einer eingeschränkten Verfügbarkeit zu tun haben, desto eher sind sie dazu bereit, die zuvor genannten Kompromisse einzugehen. Daraus lässt sich schlussfolgern, dass die Nullhypothese zu falsifizieren ist und somit die Alternativhypothese „*Je öfter die Verfügbarkeit eines monolithischen Altsystems eingeschränkt ist, desto eher ist die Bereitschaft zur Einführung eines dezentralen Datenmanagements gegeben.“* bestätigt werden kann.

4.4 Marktbetrachtung

Die abschließende Ausarbeitung des empirischen Teils beschreibt die durchgeführte Analyse publizierter Best-Practice-Ansätze namhafter Unternehmen. Zu Beginn werden ausgehend von der theoretischen Ausarbeitung und der zuvor durchgeführten Befragung Fragestellungen definiert, welche im Zuge der Marktbetrachtung beantwortet werden, um somit die bisher ermittelten Ergebnisse zu evaluieren.

Nachfolgend werden diese Fragestellungen aufgelistet:

- Welche Ansätze existieren, um die steigende Komplexität des Gesamtsystems beherrschbar zu machen?
- Wie gehen Unternehmen mit der steigenden organisatorischen Komplexität um, die durch eine Vielzahl an unabhängigen Teams entsteht?
- Wie können die durch den Einsatz von Microservices entstehenden betrieblichen Aufwände reduziert werden?
- Welche Arten von Tests können bei einer Modernisierung mittels Microservices durchgeführt werden, um die korrekte Funktionsweise bestehender Programmteile sicherzustellen?
- Gibt es sinnvolle Ansätze, Microservices ohne dezentralem Datenmanagement einzuführen, um Abstriche bei der Konsistenz vermeiden zu können?
- Wie lassen sich Microservices-Architekturen auf nicht-internetbasierte Geschäftsmodelle anwenden?

Als Ausgangspunkt für diese Analyse wurde die von Richardson (2016) erstellte Auflistung gewählt, die Unternehmen beinhaltet, welche Microservices-Architekturen eingeführt und ihre Erfahrungen publiziert haben.

4.4.1 Amazon

Amazon gilt als eines der ersten Unternehmen, welches auf Basis des CAP-Theorems ihren monolithischen Webshop in mehrere Services aufteilte, um dadurch besser skalieren und die Verfügbarkeit ihrer Systeme an die gesteigerten Benutzerzahlen anpassen zu können (Wolff, 2006). Neben der Erhöhung der Verfügbarkeit waren lange Deployment-Zyklen ein Faktor, der

Amazon dazu bewegte, ihre monolithische Anwendung in einzelne Dienste aufzuteilen, dabei den Grad der Automatisierung zu erhöhen und somit die Dauer für die Bereitstellung einer neuen Funktionalität zu verkürzen (Fulton III, 2015).

Um ihre auf einer Zwei-Schichten-Architektur basierende monolithische Webapplikation abzulösen, wurden funktionale Einheiten schrittweise aus dem System losgelöst, in eigene Services ausgelagert und mittels einer Webservice-Schnittstelle für andere Einheiten zur Verfügung gestellt. In diesem Zuge wurde die bestehende Datenbank ebenfalls geteilt und entsprechend der Services in neue Schemen und Datenbanken transferiert. Um Einschränkungen hinsichtlich Skalierung vorzubeugen, wurde der direkte Zugriff auf die Datenquellen unterbunden und dieser nur mehr über die jeweiligen Dienste ermöglicht. Um eine nahtlose Kommunikation zwischen den Services zu ermöglichen, diese bei Bedarf aggregieren zu können und systemübergreifende Mechanismen wie beispielsweise Monitoring, dezentrales Request Routing oder Request Tracking einzuführen, wurden alle Dienste um eine Webservice-Schnittstelle erweitert. Dabei wurden klare Regeln festgelegt, welche jedes Service einzuhalten hat und die Kommunikation zwischen den Services nur über diese Schnittstellen erlaubt. Diese Isolation der einzelnen Dienste und die dadurch entstehende lose Kopplung innerhalb des Systems, ermöglichte es Amazon, Services unabhängig voneinander weiter zu entwickeln und eine dem Unternehmenswachstum entsprechende Skalierung zu ermöglichen. (Hoff, 2007)

Um die Entwicklung und Bereitstellung dieser Dienste zu beschleunigen, wurden Tools entwickelt, welche heute anderen Entwicklungsteams zur Verfügung gestellt werden (Fulton III, 2015). Für die Bereitstellung der notwendigen Infrastruktur wurde die Amazon Web Services (AWS) Plattform erstellt (Iskold, 2006), welche als Vorreiter von heutigen Cloud Plattformen zählt (Fulton III, 2015). Diese Infrastruktur wird analog zur strikten Trennung zwischen den Services nach dem „shared nothing“ Prinzip zur Verfügung gestellt, um Abhängigkeiten zu vermeiden, die zu einer gegenseitigen Blockade führen können (Hoff, 2007).

Zu den Erfahrungen, die das Unternehmen im Zuge dieser Transformation sammelte, zählt nach Hoff (2007) unter anderem, dass in einem großen verteilten System Einfachheit der Schlüssel zum Erfolg sei. Sowohl die einzelnen Dienste als auch die verwendeten Technologien sollten so einfach wie möglich gehalten und versteckte Abhängigkeiten im Design vermieden werden, da diese zu einer Erhöhung der Komplexität führen. Zusätzlich sollte stets davon ausgegangen werden, dass in einem verteilten System einzelne Teile ausfallen können und daher Mechanismen benötigt werden, die darauf entsprechend reagieren. Hinsichtlich der Teilung des Systems solle genau überlegt werden, welche Funktionalitäten eine Zustandsverwaltung und welche geschäftlichen Abläufe eine transaktionsgesteuerte Ausführung benötigen, um so eine entsprechende Balance zwischen Verfügbarkeit und Konsistenz zu finden. (Hoff, 2007)

Neben den technologischen Änderungen wurde ebenfalls die Organisationstruktur adaptiert und kleine, eigenständige Teams gebildet, welche die vollständige Verantwortung für einzelne Services übernahmen. Somit konnte die herkömmliche Trennung zwischen Entwicklung und Betrieb aufgehoben und der Kommunikationsaufwand reduziert werden. (Wolff, 2006) Diese Vorgehensweise sollte in weiterer Folge die Motivation des Teams erhöhen, Anwendungen zu entwickeln, die einfach zu konfigurieren und verwalten sind, da sie im Betrieb diese Aufgaben

selbst zu übernehmen haben. Weiters legt das Unternehmen großen Wert darauf, dass diese Teams möglichst nahe am Kunden agieren, um dessen Probleme besser zu verstehen, den Einfluss ihrer Arbeit darauf erkennen zu können und dadurch die Qualität der Dienste zu verbessern. (Hoff, 2007)

4.4.2 Netflix

Gesteigerte Nutzerzahlen und das damit einhergehende Wachstum des Unternehmens führte dazu, dass Netflix gezwungen war, die Architektur der eingesetzten Softwarelösungen zu überarbeiten, um dadurch eine dem Wachstum entsprechende Skalierbarkeit zu ermöglichen. Netflix musste die Entscheidung treffen, die eigenen Datencenter auszubauen und geeignete Speicherlösungen, Netzwerk-Infrastruktur und Ausfallssicherung selbst zu betreiben oder auf eine bestehende Cloud-Infrastruktur zurückzugreifen. Da sich das Unternehmen durch Produktinnovation und Kundenerlebnis von der Konkurrenz differenzieren will, entschied man sich die Datencenter-Infrastruktur auszulagern und durch Amazon betreiben zu lassen. Dies ermöglicht Netflix eine gezielte Skalierung der Systeme und Erschließung neuer Märkte, ohne regelmäßig abschätzen zu müssen, wie groß die Kundenbasis in der Zukunft sein könnte und die Infrastruktur entsprechend anpassen zu müssen. (Ciancutti, 2010)

Um alle Dienste ohne Ausfallszeiten in die Cloud zu transferieren, wurde eine umfassende Migrationsstrategie ausgearbeitet, welche unter anderem auch eine Überführung des geschäftskritischen Billing-Systems vorsah (Sangeeta, 2016).

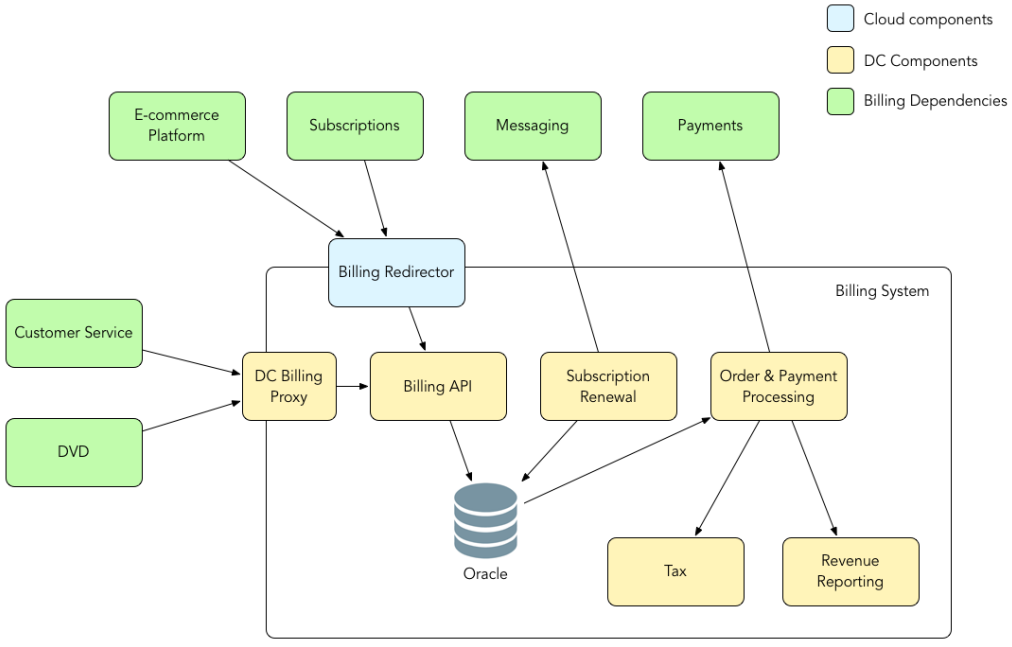

Abbildung 4-10: Architektur Netflix Billing-System vor der Migration (Sangeeta, 2016)

Kern dieses Systems war eine umfangreiche relationale Datenbank, auf welche von unterschiedlichen Anwendungen zugegriffen wurde und in Abbildung 4-10 ersichtlich ist. Für die Migration wurde eine schrittweise Vorgehensweise festgelegt, bei welcher zu Beginn das bestehende System vereinfacht wurde. Dabei wurde nicht mehr benötigter Code bereinigt und

der verbleibende in einzelne Module aufgeteilt. Als nächster Schritt wurden Abhängigkeiten zwischen den Modulen entfernt, erste Services ausgegliedert und diese in die Cloud verschoben. Weiters wurde die Datenbank um historische und nicht mehr benötigte Daten bereinigt, Datenbankstrukturen überarbeitet und Teile davon gemeinsam mit den Services in die Cloud-Datenbank überführt. Im Anschluss wurden weitere Daten, die lediglich für das User-Interface benötigt werden und nicht Teil von ACID-Transaktion sein müssen, ausgelagert und in eine verteilte, hochverfügbare Datenquelle verschoben, um diese zukünftig nach dem *Eventual Consistency* Prinzip zu verarbeiten. Die bestehende relationale Datenbank beinhaltete nun nur mehr solche Daten, die für den Bezahlprozess benötigt werden und Teil einer ACID-Transaktion sein müssen. Abschließend wurde diese Datenbank ebenfalls in die Cloud überführt und mittels Replikation ausfallssicher gestaltet. (Sangeeta, 2016)

Die durch diese Migration entstandene cloudbasierte System-Architektur wird in Abbildung 4-11 dargestellt.

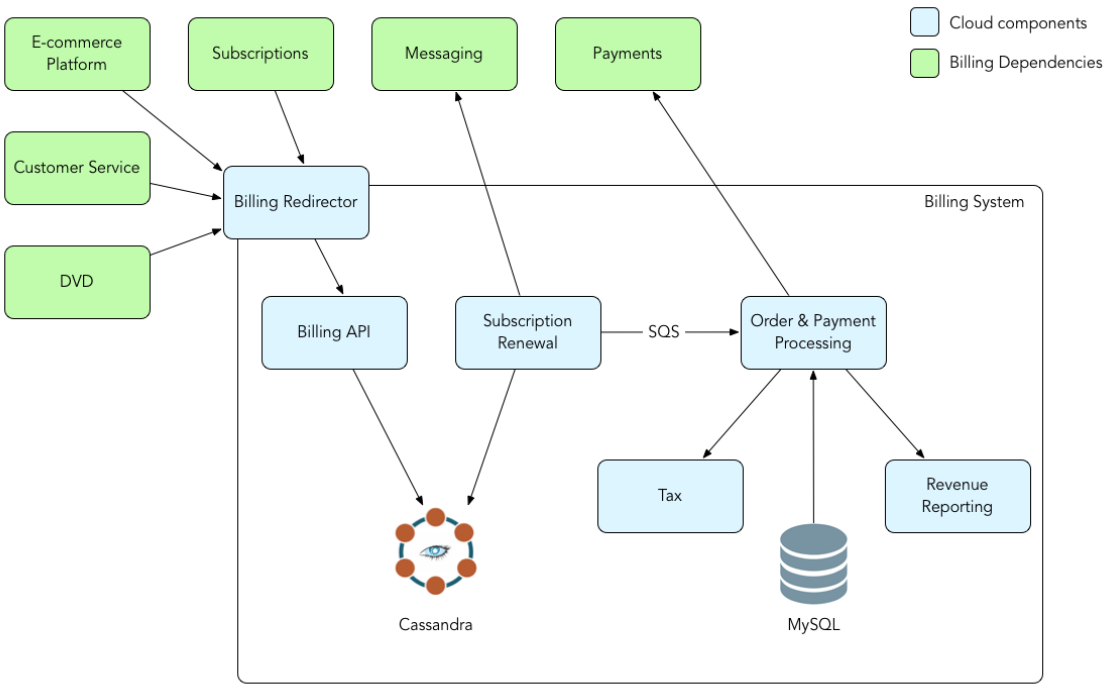

Abbildung 4-11: Architektur Netflix Billing-System nach Migration (Sangeeta, 2016)

Auf eine weitere Beschreibung dieser Migration wird an dieser Stelle verzichtet und auf Parulekar und Pilani (2016) sowie Sangeeta (2016) verwiesen, diese geben weitere Details zur Überführung der Datenbankstrukturen und Daten in die neue Umgebung.

Hinsichtlich dem Einsatz von Microservices orientierte sich Netflix am Beispiel von Amazon und setzt ebenfalls auf kleine, voneinander unabhängige Teams, die eigenständig Services entwickeln, betreiben und die Verantwortung dafür übernehmen (Newman, 2015). Neben den Teams, die für die Weiterentwicklung des Systems verantwortlich sind, werden eigene Plattform-Teams eingesetzt, die Support für die Infrastruktur leisten und dabei Werkzeuge zur Verfügung stellen, welche die in einer Microservices-Architektur notwendigen service-übergreifenden Aufgaben wie beispielsweise Service Discovery, Konfiguration oder Monitoring erledigen. Um

dabei eine lose Kopplung zu erreichen, stellen diese Tools analog zu den Microservices definierte Schnittstellen bereit und können nur über diese aufgerufen werden. (Mauro, 2015)

Einhergehend mit den organisatorischen Änderungen hat Netflix auch die Unternehmenskultur entsprechend gestaltet, um durch den Einsatz von Microservices das Wachstum des Unternehmens zu fördern. Netflix ist der Meinung, dass durch Kontrolle im Entwicklungsprozess zur Vermeidung von Fehlern und mehrfachen Aufwänden zwar eine Steigerung der Effizienz möglich ist, dabei jedoch ständig neue Regeln festgelegt werden müssen, welche in Folge dazu führen, dass Unternehmen immer langsamer agieren können. Um dies zu vermeiden, ordnet das Unternehmen der Geschwindigkeit im Entwicklungs- und Bereitstellungsprozess einen höheren Stellenwert als der Effizienz dieser Prozesse zu, da dies eine schnellere Reaktion auf Kundenwünsche ermöglicht und den Fokus auf mögliche Einnahmen statt auf Einsparungen legt. Dabei wird das Prinzip verfolgt, dass, wenn ein Unternehmen schneller auf geänderte Marktbedingungen reagieren kann, die dabei beteiligten Prozesse von selbst effizienter werden. Um dies zu ermöglichen, lebt das Unternehmen eine Kultur, die den Mitarbeiterinnen und Mitarbeitern hohe Freiheitsgrade einräumt, jedoch auch viel Verantwortung überträgt und schreibt lediglich die Regel vor, dass Personen in jeder Situation im besten Interesse von Netflix zu handeln haben. Kann eine Person diese Regel in einer Situation nicht richtig anwenden, sollte dies hinterfragt und dabei überlegt werden, warum man jemanden anstellt, dessen Urteilsvermögen man nicht trauen kann. (Mauro, 2015)

Um die Geschwindigkeit der Prozesse zu erhöhen, wird auf *Continuous Delivery* und der Verwendung von Containertechnologien gesetzt. Jedes Microservice wird mit allen benötigten Abhängigkeiten in einem Container zusammengefasst und an die jeweilige *Continuous-Delivery-Pipeline* übergeben. Die Verwendung von Containertechnologien ermöglicht es, die Komplexität der Infrastruktur niedrig zu halten, da nur ein System zur Ausführung dieser Container bereitgestellt werden muss, welches dann auf beliebige Staging-Ebenen verteilt werden kann. (Mauro, 2015) Weiters wird aufbauend auf *Continuous Delivery* das OODA-Loop Konzept eingesetzt, welches ausgehend auf Beobachtungen und Big-Data Analysen, der Entwicklung ein weitgehend unabhängiges und iteratives Arbeiten ermöglicht, um dadurch Innovationen zu fördern (Pronschinske, 2015).

Weiters gilt Netflix als Vorreiter in Bezug auf das Veröffentlichen ihrer im Microservices Umfeld eingesetzten Tools als Open Source[5]. Einerseits soll dadurch sowohl die Kreativität in der Entwicklung gefördert sowie die Motivation gesteigert werden, die Qualität des Codes zu verbessern, als auch andererseits die Strategie verfolgt, ihre Technologien anderen Unternehmen zur Verfügung zu stellen, um dadurch zukünftige Trends der Industrie mitbestimmen zu können. (Pronschinske, 2015)

Hinsichtlich Qualitätssicherung setzt das Unternehmen auf verschiedene Teststrategien wie beispielsweise umfangreiche Integrationstests, um das Zusammenspiel zwischen Services zu prüfen, oder A/B Tests, welche das Verhalten einer neuen Funktionalität einer bisher

[5] Netflix OSS Stack: https://netflix.github.io

eingesetzten Lösung gegenüberstellen, um daraus Rückschlüsse ziehen zu können, ob diese neue Funktion einen größeren Mehrwert erzeugen kann. Weitere Details zu den dabei verwendeten Verfahren finden sich bei Urban, Sreenivasan und Kannan (2016) sowie Allanabanda, Kannan, Prasad und Veeraraghava (2016) und werden an dieser Stelle nicht näher erläutert.

Um die Verfügbarkeit und Zuverlässigkeit der Services weiter zu verbessern, werden unter dem Namen „Netflix Simian Army“ verschiedene Tools entwickelt, welche in unregelmäßigen Abständen beispielsweise das Verhalten von Services bei einem Ausfall des Netzwerkes oder anderer Services prüfen. Um reale Bedingungen zu schaffen, werden diese Tests direkt im produktiven System ausgeführt und sollen dazu führen, die Fehlertoleranz der Dienste nachhaltig zu verbessern. (Izrailevsky & Tseitlin, 2011) Diese Werkzeuge stehen ebenfalls als Open Source Software zur Verfügung und können zum Testen der eigenen Cloud-Infrastruktur verwendet werden.

4.4.3 Volkswagen

Der Automobilkonzern Volkswagen hat sich dafür entschieden, den Geschäftsfokus vom traditionellen Autohersteller auf einen Mobility-Service-Provider zu richten. Im Zuge der Definition der dafür notwendigen IT Strategie stellte man fest, dass Software schneller entwickelt und ausgeliefert werden muss, um Verbesserungen und neue Services bereitstellen zu können. Um dafür eine moderne, den Unternehmensrichtlinien entsprechende Infrastruktur aufzubauen, entschied man sich für eine Private Cloud Lösung. Zur Vermeidung von Herstellerabhängigkeiten wählte man dafür eine auf der quelloffenen OpenStack Cloud Plattform basierende Lösung[6]. Die für die Entwicklung notwendige Infrastruktur wurde mittels der ebenfalls quelloffenen PaaS-Lösung *Cloud Foundry*[7] zur Verfügung gestellt. Volkswagen konnte dadurch die Betriebskosten für die Infrastruktur um etwa die Hälfte senken und stellte in weiterer Folge eine Plattform zur Verfügung, die zur Erreichung der gewünschten Beschleunigung des Entwicklungsprozesses beitragen soll. (Finnegan, 2016)

Hinsichtlich der Überführung bestehender Systeme auf die neue Infrastruktur plant das Unternehmen eine eher langsame Vorgehensweise. Neuentwicklungen sollen direkt auf der neuen Plattform angesiedelt werden, bei bestehenden Legacy-Systemen werden in einem ersten Schritt nur solche überführt, die sich gut für eine Microservices-Architektur eignen. Dies führt Volkswagen darauf zurück, dass vor einer großflächigen Migration der Systeme eine der modernen Infrastruktur entsprechende Unternehmenskultur etabliert werden und das notwendige Wissen für die cloudbasierte Softwareentwicklung schrittweise gesammelt werden muss. (Finnegan, 2016)

Bei einem der ersten Systeme, das auf der neuen Plattform bereitgestellt werden soll, wurde eine neu zu entwickelnde Software für das unternehmensweite Lizenzmanagement gewählt. Dabei

[6] Mirantis OpenStack Cloud siehe https://www.mirantis.com/openstack-cloud/

[7] Cloud Foundry: https://www.cloudfoundry.org/

wurde auf eine Microservices-Architektur gesetzt, um die hohen Anforderungen in Bezug auf Verfügbarkeit und Skalierbarkeit erfüllen zu können. Weiters sollten durch das neue System verschiedene Altsysteme abgelöst werden, wofür eine möglichst flexible und erweiterbare Architektur benötigt wurde. Da sich Anforderungen in einem großen Konzern schnell ändern können, wurden *Continuous-Delivery-Pipelines* aufgesetzt und das Projekt nach einer agilen Vorgehensweise konzipiert. Die Entwicklung selbst erfolgte nach dem Test-Driven Development Ansatz und in mehreren Phasen wurde aus einzelnen Services das Gesamtsystem zusammengebaut. Das User-Interface für die neue Anwendung wurde als eigenes Microservice entwickelt, welches über ein API-Gateway mit den Backend-Services kommuniziert. Die Backend-Services selbst beinhalten jeweils eine eigene Datenbank und stellen ihre Daten und Funktionalitäten über eine REST-Schnittstelle zur Verfügung. (Bindick & Menk, 2016)

Nähere Details zu den eingesetzten Technologien und dem Umgang mit betrieblichen Aspekten wie beispielsweise Konfiguration oder Monitoring konnten aus den publizierten Quellen nicht ermittelt werden, nach Bindick und Menk (2016) konnten durch diese Lösung jedoch Betriebs- und Wartungskosten eingespart sowie die Time-to-Market erheblich verbessert werden.

4.4.4 Fazit

Die drei soeben beschriebenen Unternehmen skizzieren Ansätze, die für die Einführung von Microservices gewählt werden können und in der Praxis vermehrt aufzufinden sind. Vor allem Unternehmen mit einem internetbasierten Geschäftsmodell setzen verstärkt auf eine Cloud-Infrastruktur und damit einhergehend der Entwicklung von Microservices, um die Anforderungen hinsichtlich Skalierbarkeit und Verfügbarkeit zu erfüllen sowie die eigene Innovationsfähigkeit zu erhöhen. Stellvertretend für diese sei an dieser Stelle das Unternehmen Zalando zu erwähnen, welches ähnlich wie Netflix einen großen Wert auf quelloffene Tools legt und im Zuge des „Project Mosaic“[8] verschiedene Frameworks zur Verfügung stellt, die für die Erstellung des User-Interfaces in einer Microservices-Architektur eingesetzt werden können (Persa, 2015).

Aus der Betrachtung dieser Best-Practice-Ansätze können Rückschlüsse auf die eingangs gestellten Fragen gemacht werden, welche nachfolgend diskutiert werden.

Hinsichtlich der steigenden Komplexität des Gesamtsystems sowie der Erhöhung der betrieblichen Aufwände zeigen die diskutierten Praxisbeispiele, dass eine moderne Infrastruktur, die Verwendung von Containertechnologien sowie der durchgängige Einsatz von *Continuous-Delivery-Pipelines* Voraussetzungen für einfach zu verwaltende Systeme darstellen. Weiters sollten klare Schnittstellen definiert, die Kommunikation zwischen Diensten nur über diese erlaubt und versteckte Abhängigkeiten zwischen den Services vermieden werden. Zur Unterstützung der betrieblichen Aktivitäten sollte ein umfangreiches Logging und Monitoring vorgeschrieben und entsprechende Tools von einem eigens dafür zuständigen Team bereitgestellt werden. Dabei

[8] Project Mosaic https://www.mosaic9.org/

kann auf etablierte Werkzeuge zurückgegriffen werden, die unter anderem von den genannten Unternehmen als Open Source Software veröffentlicht wurden.

Um mit der steigenden organisatorischen Komplexität umzugehen, adaptieren viele Unternehmen ihre Unternehmenskultur, um den Teams mehr Freiräume zu ermöglichen, übertragen in diesem Zuge aber auch mehr Verantwortung auf diese. Demgegenüber geht aus der Betrachtung von Volkswagen hervor, dass der Einsatz von Microservices auch ohne Umstellung auf eigenständige Teams funktionieren kann. Dieses Beispiel zeigt auch eine Möglichkeit, wie das Microservices-Architekturparadigma auf Unternehmen angewendet werden kann, die kein rein internetbasiertes Geschäftsmodell aufweisen. Jedoch sollte auch in diesem Fall eine moderne Infrastruktur eingeführt werden, welche der Entwicklung eine geeignete Plattform zur Verfügung stellt und somit zu einer Verbesserung der Reaktionszeit beitragen kann.

Netflix zeigt anhand der Migration seines Billing-Systems, wie aus einer bestehenden Datenbank nur solche Teile ausgelagert werden, die für ein dezentrales Datenmanagement und *Eventual Consistency* geeignet sind. Daten, bei denen die Konsistenz wichtiger ist als die Verfügbarkeit, werden weiterhin in einer relationalen Datenbank gespeichert. Entscheidend ist, dass direkte Zugriffe auf die Datenbank verhindert werden und diese nur über die Schnittstelle eines einzelnen Service zugänglich ist.

In Bezug auf Tests konnte festgestellt werden, dass in der Praxis verstärkt auf A/B Tests gesetzt wird, um damit den Mehrwert einer neuen Funktionalität zu messen. Zum Testen der Schnittstellen empfiehlt sich der Einsatz von Consumer-Driven Contract Tests, bei welchen eine Schnittstelle aus Sicht des Konsumenten getestet wird (Wolff, 2015). Weiters sollten alle Tests möglichst automatisiert und als fester Bestandteil der *Continuous-Delivery-Pipeline* ausgeführt werden.

5 ERGEBNISSE, ZUSAMMENFASSUNG UND AUSBLICK

Die theoretische als auch praktische Ausarbeitung hat gezeigt, dass Microservices viele Vorteile mit sich bringen und sich für die Erweiterung oder Modernisierung von Legacy-Systemen eignen können, diese jedoch auch Nachteile haben und nicht für jedes Unternehmen geeignet sind.

In diesem abschließenden Kapitel werden die Ergebnisse dieser Arbeit zusammengefasst und daraus eine Handlungsempfehlung abgeleitet. Weiters wird die eingangs gestellte Forschungsfrage beantwortet und abschließend ein Ausblick gegeben, welche weiteren Forschungsarbeiten aufbauend auf dieser möglich wären.

5.1 Handlungsempfehlung

Basierend auf den gewonnenen Erkenntnissen wird in diesem Abschnitt eine Handlungsempfehlung für die Adaptierung des Microservices-Architekturparadigmas und einer damit verbundenen Modernisierung und Erweiterung von monolithischen Legacy-Systemen vorgestellt. Die nachfolgende Empfehlung wird in zwei Teilen präsentiert, wobei der erste Teil Rahmenbedingungen für die Einführung von Microservices im Unternehmen behandelt, auf welchen aufbauend der zweite Teil Empfehlungen für die Vorgehensweise bei der Modernisierung und Erweiterung von Legacy-Systemen beschreibt.

5.1.1 Rahmenbedingungen für Microservices im Unternehmen schaffen

In diesem Abschnitt werden grundlegende Rahmenbedingungen beschrieben, die ein Unternehmen bei der Einführung von Microservices berücksichtigten sollte. Als grundlegende Regel wird an dieser Stelle die Aussage von Fowler (2015) herangezogen, welche die erhöhte Komplexität in den Vordergrund stellt und wie folgt lautet:

> *“So my primary guideline would be don't even consider microservices unless you have a system that's too complex to manage as a monolith. The majority of software systems should be built as a single monolithic application. Do pay attention to good modularity within that monolith, but don't try to separate it into separate services.”* (Fowler, 2015, Abs. 4)

Auch wenn man dieser Empfehlung folgt, können einzelne Elemente der folgenden Ausführungen hilfreich sein, um den Prozess der Entwicklung und Bereitstellung von Software zu beschleunigen und damit einhergehend eine Steigerung des Kundennutzens zu erzielen. Boyd (2016) beschreibt in diesem Zusammenhang, dass das Microservices-Architekturparadigma nicht nur als Ansatz für Software-Architekturen betrachtet werden sollte, sondern auch als philosophischer Ansatz angesehen werden kann, welcher die Art und Weise, wie ein Unternehmen arbeitet und sich organisiert, berücksichtigt und somit nachhaltige Verbesserung der Unternehmenskultur bewirken kann.

Cloudbasierte Infrastruktur einführen

Wie aus der vorangegangen Marktbetrachtung ersichtlich, setzen Unternehmen, die Microservices erfolgreich eingeführt haben, auf eine moderne, cloudbasierte Infrastruktur. Diese erhöht die Flexibilität des Unternehmens, um schneller auf geänderte Marktbedingungen zu reagieren, und können zu einer Reduktion der Infrastrukturkosten führen.

Ob dabei eine Private oder Public Cloud Lösung eingesetzt wird, hängt unter anderem von den Anforderungen des jeweiligen Unternehmens hinsichtlich Datenschutz oder Sicherheit ab. Weiters sollte bei der Entscheidungsfindung berücksichtigt werden, ob das Unternehmen über entsprechendes Personal mit dem notwendigem betrieblichen Know-How verfügt, um eine Private Cloud Infrastruktur betreiben zu können. (Jadeja & Modi, 2012)

Application Development Layer bereitstellen

Hinsichtlich dem Einsatz von Microservices sollte eine PaaS-Plattform gewählt oder Containertechnologien wie beispielsweise Docker eingesetzt werden (Büst, 2016), um der Entwicklung einen „Application Development Layer“ zur Verfügung zu stellen (Finnegan, 2016). PaaS bieten eine weitreichend automatisierte und standardisierte Umgebung zur Bereitstellung von Anwendungen, schränken jedoch die freie Technologiewahl von Microservices ein, da nur Technologien verwendet werden können, welche von der Plattform unterstützt werden. Containertechnologien bieten hingegen einen größeren Freiheitsgrad hinsichtlich Technologiewahl und ermöglichen beispielsweise auch die Bereitstellung von Legacy-Applikationen innerhalb eines Containers, der betriebliche Aufwand ist jedoch in der Regel höher als bei PaaS-Lösungen. (Wolff, 2015)

Nach Büst (2016) ist es nicht mehr notwendig, eine Entscheidung zwischen PaaS oder Containertechnologien zu treffen, da PaaS-Anbieter vermehrt die Unterstützung von Container-Deployments anbieten und somit die Vorteile dieser beiden Ansätze kombinieren.

Continuous-Delivery-Pipelines aufsetzen

Aufbauend auf der zuvor beschriebenen Infrastruktur sollten entsprechende Werkzeuge zur Verfügung gestellt werden, die einen möglichst hohen Grad der Automatisierung ermöglichen und die betrieblichen Aktivitäten unterstützen. Um Abhängigkeiten zu vermeiden, sollte jedes bereitzustellende Artefakt über eine eigene *Continuous-Delivery-Pipeline* verfügen, um es dadurch schnell und einfach auf das jeweilige Zielsystem ausrollen zu können.

Werkzeuge für Cross-Cutting Concerns vorgeben

Um die Komplexität und betrieblichen Aufwände zu reduzieren, sollten einheitliche Tools eingesetzt werden, welche die Konfiguration und Überwachung der Services ermöglichen sowie die Kommunikation und Koordination zwischen den Diensten regeln. Diese Werkzeuge sollten selbst als Service bereitgestellt und in die Verantwortlichkeit eines Teams übertragen sowie nur über definierte Schnittstellen angesprochen werden (Mauro, 2015).

Wolff (2015) empfiehlt dabei die Erstellung von Vorlagen für Microservices, welche aufbauend auf den in der Makro-Architektur definierten Regeln einen grundlegenden Aufbau eines Service vorgeben und dabei die Verwendung der Schnittstellen zu diesen Werkzeugen vorzeigen.

Anforderungen an die Organisation berücksichtigen

Wie ausführlich diskutiert, sollte ein Microservice nur in der Verantwortung eines einzelnen Teams sein und von diesem unabhängig entwickelt und betrieben werden, um möglichst viele Vorteile durch die Einführung dieses Architekturansatzes zu erhalten. Weiters sollten Werkzeuge bereitgestellt werden, welche die Kommunikation und den Informationsaustausch innerhalb eines Unternehmens fördern wie beispielsweise Chat-Tools oder Entwickler-Blogs (Boyd, 2016). Die erhöhten Anforderungen an den Betrieb und die Steigerung der technischen Komplexität von Systemen erfordern ein entsprechend qualifiziertes Personal sowie eine möglichst hohe Lernbereitschaft der Mitarbeiterinnen und Mitarbeiter (Fowler & Lewis, 2015).

Wolff (2017) empfiehlt in diesem Zusammenhang, von Unternehmen wie Netflix oder Amazon nicht nur die technischen Lösungen zu übernehmen, sondern vielmehr zu überlegen, welche Aspekte der Organisation und Unternehmenskultur in das eigene Unternehmen übernommen werden können, um dadurch Microservices erfolgreich einzusetzen.

5.1.2 Erweiterung und Modernisierung monolithischer Legacy-Systeme

Aufbauend auf der zuvor beschriebenen Handlungsempfehlung für die Einführung von Microservices in Unternehmen wird nachfolgend eine empfohlene Vorgehensweise für die Erweiterung und Modernisierung von monolithischen Legacy-Systemen beschrieben.

Rahmenbedingungen schaffen und Migrationsstrategie festlegen

Bevor mit der Modernisierung von Legacy-Systemen begonnen wird, sollten die im vorherigen Abschnitt beschriebenen Rahmenbedingungen geschaffen werden, um damit eine geeignete Zielplattform für die Einführung von Microservices im Umfeld einer Legacy-Applikation verwenden zu können.

Wie aus den Ergebnissen dieser Arbeit abgeleitet werden kann, eignet sich im Zuge des Software-Reengineerings eine schrittweise Vorgehensweise für die Erweiterung und Modernisierung von monolithischen Legacy-Systemen. Dafür kann die in Abschnitt 3.1.3 vorgestellte *Architecture Improvement Method* verwendet werden, welche nach Starke (2017) eine iterative Vorgehensweise unter Berücksichtigung des Kosten-Nutzen-Verhältnisses beschreibt.

Monolithische Applikation stabilisieren und vereinfachen

Zu Beginn sollte der Fokus auf das bestehende System gelegt und dabei dem *Stabilize First* Prinzip gefolgt werden (Landwehr & Kraus, 2016). In dieser Phase der Stabilisierung sollte unter anderem das bestehende System um nicht mehr benötigte Funktionalitäten bereinigt und Abhängigkeiten zwischen bestehenden Modulen gelöst werden, um dadurch eine Vereinfachung des Systems herbeizuführen (Sangeeta, 2016). Boyd (2016) bezieht sich in diesem Zusammenhang auf das von Toyota entwickelte „Lean Transformation“ Modell, welches eine Stabilisierung und Optimierung vor der Transformation vorsieht.

Schrittweise Erweiterung und Modernisierung durchführen

Welche Methoden für eine schrittweise Erweiterung und Modernisierung gewählt werden, hängt vom jeweiligen Anwendungsfall ab.

Soll ein bestehendes System lediglich um neue Funktionalitäten erweitert werden, empfiehlt sich der Einsatz der *Strangler Application* Methode, um Microservices im Umfeld der Legacy-Applikation zu platzieren.

Wird eine Teilung des Altsystems angestrebt, sollten mit Hilfe von *Bounded Contexts* zusammenhängende Bereiche ermittelt und fachliche Grenzen festgelegt werden. Hinsichtlich dieser Grenzen sollten jedoch auch technische Aspekte berücksichtigt werden, bei welchen beispielsweise zu prüfen ist, was für eine Art des Datenmanagements die geschäftlichen Aktivitäten erlauben und ob diese nach dem *Eventual Consistency* Prinzip abgebildet werden können. Anschließend können Module der Reihe nach aus dem monolithischen System losgelöst und in eigene Microservices ausgelagert werden, wobei ebenfalls auf die im Verlauf dieser Arbeit diskutierten Strategien zurückgegriffen werden kann.

Microservices entwickeln

In Bezug auf die Entwicklung von Microservices und der damit verbundenen Steigerung der Komplexität kann der Einsatz von in der Praxis erprobten Open Source Tools empfohlen werden. Hier finden sich mittlerweile genügend Werkzeuge, welche einerseits den betrieblichen Aufwand reduzieren und andererseits eine Hilfestellung für die Entwicklung von robusten und fehlertoleranten Services liefern. Das im Zuge dieser Arbeit bereits erwähnte „Netflix Open Source Software Center“ bietet dafür eine reichliche Auswahl. In Hinblick auf die Integration auf Ebene der Benutzerschnittstelle liefert das von Zalando federführend entwickelte „Project Mosaic9“ verschiedene Frameworks. Diese Werkzeuge können vor allem dabei behilflich sein, die technische Komplexität unter Kontrolle zu bringen und den Aufbau von Microservices-Architekturen zu beschleunigen.

5.2 Zusammenfassung und Diskussion

In dieser Arbeit wurden die Voraussetzungen für den Einsatz von Microservices zur Erweiterung und Modernisierung von monolithischen Webapplikationen sowie die Erwartungshaltung an dieses Architekturparadigma in der Praxis untersucht.

Zu Beginn wurden ausgehend von einer Literaturrecherche die Grundlagen von Microservices-Architekturen diskutiert und deren Vor- und Nachteile beleuchtet. Dabei wurde festgestellt, dass dieses Architekturmuster zwar zu einer Reduktion der fachlichen Komplexität einer Anwendungsdomäne führen und Qualitätsaspekte einer Software wie beispielsweise die Wartbarkeit nachhaltig verbessern kann, dabei jedoch auch wesentliche Nachteile gegenüber einer monolithischen Lösung entstehen, die zu einer Erhöhung der technischen und organisatorischen Komplexität führen und somit vor allem eine Steigerung der betrieblichen Aufwände zur Folge haben können.

Im weiteren Verlauf wurden Möglichkeiten zur Erweiterung und Modernisierung von monolithischen Webapplikationen ermittelt und deren Anwendbarkeit auf Microservices untersucht. Hier zeigte sich, dass sich keine allgemeingültigen Vorgehensweisen festlegen lassen, jedes Altsystem gesondert betrachtet werden muss und eine geeignete Strategie für das jeweilige Migrationsvorhaben definiert werden sollte. Aus den Ergebnissen lässt sich jedoch ableiten, dass eine schrittweise Vorgehensweise einer Big-Bang Migration zu bevorzugen ist und der Einsatz von Microservices eine geeignete Alternative darstellt. Wichtig dabei ist, den Fokus vor allem auf die abzubildende Fachlichkeit zu richten und nicht nur technische Aspekte zu betrachten. Als geeignetes Mittel dafür hat sich unter anderem das Domain-Driven Design gezeigt.

Anschließend wurden im empirischen Teil dieser Arbeit Hypothesen aufgestellt und darauf aufbauend ein Fragebogen erstellt, um damit die Erwartungshaltung von Personen im Umfeld der Softwareentwicklung an dieses Architekturparadigma zu messen. Aus dieser Evaluierung geht hervor, dass sich die Befragten unter anderem eine Verbesserung der Wartbarkeit und damit einhergehend eine Reduktion der Wartungs- und Betriebskosten erhoffen. Auch hier zeigte sich, dass die Komplexität bestehender Systeme einen hohen Stellenwert einnimmt und positive Auswirkungen auf die Bereitschaft zur Einführung von Microservices hat. Weiters ist hervorzuheben, dass zwar ein Großteil der Befragten Microservices in zukünftigen Projekten einsetzen will, die Ergebnisse jedoch auch zeigen, dass der dafür notwendige Erfahrungslevel im Umgang mit relevanten technologischen Ansätzen teilweise fehlt und hier sowohl seitens des Personals als auch des Unternehmens eine Bereitschaft zur Weiterbildung gegeben sein sollte.

Abschließend wurde eine Marktbetrachtung durchgeführt und Best-Practice-Ansätze in der Praxis ermittelt, um damit die zuvor erhobenen Ergebnisse zu validieren. Diese Untersuchung lieferte Antworten auf die zuvor ermittelten Fragestellungen und zeigte, dass vor allem die in einem Unternehmen vorherrschende Kultur einen wichtigen Beitrag zu einer erfolgreichen Einführung des Microservices-Architekturparadigmas leisten kann. Weiters konnten dabei Werkzeuge ermittelt werden, welche eine Hilfestellung hinsichtlich der steigenden Komplexität des Gesamtsystems darstellen können.

Somit kann die zu Beginn dieser Arbeit gestellte Forschungsfrage *„Unter welchen Voraussetzungen können monolithische Java Enterprise Webapplikationen durch autonom agierende Microservices schrittweise abgelöst oder erweitert werden?“* wie folgt beantworten werden.

Grundsätzlich lässt sich aus den vorliegenden Ergebnissen ableiten, dass es keine expliziten Voraussetzungen gibt, die bei einer Nichterfüllung eine schrittweise Ablöse oder Erweiterung einer monolithischen Webapplikation verhindern würden. Selbst Systeme, welche nicht teilbar sind und somit eine inkrementelle Aufteilung auf einzelne Services unterbinden, können mittels Microservices und beispielsweise der *Strangler Application* Vorgehensweise erweitert werden.

Um jedoch ein System schrittweise abzulösen und durch den Microservices-Architekturansatz zu modernisieren, sollten sowohl auf technischer als auch organisatorischer Seite folgende Voraussetzungen erfüllt werden, um durch diese Vorgehensweise keine negativen Auswirkungen auf das Unternehmen zu erwirken:

- Ein System sollte eine ausreichend hohe fachliche Komplexität aufweisen, damit sich der durch Microservices zu erwartende Mehraufwand rentiert und zu keiner Einschränkung der Produktivität führt. Dabei muss vor allem die zu erwartende Steigerung der Komplexität des Gesamtsystems handhabbar sein und das Unternehmen bereit sein, die dafür notwendigen Ressourcen zur Verfügung zu stellen.
- Weiters sollte die Teilung des Systems nach fachlichen Aspekten erfolgen und Teams gebildet werden, in deren Verantwortung die Abbildung dieser fachlichen Anforderungen vollständig übertragen werden. Seitens des Unternehmens sind dafür die Bereitschaft zur Übertragung der Verantwortung und der Durchführung von organisatorischen Änderungen erforderlich.
- Auch hinsichtlich der technischen Umsetzung ist die Fachlichkeit in den Vordergrund zu stellen, aus welcher hervorgehen muss, ob die abzubildenden Prozesse ein verteiltes Datenmanagement ermöglichen und vorübergehende Inkonsistenzen akzeptiert werden können.
- Um schlussendlich tatsächliche Vorteile in Bezug auf die Verbesserung der Reaktionszeit auf geänderte Anforderungen zu erzielen, wird eine entsprechende Infrastruktur sowie ein hoher Grad an Automatisierung in der Entwicklung und Bereitstellung benötigt.

5.3 Ausblick

Diese Arbeit hat gezeigt, dass sich der Microservices-Ansatz vor allem in Unternehmen wiederfindet, welche ein internetbasiertes Geschäftsmodell aufweisen und hohe Anforderungen an die Skalierbarkeit und Verfügbarkeit der eingesetzten Systeme stellen. Dies erfordert jedoch in der Regel Abstriche bei der Konsistenz, was in vielen Unternehmen nicht möglich oder nicht gewollt ist. Kürzlich wurde von Srivastava (2017) die „Google Cloud Spanner“ Technologie angekündigt, ein Ansatz, welcher die Vorteile von relationalen und NoSQL Datenbanken vereinen und dadurch ACID-Transaktionen in hoch-skalierbare Datenbanken ermöglichen soll. Dabei könnte untersucht werden, welche Auswirkungen diese neue Technologie auf Microservices und vor allem auf die Möglichkeiten hinsichtlich der Modernisierung von monolithischen Systemen hat.

Weiters wurden im Zuge dieser Arbeit verschiedene Open Source Werkzeuge erwähnt, welche die Einführung von Microservices beschleunigen und als Mittel gegen die steigende technische Komplexität gelten. Hier könnte geprüft werden, in welchem Ausmaß sich diese Tools für eine Integration in bestehende monolithische Anwendungen eignen und ob dies die Einführung von Microservices vereinfachen kann.

Außerdem konnte nicht vollständig ermittelt werden, welchen Stellenwert die Verbesserung der Reaktionszeit auf geänderte Marktbedingungen hinsichtlich Microservices hat und ob diese die Bereitschaft zur Einführungen dieses Architekturansatzes erhöht. Dies könnte als zukünftiges Forschungsthema dienen und dabei in weiterer Folge die Möglichkeiten untersucht werden, welche neuen Geschäftsmodelle durch Microservices ermöglicht werden.

ANHANG A - Fragebogen

Umfrage „Modernisierung von Legacy-Systemen mit Microservices“

Im Rahmen meiner Diplomarbeit zum Thema "*Microservices im Praxiseinsatz - Modernisierung von Legacy-Systemen mit Microservices*" möchte ich herausfinden, welche Erwartungen an Microservices in der Praxis gestellt beziehungsweise ob Microservices für die Ablöse/Erweiterung von monolithischen Legacy-Systemen bereits eingesetzt werden.

Ich würde mich freuen, wenn Sie an meiner Befragung teilnehmen.

Allgemeine Informationen

Berufliche Position

○ Software-Entwickler ○ Software-Architekt ○ IT-Manager ○ Andere: ……..

Ihr Erfahrungslevel in der Softwareentwicklung

○ weniger als 3 Jahre ○ 3 bis 10 Jahre ○ mehr als 10 Jahre

Anzahl Mitarbeiter im Unternehmen

○ < 10 ○ 10 – 99 ○ 100 – 499 ○ > 500

Software Allgemein

#1: Wie wichtig sind Ihnen folgende Qualitätsmerkmale?

	sehr wichtig	eher wichtig	weniger wichtig	nicht wichtig
Änderbarkeit/Wartbarkeit	○	○	○	○
Austauschbarkeit	○	○	○	○
Benutzbarkeit	○	○	○	○
Effizienz/Performanz	○	○	○	○
Funktionalität	○	○	○	○
Fehlertoleranz	○	○	○	○
Wiederverwendbarkeit	○	○	○	○
Zuverlässigkeit/Verfügbarkeit	○	○	○	○

#2: Wenn Sie monolithische Softwarelösungen in Ihrem Arbeitsumfeld betrachten, wie oft treten folgende Probleme auf?

	oft	gelegentlich	selten	nie
eingeschränkte Verfügbarkeit	○	○	○	○
eingeschränkte Wartbarkeit	○	○	○	○
schlechte Performance	○	○	○	○
veraltete Technologien	○	○	○	○
hohe Komplexität	○	○	○	○
unzureichende Testabdeckung	○	○	○	○
lange Deployment-Zyklen	○	○	○	○
hohe Fehlerrate	○	○	○	○

#3: Sind Sie mit nachfolgenden Begriffen vertraut?

	Ja, in der Praxis eingesetzt	Ja, theoretische Kenntnisse	Nein
Service-orientierte Architekturen / verteilte Systeme	○	○	○
Resilience / Design for Failure	○	○	○
CAP-Theorem	○	○	○
Dezentrales Datenmanagement	○	○	○
Eventual Consistency	○	○	○
Domain-Driven Design	○	○	○
Ereignisgesteuerte Architekturen	○	○	○
Continuous-Delivery-Pipeline	○	○	○

#4: Sind in Ihrem Arbeitsumfeld verteilte Systeme im Einsatz?

○ Ja ○ Nein

#5: Ist in Ihrem Arbeitsumfeld die Bereitschaft gegeben, dezentrales Datenmanagement einzuführen und möglicherweise redundante Daten zu speichern?

○ ja ○ eher ja ○ eher nein ○ nein ○ keine Angabe

#6: Ist in Ihrem Arbeitsumfeld die Bereitschaft gegeben, Abstriche bei der Konsistenz von Daten hinzunehmen, um dadurch die Verfügbarkeit von Systemen zu erhöhen?

○ ja ○ eher ja ○ eher nein ○ nein ○ keine Angabe

Erwartungshaltung Microservices

#7: Sind Sie mit dem Thema Microservices vertraut?

○ Ja ○ Nein

#8: Haben Sie bereits praktische Erfahrungen mit Microservices?

○ Ja, Microservices sind produktiv im Einsatz

○ Ja, Microservices in Entwicklung

○ Nein, Evaluierung läuft

○ Nein, keine Erfahrung mit Microservices

○ Kein Interesse an Microservices

#9: In welchen Bereichen erwarten Sie sich die meisten Vorteile durch Microservices?

☐ Aufteilung der Komplexität eines Systems

☐ Selbstständige, voneinander unabhängige Teams

☐ Freie Technologiewahl

☐ Unabhängiges Deployment / Betrieb

☐ Reduktion der Betriebs-/Wartungskosten

☐ Verbesserung der Reaktionszeit auf geänderte Marktbedingungen
☐ Entwicklung neuer Geschäftsmodelle
☐ Erschließung neuer Kundenkreise
☐ *keine Erwartung von Vorteilen*
☐ Sonstige Vorteile: ……

#10: In welchen Bereichen erwarten Sie sich die meisten Nachteile durch Microservices?

☐ Erhöhung der Komplexität des Gesamtsystems
☐ Freie Technologiewahl
☐ Kommunikation zwischen Services
☐ Dezentrales Datenmanagement
☐ Organisatorische Änderungen
☐ Erhöhung des betrieblichen Aufwands
☐ *keine Erwartung von Nachteilen*
☐ Sonstige Nachteile: ……

#11: Was könnte aus Sicht Ihrer Kunden für die Einführung von Microservices sprechen?

……

#12: Planen Sie den Microservices-Ansatz bei einem Ihrer nächsten Projekte einzusetzen?

○ Ja ○ Nein

Betrieb / Infrastruktur

#13: Werden in Ihrem Arbeitsumfeld Tools eingesetzt, um ein (teil-) automatisiertes Bereitstellen von Software zu ermöglichen?

☐ Continuous Integration (Code Integration, Build, Unit Tests)
☐ Continuous Testing (Integrationstests, UI-Tests, manuelle Tests)
☐ Continuous Delivery (Software kann jederzeit per Knopfdruck bereitgestellt werden)
☐ Continuous Deployment (Code-Änderungen, die alle Qualitätskriterien erfüllen, werden sofort in Produktion übernommen)
☐ Nein, keine Tools im Einsatz
☐ Nicht bekannt
☐ Sonstige Tools/Ansätze: ……..

#14: Werden in Ihrem Arbeitsumfeld Containertechnologien (zb Docker, Kubernetes) eingesetzt?

○ Entwicklung und Deployment erfolgt mit Containertechnologien
○ Containertechnologien in Evaluierung
○ Nein, keine Containertechnologien im Einsatz
○ Nicht bekannt

#15: Werden in Ihrem Arbeitsumfeld Cloudbasierte Dienste eingesetzt?

☐ Public Cloud
☐ Private Cloud
☐ Platform as a Service (PaaS)
☐ Infrastructure as a Service (IaaS)
☐ Nein, Cloudbasierte Dienste werden nicht genutzt
☐ Nicht bekannt
☐ Sonstige Dienste: ……..

Organisation

#16: Erfolgt in Ihrem Arbeitsumfeld eine organisatorische Trennung von Entwicklung und Betrieb?

○ Ja ○ Nein

#17: Werden in Ihrem Arbeitsumfeld DevOps-Ansätze verfolgt?

○ Ja ○ Nein

#18: Besteht in Ihrem Arbeitsumfeld die Bereitschaft organisatorische Änderungen durchzuführen, um neue technologische Ansätze einzuführen?

○ Ja ○ Nein

#19: Der Microservices-Ansatz ermöglicht die Umsetzung selbstständiger, voneinander unabhängiger Services. Würden Sie die Verantwortung für Entwicklung, Betrieb und Qualitätssicherung dieser Services an einzelne Teams übertragen?

○ Ja ○ Nein

Legacy-Anwendungen

#20: Welche sind in der Regel die Hauptgründe, um ein monolithisches Altsystem abzulösen/zu modernisieren?

☐ Veraltete Technologien
☐ Eingeschränkte Wartbarkeit
☐ Hohe Wartungs-/Betriebskosten
☐ Hohe Komplexität des Systems
☐ Neue Funktionalitäten
☐ Sonstige: ……

#21: Planen Sie aktuell eine Ablöse/Modernisierung einer bestehenden Software-Applikation?

○ Ja, Microservices werden eingesetzt
○ Ja, Einsatz von Microservices wird evaluiert
○ Ja, Microservices werden nicht eingesetzt

○ Nein, aktuell nicht geplant
○ Nicht bekannt

#22: Welche Voraussetzungen muss eine monolithische Anwendung aus Ihrer Sicht erfüllen, um für eine Erweiterung mittels Microservices geeignet zu sein?

……

#23: Gibt es aus Ihrer Sicht Gründe, die gegen den Einsatz von Microservices sprechen, um Altsysteme zu modernisieren?

……

#24: Sind Ihnen Strategien für die Ablöse/Erweiterung von monolithischen Systemen mittels Microservices bekannt?

○ Ja ○ Nein

#24.1: Wenn ja, welche?

……

Abschluss

#25: Treffen Sie in ihrem Arbeitsumfeld technische Entscheidungen?

○ Ja ○ Nein

#26: Treffen Sie in ihrem Arbeitsumfeld organisatorische Entscheidungen?

○ Ja ○ Nein

#27: Gibt es aus Ihrer Sicht noch wichtige Punkte zu den Themen Microservices und Legacy-Applikationen, welche in diesem Fragebogen nicht erwähnt wurden? Was wären für Sie beispielsweise No-Gos bei der Einführung von Microservices?

……

ANHANG B - Ergebnisse der Befragung

Berufliche Position		
Antwort	Anzahl	Prozent
Software-Entwickler (A1)	31	46,97%
Software-Architekt (A2)	20	30,30%
IT-Manager (A3)	13	19,70%
Sonstiges	2	3,03%

Ihr Erfahrungslevel in der Softwareentwicklung		
Antwort	Anzahl	Prozent
weniger als 3 Jahre (A1)	5	7,58%
3 bis 10 Jahre (A2)	27	40,91%
mehr als 10 Jahre (A3)	34	51,52%

Anzahl Mitarbeiter im Unternehmen		
Antwort	Anzahl	Prozent
< 10 (A1)	10	15,15%
10 - 99 (A2)	33	50,00%
100 - 499 (A3)	9	13,64%
> 500 (A4)	14	21,21%

#1: Wie wichtig sind Ihnen folgende Qualitätsmerkmale?

	sehr wichtig	eher wichtig	weniger wichtig	nicht wichtig
Änderbarkeit/Wartbarkeit	54	12		
Zuverlässigkeit/Verfügbarkeit	43	22	1	
Benutzbarkeit	42	22	2	
Funktionalität	36	28	2	
Effizienz/Performanz	29	33	4	
Fehlertoleranz	23	33	10	
Wiederverwendbarkeit	18	27	19	2
Austauschbarkeit	10	33	23	

#2: Wenn Sie monolithische Softwarelösungen in Ihrem Arbeitsumfeld betrachten, wie oft treten folgende Probleme auf?

	oft	gelegentlich	selten	nie
unzureichende Testabdeckung	45	17	4	
hohe Komplexität	43	19	4	
eingeschränkte Wartbarkeit	41	18	7	
veraltete Technologien	38	23	5	
lange Deployment-Zyklen	21	23	19	3
schlechte Performance	14	39	12	1
hohe Fehlerrate	8	34	21	3
eingeschränkte Verfügbarkeit	4	28	32	2

#3: Sind Sie mit nachfolgenden Begriffen vertraut?

	Ja, in der Praxis eingesetzt	Ja, theoretische Kenntnisse	Nein
SOA / verteilte Systeme	41	24	1
Continuous-Delivery-Pipeline	28	30	8
Ereignisgesteuerte Architekturen	22	34	10
Dezentrales Datenmanagement	16	37	13
Domain-Driven Design	26	26	14
Resilience / Design for Failure	8	34	24
Eventual Consistency	11	27	28
CAP-Theorem	9	22	35

#4: Sind in Ihrem Arbeitsumfeld verteilte Systeme im Einsatz?

Antwort	Anzahl	Prozent
Ja (Y)	39	59,09%
Nein (N)	19	28,79%
keine Angabe	8	12,12%

#5: Ist in Ihrem Arbeitsumfeld die Bereitschaft gegeben, dezentrales Datenmanagement einzuführen und möglicherweise redundante Daten zu speichern?

Antwort	Anzahl	Prozent
ja (A1)	19	28,79%
eher ja (A2)	21	31,82%
eher nein (A3)	14	21,21%
nein (A4)	5	7,58%
keine Angabe (A5)	7	10,61%

#6: Ist in Ihrem Arbeitsumfeld die Bereitschaft gegeben, Abstriche bei der Konsistenz von Daten hinzunehmen, um dadurch die Verfügbarkeit von Systemen zu erhöhen?

Antwort	Anzahl	Prozent
ja (A1)	10	15,15%
eher ja (A2)	20	30,30%
eher nein (A3)	16	24,24%
nein (A4)	16	24,24%
keine Angabe (A5)	4	6,06%

#7: Sind Sie mit dem Thema Microservices vertraut?

Antwort	Anzahl	Prozent
Ja (Y)	54	81,82%
Nein (N)	12	18,18%

#8: Haben Sie bereits praktische Erfahrungen mit Microservices?

Antwort	Anzahl	Prozent
Ja, Microservices sind produktiv im Einsatz (A1)	15	22,73%
Ja, Microservices in Entwicklung (A2)	15	22,73%
Nein, Evaluierung läuft (A3)	14	21,21%
Nein, keine Erfahrung mit Microservices (A6)	21	31,82%
Kein Interesse an Microservices (A5)	1	1,52%
keine Angabe	0	0,00%

#9: In welchen Bereichen erwarten Sie sich die meisten Vorteile durch Microservices?

Antwort	Anzahl	Prozent
Aufteilung der Komplexität eines Systems (SQ001)	47	71,21%
Selbstständige, voneinander unabhängige Teams (SQ002)	28	42,42%
Freie Technologiewahl (SQ003)	32	48,48%
Unabhängiges Deployment / Betrieb (SQ004)	43	65,15%
Reduktion der Betriebs-/Wartungskosten (SQ005)	14	21,21%
Verbesserung der Reaktionszeit auf geänderte Marktbedingungen (SQ006)	17	25,76%
Entwicklung neuer Geschäftsmodelle (SQ007)	11	16,67%
Erschließung neuer Kundenkreise (SQ008)	8	12,12%
keine Erwartung von Vorteilen (SQ009)	8	12,12%
Sonstiges	5	7,58%

#10: In welchen Bereichen erwarten Sie sich die meisten Nachteile durch Microservices?

Antwort	Anzahl	Prozent
Erhöhung der Komplexität des Gesamtsystems (SQ001)	33	50,00%
Freie Technologiewahl (SQ002)	8	12,12%
Kommunikation zwischen Services (SQ003)	32	48,48%
Dezentrales Datenmanagement (SQ004)	26	39,39%
Organisatorische Änderungen (SQ005)	26	39,39%
Erhöhung des betrieblichen Aufwands (SQ006)	26	39,39%
keine Erwartung von Nachteilen (SQ007)	8	12,12%
Sonstiges	4	6,06%

#11: Was könnte aus Sicht Ihrer Kunden für die Einführung von Microservices sprechen?

Antwort	19	28,79%
keine Angabe	47	71,21%

#12: Planen Sie den Microservices-Ansatz bei einem Ihrer nächsten Projekte einzusetzen?

Antwort	Anzahl	Prozent
Ja (Y)	38	57,58%
Nein (N)	28	42,42%

#13: Werden in Ihrem Arbeitsumfeld Tools eingesetzt, um ein (teil-) automatisiertes Bereitstellen von Software zu ermöglichen?

Antwort	Anzahl	Prozent
Continuous Integration (Code Integration, Build, Unit Tests) (SQ001)	56	84,85%
Continuous Testing (Integrationstests, UI-Tests, manuelle Tests) (SQ002)	44	66,67%
Continuous Delivery (Software kann jederzeit per Knopfdruck bereitgestellt werden) (SQ003)	34	51,52%
Continuous Deployment (Code-Änderungen, die alle Qualitätskriterien erfüllen, werden sofort in Produktion übernommen) (SQ004)	15	22,73%
Nein, keine Tools im Einsatz (SQ005)	4	6,06%
Nicht bekannt (SQ006)	0	0,00%
Sonstiges	2	3,03%

#14: Werden in Ihrem Arbeitsumfeld Containertechnologien (zb Docker, Kubernetes) eingesetzt?

Antwort	Anzahl	Prozent
Entwicklung und Deployment erfolgt mit Containertechnologien (A1)	13	19,70%
Containertechnologien in Evaluierung (A2)	20	30,30%
Nein, keine Containertechnologien im Einsatz (A3)	28	42,42%
Nicht bekannt (A4)	5	7,58%
keine Angabe	0	0,00%

#15: Werden in Ihrem Arbeitsumfeld Cloudbasierte Dienste eingesetzt?

Antwort	Anzahl	Prozent
Public Cloud (SQ001)	15	22,73%
Private Cloud (SQ002)	22	33,33%
Platform as a Service (PaaS) (SQ003)	12	18,18%
Infrastructure as a Service (IaaS) (SQ004)	10	15,15%
Nein, Cloudbasierte Dienste werden nicht genutzt (SQ005)	27	40,91%
Nicht bekannt (SQ006)	6	9,09%
Sonstiges	0	0,00%

#16: Erfolgt in Ihrem Arbeitsumfeld eine organisatorische Trennung von Entwicklung und Betrieb?

Antwort	Anzahl	Prozent
Ja (Y)	33	50,00%
Nein (N)	33	50,00%
keine Angabe	0	0,00%

#17: Werden in Ihrem Arbeitsumfeld DevOps-Ansätze verfolgt?

Antwort	Anzahl	Prozent
Ja (Y)	39	59,09%
Nein (N)	15	22,73%
keine Angabe	12	18,18%

#18: Besteht in Ihrem Arbeitsumfeld die Bereitschaft organisatorische Änderungen durchzuführen, um neue technologische Ansätze einzuführen?

Antwort	Anzahl	Prozent
Ja (Y)	41	62,12%
Nein (N)	14	21,21%
keine Angabe	11	16,67%

#19: Der Microservices-Ansatz ermöglicht die Umsetzung selbstständiger, voneinander unabhängiger Services. Würden Sie die Verantwortung für Entwicklung, Betrieb und Qualitätssicherung dieser Services an einzelne Teams übertragen?

Antwort	Anzahl	Prozent
Ja (Y)	47	71,21%
Nein (N)	12	18,18%
keine Angabe	7	10,61%

#20: Welche sind in der Regel die Hauptgründe, um ein monolithisches Altsystem abzulösen/zu modernisieren?

Antwort	Anzahl	Prozent
Veraltete Technologien (SQ001)	44	66,67%
Eingeschränkte Wartbarkeit (SQ002)	46	69,70%
Hohe Wartungs-/Betriebskosten (SQ003)	40	60,61%
Hohe Komplexität des Systems (SQ004)	36	54,55%
Neue Funktionalitäten (SQ005)	26	39,39%
Sonstiges	2	3,03%

#21: Planen Sie aktuell eine Ablöse/Modernisierung einer bestehenden Software-Applikation?

Antwort	Anzahl	Prozent
Ja, Microservices werden eingesetzt (A1)	11	16,67%
Ja, Einsatz von Microservices wird evaluiert (A2)	19	28,79%
Ja, Microservices werden nicht eingesetzt (A3)	12	18,18%
Nein, aktuell nicht geplant (A4)	20	30,30%
Nicht bekannt (A5)	4	6,06%
keine Angabe	0	0,00%

#22: Welche Voraussetzungen muss eine monolithische Anwendung aus Ihrer Sicht erfüllen, um für eine Erweiterung mittels Microservices geeignet zu sein?

Antwort	25	37,88%
keine Angabe	41	62,12%

#23: Gibt es aus Ihrer Sicht Gründe, die gegen den Einsatz von Microservices sprechen um Altsysteme zu modernisieren?

Antwort	26	39,39%
keine Angabe	40	60,61%

#24: Sind Ihnen Strategien für die Ablöse/Erweiterung von monolithischen Systemen mittels Microservices bekannt?

Antwort	Anzahl	Prozent
Ja (Y)	14	21,21%
Nein (N)	52	78,79%

#24.1: Wenn ja, welche?

Antwort	12	18,18%
keine Angabe	4	6,06%

#25: Treffen Sie in ihrem Arbeitsumfeld technische Entscheidungen?

Antwort	Anzahl	Prozent
Ja (Y)	52	78,79%
Nein (N)	14	21,21%

#26: Treffen Sie in ihrem Arbeitsumfeld organisatorische Entscheidungen?

Antwort	Anzahl	Prozent
Ja (Y)	26	39,39%
Nein (N)	40	60,61%

#27: Gibt es aus Ihrer Sicht noch wichtige Punkte zu den Themen Microservices und Legacy-Applikationen, welche in diesem Fragebogen nicht erwähnt wurden? Was wären für Sie beispielsweise No-Gos bei der Einführung von Microservices?

Antwort	12	18,18%
keine Angabe	54	81,82%

ABKÜRZUNGSVERZEICHNIS

ACID	Atomicity, Consistency, Integrity, Durability
ACL	Anticorruption Layer
API	Application programming interface
AWS	Amazon Web Services
BASE	Basically Available, Soft state, Eventual consistency
CAP	Consistency, Availability, Partition Tolerance
DAA	Data Access Allocator
DDD	Domain-Driven Design
IaaS	Infrastructure-as-a-Service
JWT	JSON Web Tokens
MSA	Microservices-Architektur
OODA	Observe, Orient, Decide, Act
PaaS	Platform-as-a-Service
SCS	Self-Contained Systems
SOA	Service-orientierte Architekturen
SPA	Single-Page-Application
SSO	Single-Sign-On
UI	User Interface

ABBILDUNGSVERZEICHNIS

Abbildung 2-1: Module und Komponenten (Clements et al., 2010) 6
Abbildung 2-2: Einflussfaktoren für die Größe eines Microservice (Wolff, 2015) 16
Abbildung 2-3: Circuit Breaker (Nygard, 2007) 18
Abbildung 3-1: Verteilung des Wartungsaufwands (in Anlehnung an Sommerville, 2012) 24
Abbildung 3-2: Reengineeringpyramide nach Byrne (Masak, 2006) 28
Abbildung 3-3: Zusammenhänge zwischen Problemen, Maßnahmen und Kosten (Starke, 2016) 29
Abbildung 3-4: Database-First vs. Database-Last (Kaps, 2016) 31
Abbildung 3-5: System-Aufbau für die Migration nach der Composite-Database-Strategie (Starke, 2017) 32
Abbildung 3-6: Butterfly-Methode (Bisbal et al., 1999) 33
Abbildung 3-7: Branch by Abstraction (Smith, 2013) 35
Abbildung 3-8: Microservice Proxy (Hughes, 2013) 36
Abbildung 3-9: Architektur einer Strangler Application Vorgehensweise (Richardson & Smith, 2016) 37
Abbildung 3-10: Muster des Domain-Driven Design (Starke, 2015) 41
Abbildung 3-11: Bubble Context mit Anticorruption Layer (Evans, 2013) 43
Abbildung 3-12: Autonomous Bubble mit Domain Events (Evans, 2013) 44
Abbildung 3-13: Event Sourcing (Wolff, 2015) 48
Abbildung 3-14: UI Fragment Composition (Newman, 2015) 51
Abbildung 4-1: Stellenwert ausgewählter Qualitätsmerkmale von Software 63
Abbildung 4-2: Häufigkeit von Problemen monolithischer Anwendungen 63
Abbildung 4-3: Kenntnisse und Erfahrungen mit Begrifflichkeiten von verteilten Systemen 64
Abbildung 4-4: Bereitschaft für redundante Datenhaltung 64
Abbildung 4-5: Bereitschaft, Abstriche bei der Konsistenz von Daten hinzunehmen 64
Abbildung 4-6: Einsatz von Werkzeugen, um Software (teil-) automatisiert bereitzustellen 67
Abbildung 4-7: Bereitschaft, organisatorische Ände-rungen durchzuführen 68
Abbildung 4-8: Bereitschaft, Verantwortung zu übertragen 68
Abbildung 4-9: Einsatz von Microservices bei der Planung von Modernisierungsmaßnahmen von Altsystemen 68
Abbildung 4-10: Architektur Netflix Billing-System vor der Migration (Sangeeta, 2016) 79
Abbildung 4-11: Architektur Netflix Billing-System nach Migration (Sangeeta, 2016) 80

TABELLENVERZEICHNIS

Tabelle 3-1: Gegenüberstellung Big-Bang und inkrementelle Migration (Masak, 2006) 25
Tabelle 4-1: Merkmale der teilnehmenden Personen 62
Tabelle 4-2: Erwartete Vorteile durch den Einsatz von Microservices 65
Tabelle 4-3: Erwartete Nachteile durch den Einsatz von Microservices 66
Tabelle 4-4: Vorteile, die aus Kundensicht für die Einführung von Microservices sprechen könnten 66
Tabelle 4-5: Gründe für die Ablöse oder Modernisierung von monolithischen Applikationen 68
Tabelle 4-6: Voraussetzungen, die eine monolithische Anwendung für die Erweiterung mit Microservices erfüllen soll 69
Tabelle 4-7: Gründe, die gegen den Einsatz von Microservices sprechen, um ein Altsystem zu modernisieren 70
Tabelle 4-8: Erfahrung mit verschiedenen Methoden und Ansätzen im Microservices Umfeld 71
Tabelle 4-9: Probleme mit Altsystemen aufgrund hoher Komplexität 73
Tabelle 4-10: Komplexität als Grund für eine Modernisierung 73
Tabelle 4-11: Wartbarkeit und Kosten als Gründe für eine Modernisierung 74
Tabelle 4-12: Erfahrung mit verteilten Systemen und Bereitschaft, Microservices bei zukünftigen Projekten einzusetzen 75
Tabelle 4-13: Bereitschaft, dezentrales Datenmanagement einzuführen, in Bezug auf eine eingeschränkte Verfügbarkeit 76
Tabelle 4-14: Bereitschaft, Abstriche bei der Konsistenz zu machen, um die Verfügbarkeit eines Systems zu erhöhen 76

LITERATURVERZEICHNIS

Allanabanda, F., Kannan, H., Prasad, R., & Veeraraghavan, V. (2016, Juli 5). Product Integration Testing at the Speed of Netflix. Abgerufen von http://techblog.netflix.com/2016/07/product-integration-testing-at-speed-of-Netflix.html

Ambler, S. W., & Sadalage, P. J. (2006). *Refactoring Databases: Evolutionary Database Design*. Addison Wesley.

Balakrushnan, S., Mamnoon, O., Bell, J., Currier, B., Harrington, E., Helstrom, B., … Martins, M. (2016, Juli). Microservices Architecture. (The Open Group, Hrsg.). Abgerufen von https://www2.opengroup.org/ogsys/jsp/publications/PublicationDetails.jsp?publicationid=14540

Balzert, H. (2011). *Lehrbuch der Softwaretechnik: Entwurf, Implementierung, Installation und Betrieb*. Heidelberg: Spektrum Akademischer Verlag. Abgerufen von http://link.springer.com/10.1007/978-3-8274-2246-0

Becker, J., Mathas, C., & Winkelmann, A. (2009). *Geschäftsprozessmanagement*. Berlin, Heidelberg: Springer.

Bennett, K. (1995). Legacy systems: coping with success. *IEEE Software*, *12*(1), 19–23. doi:10.1109/52.363157

Bindick, S., & Menk, C. (2016, Dezember 16). Minimale Time-to-Market: Agilität und Continuous Delivery im Volkswagen Konzern. *OBJEKTspektrum*, (01/2017), 67–72.

Bisbal, J., Lawless, D., Wu, B., & Grimson, J. (1999). Legacy information systems: issues and directions. *IEEE Software*, *16*(5), 103–111. doi:10.1109/52.795108

Boyd, M. (2016, August 8). How To Succeed at Failure with Microservices. Abgerufen von https://thenewstack.io/succeed-failure-microservices/

Brereton, P., & Budgen, D. (2000). Component-based systems: A classification of issues. *Computer*, *33*(11), 54–62.

Brewer, E. A. (2000). Towards robust distributed systems. In *PODC* (Bd. 7). Abgerufen von http://awoc.wolski.fi/dlib/big-data/Brewer_podc_keynote_2000.pdf

Brüsemeister, T. (2008). *Qualitative Forschung: ein Überblick* (2., überarb. Aufl.). Wiesbaden: VS, Verl. für Sozialwiss.

Büst, R. (2016, April 14). IaaS vs. PaaS vs. Container: Moderne Anwendungsentwicklung in der Cloud. Abgerufen 17. März 2017, von https://www.crisp-research.com/iaas-vs-paas-vs-container-moderne-anwendungsentwicklung-der-cloud/

Byars, B. (2013, November 18). Enterprise Integration Using REST. Abgerufen 11. September 2016, von http://martinfowler.com/articles/enterpriseREST.html

Chu, S. C. (2005). From component-based to service oriented software architecture for healthcare. In *Proceedings of 7th International Workshop on Enterprise networking and Computing in Healthcare Industry, 2005. HEALTHCOM 2005.* (S. 96–100). doi:10.1109/HEALTH.2005.1500402

Ciancutti, J. (2010, Dezember 14). Four Reasons We Choose Amazon's Cloud as Our Computing Platform. Abgerufen von http://techblog.netflix.com/2010/12/four-reasons-we-choose-amazons-cloud-as.html

Clements, P., Bachmann, F., Bass, L., Garlan, D., Ivers, J., Little, R., … Stafford, J. (2010). *Documenting Software Architectures: Views and Beyond*. Pearson Education.

Demeyer, S., Ducasse, S., & Nierstrasz, O. (2009). *Object-oriented reengineering patterns: version of 2009-09-28*. Square Bracket Ass.

Dreifus, F., Leyking, K., & Loos, P. D. P. (2007). Systematisierung der Nutzenpotentiale einer SOA. In V. Nissen, M. Petsch, & H. Schorcht (Hrsg.), *Service-orientierte Architekturen* (S. 19–38). Gabler. doi:10.1007/978-3-8349-9636-7_2

Edlich, S., Friedland, A., Hampe, J., Brauer, B., & Brückner, M. (2011). *NoSQL: Einstieg in die Welt nichtrelationaler Web 2.0 Datenbanken* (2. Aufl.). München: Carl Hanser Verlag GmbH & Co. KG. doi:10.3139/9783446428553

Erl, T. (2007). *SOA: Principles of Service Design* (1st edition.). Upper Saddle River, NJ: Prentice Hall.

Evans, E. (2004). *Domain-Driven Design: Tackling Complexity in the Heart of Software*. Addison-Wesley Professional. Abgerufen von http://proquest.techbus.safaribooksonline.de/book/project-management/0321125215

Evans, E. (2013). Getting started with DDD when surrounded by legacy systems. Domain Language, Inc. Abgerufen von http://domainlanguage.com/ddd/surrounded-by-legacy-software/

Fairbanks, G. (2010). *Just enough software architecture: a risk-driven approach*. Boulder, Colo: Marshall & Brainerd.

Feathers, M. (2014, Juli 3). Microservices Until Macro Complexity. Abgerufen von https://michaelfeathers.silvrback.com/microservices-until-macro-complexity

Feathers, M., & Martin, R. C. (2005). *Working effectively with legacy code*. Upper Saddle River, NJ: Prentice Hall Professional Technical Reference.

Finnegan, M. (2016, September 27). Volkswagen picks Pivotal over HPE for platform as a service cloud. Abgerufen von http://www.computerworlduk.com/cloud-computing/volkswagen-continues-open-source-cloud-push-with-cloud-foundry-paas-3646928/

Flohre, T. (2015, November 9). Kommunikation von Microservices - Die vier Ebenen der Entkopplung. Abgerufen von https://blog.codecentric.de/2015/11/kommunikation-von-microservices-die-vier-ebenen-der-entkopplung/

Fowler, M. (2002). *Patterns of Enterprise Application Architecture*. Addison-Wesley Professional. Abgerufen von http://proquest.techbus.safaribooksonline.de/book/software-engineering-and-development/patterns/0321127420

Fowler, M. (2004a, Jänner 23). Inversion of Control Containers and the Dependency Injection pattern. Abgerufen 3. Oktober 2016, von http://martinfowler.com/articles/injection.html

Fowler, M. (2004b, Juni 29). StranglerApplication. Abgerufen von https://martinfowler.com/bliki/StranglerApplication.html

Fowler, M. (2014, Juli 1). BranchByAbstraction. Abgerufen von https://martinfowler.com/bliki/BranchByAbstraction.html

Fowler, M. (2015, Mai 13). MicroservicePremium. Abgerufen von https://martinfowler.com/bliki/MicroservicePremium.html

Fowler, M., & Lewis, J. (2015). Microservices: Nur ein weiteres Konzept in der Softwarearchitektur oder mehr. *OBJEKTspektrum*, (1/2015), 14–20.

Freudl-Gierke, T. (2014, November 3). Micro Services in der Praxis: Nie wieder Monolithen! Abgerufen von https://jaxenter.de/micro-services-in-der-praxis-nie-wieder-monolithen-391

Fulton III, S. M. (2015, Oktober 8). What Led Amazon to its Own Microservices Architecture. Abgerufen von https://thenewstack.io/led-amazon-microservices-architecture/

Gimnich, R., & Winter, A. (2005). Workflows der Software-Migration. *Softwaretechnik-Trends*, *25*(2), 22–24.

Goliath, A. (2016). Enterprise IT - Warum Microservices in großen Unternehmen mehr als nur Software sind. *iX Developer*, *Effektiver entwickeln*(2/2016). Abgerufen von http://d-nb.info/1098428161

Hammant, P. (2013, Juli). Legacy Application Strangulation: Case Studies. Abgerufen von http://paulhammant.com/2013/07/14/legacy-application-strangulation-case-studies/

Hoff, T. (2007, September 18). Amazon Architecture. Abgerufen von http://highscalability.com/amazon-architecture

Hughes, J. (2013, April 29). Micro Service Architecture. Abgerufen von https://yobriefca.se/blog/2013/04/29/micro-service-architecture/

Iskold, A. (2006, August 17). Amazon - The Real Web Services Company. Abgerufen von http://soa.sys-con.com/node/262024

Izrailevsky, Y., & Tseitlin, A. (2011, Juli 19). The Netflix Simian Army. Abgerufen von http://techblog.netflix.com/2011/07/netflix-simian-army.html

Jadeja, Y., & Modi, K. (2012). Cloud computing - concepts, architecture and challenges. In *2012 International Conference on Computing, Electronics and Electrical Technologies (ICCEET)* (S. 877–880). doi:10.1109/ICCEET.2012.6203873

Joel, S. (2000, April 6). Things You Should Never Do, Part I. Abgerufen 27. Jänner 2017, von https://www.joelonsoftware.com/2000/04/06/things-you-should-never-do-part-i/

Josuttis, N. M. (2007). *SOA in Practice: The Art of Distributed System Design* (1. edition.). Beijing; Sebastopol: O'Reilly Media.

Kaps, S. (2016). *iX Developer - Altlasten im Griff*. (iX-Redaktion, Hrsg.). Hannover: Heise Medien. Abgerufen von http://public.eblib.com/choice/PublicFullRecord.aspx?p=4753265

Landwehr, A., & Kraus, H. (2016, August 23). *Legacy-Apps mit Microservices, Hystrix und RxJava modernisieren*. W-JAX 2015. Abgerufen von https://jaxenter.de/microservices-hystrix-rxjava-45442

Lehman, M. M. (1980). Programs, life cycles, and laws of software evolution. *Proceedings of the IEEE*, *68*(9), 1060–1076. doi:10.1109/PROC.1980.11805

Martincevic, N. (2016, November 22). DDD: Context is King – Kein Context, keine Microservices. Abgerufen von https://www.informatik-aktuell.de/entwicklung/methoden/ddd-context-is-king-kein-context-keine-microservices.html

Masak, D. (2006). *Legacysoftware - Das lange Leben der Altsysteme*. Berlin/Heidelberg: Springer-Verlag. doi:10.1007/3-540-30320-0

Mauro, T. (2015, März 10). Microservices at Netflix: Lessons for Team and Process Design. Abgerufen von https://www.nginx.com/blog/adopting-microservices-at-netflix-lessons-for-team-and-process-design/

Namiot, D., & Sneps-Sneppe, M. (2014). On micro-services architecture. *International Journal of Open Information Technologies*, *2*(9). Abgerufen von http://cyberleninka.ru/article/n/on-micro-services-architecture

Newman, S. (2015). *Building Microservices* (1. Aufl.). Beijing Sebastopol, CA: O'Reilly and Associates. Abgerufen von http://shop.oreilly.com/product/0636920033158.do

Nygard, M. T. (2007). *Release it!: design and deploy production-ready software*. Raleigh, N.C: Pragmatic Bookshelf.

Overhage, D. S., & Turowski, P. D. K. (2007). Serviceorientierte Architekturen — Konzept und methodische Herausforderungen. In V. Nissen, M. Petsch, & H. Schorcht (Hrsg.), *Service-orientierte Architekturen* (S. 3–17). Gabler. doi:10.1007/978-3-8349-9636-7_1

Parulekar, S., & Pilani, R. (2016, Juli 20). Netflix Billing Migration to AWS - Part II. Abgerufen von http://techblog.netflix.com/2016/07/netflix-billing-migration-to-aws-part-ii.html

Persa, D. (2015, Oktober 16). From Jimmy to Microservices: Rebuilding Zalando's Fashion Store. Abgerufen von https://tech.zalando.com/blog/from-jimmy-to-microservices-rebuilding-zalandos-fashion-store/

Porst, R. (2014). *Fragebogen: Ein Arbeitsbuch*. Wiesbaden: Springer Fachmedien Wiesbaden. doi:10.1007/978-3-658-02118-4

Posta, C. (2016, Juli 14). The Hardest Part About Microservices: Your Data. Abgerufen von http://blog.christianposta.com/microservices/the-hardest-part-about-microservices-data/

Pronschinske, M. (2015, März 19). Key Takeaways: Adrian Cockcroft's talk on Netflix, CD, and Microservices. Abgerufen von https://dzone.com/articles/key-takeaways-adrian-cockrofts

Raithel, J. (2008). *Quantitative Forschung: ein Praxiskurs* (2., durchgesehene Auflage.). Wiesbaden: VS Verlag für Sozialwissenschaften.

Richards, M. (2016). *Microservices vs. Service-Oriented Architecture*. O'Reilly Media, Inc. Abgerufen von https://www.safaribooksonline.com/library/view/microservices-vs-service-oriented/9781491975657/

Richardson, C. (2016, März 24). Who is using microservices? Abgerufen 19. Februar 2017, von http://microservices.io/articles/whoisusingmicroservices.html

Richardson, C., & Smith, F. (2016). *Microservices: From Design to Deployment*. Abgerufen von https://www.nginx.com/resources/library/designing-deploying-microservices/

Rotem-Gal-Oz, A. (2014, März 25). Services, Microservices, Nanoservices - oh my! Abgerufen von http://arnon.me/2014/03/services-microservices-nanoservices/

Röwekamp, L., & Limburg, A. (2016). Patterns - Der perfekte Microservice. *iX Developer*, *Effektiver entwickeln*(2/2016). Abgerufen von http://d-nb.info/1098428161

Sangeeta, H. (2016, Juni 21). Netflix Billing Migration to AWS. Abgerufen von http://techblog.netflix.com/2016/06/netflix-billing-migration-to-aws.html

Santis, S. D., Florez, L., Nguyen, D. V., & Rosa, E. (2016). *Evolve the Monolith to Microservices with Java and Node*. IBM Redbooks.

Schmelzer, H. J., & Sesselmann, W. (2013). *Geschäftsprozessmanagement in der Praxis: Kunden zufrieden stellen - Produktivität steigern - Wert erhöhen* (Auflage: 8., überarbeitete und erweiterte Auflage.). München: Carl Hanser Verlag GmbH & Co. KG.

Smith, S. (2013, Oktober 14). Application Pattern: Verify Branch By Abstraction. Abgerufen von http://www.alwaysagileconsulting.com/articles/application-pattern-verify-branch-by-abstraction/

Sneed, H. M. (2003). Aufwandsschätzung von Software-Reengineering-Projekten. *Wirtschaftsinformatik*, *45*(6), 599–610.

Sneed, H. M. (2012). Nachhaltigkeit durch gesteuerte Software-Evolution. In *Nachhaltiges Software Management* (S. 50–68). Abgerufen von https://pdfs.semanticscholar.org/b7cc/dbf0719fc7d1c14d277d1636c44a241b0dfd.pdf

Sneed, H. M., & Sneed, S. H. (2003). *Web-basierte Systemintegration*. Wiesbaden: Vieweg+Teubner Verlag. Abgerufen von http://link.springer.com/10.1007/978-3-322-89822-7

Sommerville, I. (2012). *Software Engineering* (9., aktualisierte Auflage.). München Harlow Amsterdam: Pearson.

Srivastava, D. (2017, Februar 14). Introducing Cloud Spanner: a global database service for mission-critical applications. Abgerufen von https://cloudplatform.googleblog.com/2017/02/introducing-Cloud-Spanner-a-global-database-service-for-mission-critical-applications.html

Starke, G. (2015). *Effektive Softwarearchitekturen: Ein praktischer Leitfaden* (7. Aufl.). München: Carl Hanser Verlag GmbH & Co. KG.

Starke, G. (2016, Oktober 17). Software systematisch verbessern. Abgerufen von https://www.innoq.com/de/articles/2016/10/software-verbessern-aim42/

Starke, G. (2017, Jänner 23). Architecture Improvement Method. Abgerufen 10. Februar 2017, von http://aim42.github.io/

Steinacker, G. (2015, Februar 6). Von Monolithen und Microservices. Abgerufen 29. August 2016, von https://www.informatik-aktuell.de/entwicklung/methoden/von-monolithen-und-microservices.html

Stenberg, J. (2015, Februar 28). Characteristics of Microservices, Applications and Systems. Abgerufen 14. September 2016, von https://www.infoq.com/news/2015/02/characteristics-microservices-ap

ten Hompel, M., & Schmidt, T. (Hrsg.). (2008). Softwareengineering. In *Warehouse Management: Organisation und Steuerung von Lager- und Kommissioniersystemen* (S. 221–254). Berlin, Heidelberg: Springer Berlin Heidelberg. Abgerufen von http://dx.doi.org/10.1007/978-3-540-74876-2_6

Tilkov, S. (2014, Dezember 22). *Nano, Micro, Mini, oh my: Modularization for sustainable systems*. Abgerufen von https://www.youtube.com/watch?v=HYiLzji7MuY

Tilkov, S., & Wolff, E. (2016). Architekturkonzept - Microservices: Modularisierung ohne Middleware. *iX Developer*, *Effektiver entwickeln*(2/2016). Abgerufen von http://d-nb.info/1098428161

Urban, S., Sreenivasan, R., & Kannan, V. (2016, April 29). It's All A/Bout Testing: The Netflix Experimentation Platform. Abgerufen von http://techblog.netflix.com/2016/04/its-all-about-testing-netflix.html

Vernon, V. (2016). *Domain-Driven Design Distilled* (1. Aufl.). Addison-Wesley Professional.

Villamizar, M., Garcés, O., Castro, H., Verano, M., Salamanca, L., Casallas, R., & Gil, S. (2015). Evaluating the monolithic and the microservice architecture pattern to deploy web applications in the cloud. In *Computing Colombian Conference (10CCC), 2015 10th* (S. 583–590). IEEE. Abgerufen von http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=7333476

Vogel, O., Arnold, I., Chughtai, A., Ihler, E., Kehrer, T., Mehlig, U., & Zdun, U. (2009). *Software-Architektur*. Heidelberg: Spektrum Akademischer Verlag. Abgerufen von http://link.springer.com/10.1007/978-3-8274-2267-5

Wolff, E. (2006, Oktober 2). JAOO 2006: Werner Vogels - CTO Amazon. Abgerufen von http://jandiandme.blogspot.co.at/2006/10/jaoo-2006-werner-vogels-cto-amazon.html

Wolff, E. (2015). *Microservices: Grundlagen flexibler Softwarearchitekturen*. Dpunkt.Verlag GmbH.

Wolff, E. (2017, Februar 25). Was können wir von Google, Amazon und Netflix lernen? Abgerufen von http://www.heise.de/developer/artikel/Was-koennen-wir-von-Google-Amazon-und-Netflix-lernen-3634647.html

Züll, C., & Mohler, P. P. (2001). Computerunterstützte Inhaltsanalyse: Codierung und Analyse von Antworten auf offene Fragen. Abgerufen von http://www.ssoar.info/ssoar/handle/document/20140