Fakultät für
Informations-, Medien-
und Elektrotechnik

Technology
Arts Sciences
TH Köln

Masterarbeit

Eine Heuristik zur Eignungsprüfung von Microservice-Architekturen

Name: *Kamil Szuster*

Email: *ka.szuster@gmail.com*
Matrikelnummer: *11106921*
Studiengang: *Informatik/Computer Science (Master)*

Erstprüfer: *Prof. Dr. Stefan Bente*

stefan.bente@th-koeln.de
Technology, Arts and Sciences TH Köln

Zweitprüfer: *Dr. Thomas Franz*

thomas.franz@adesso.de
adesso AG

Abgabedatum: 21.12.2016

Zusammenfassung

Die vorliegende Arbeit wurde im Rahmen der Masterarbeit an der Technischen Hochschule Köln - Campus Gummersbach als Teil des Studiengangs Informatik/Computer Science (Master) mit dem Schwerpunkt Software Engineering in Zusammenarbeit mit der adesso AG verfasst.

Diese Arbeit behandelt die Frage, ob die Einführung des Softwarearchitektur-Paradigmas Microservices in dem jeweiligen Projektkontext sinnvoll ist oder nicht. In diesem Zusammenhang warnen viele Experten in Fachbeiträgen[1,2] vor einer überhasteten Adaption von Microservices. Trotz der zahlreichen Publikationen zu diesem Themenkomplex, die sich sowohl mit den Vor- und Nachteilen als auch mit Praxisbeispielen beschäftigen, stellt die Entscheidungsfindung offenbar noch immer ein Problem dar.

Um diesen Entscheidungsprozess zu unterstützen, wurden im Rahmen dieser Arbeit namhafte Experten zur Entscheidung, Planung und Durchführung von Microservice-Projekten befragt. Ausgehend von den Projekterfahrungen der Experten und einer etablierten Methode zur Evaluierung von Softwarearchitekturen (ATAM), wurde ein Wirkungsmodell entwickelt, das eine Architekturentscheidung aus der Sicht der Eignung sowie der Effizienz betrachtet. Dieses abstrakte Modell wurde mit microservicespezifischen Mehrwerten, Komplexitäten und deren Beziehungen zueinander angereichert, diese wurden verschiedenen Veröffentlichungen abgeleitet.

Das entwickelte Modell stellt die Basis für ein systematisches Aufarbeiten der zentralen Fragestellung dieser Arbeit dar. Mit dem Hintergrund der Architekturziele können die Mehrwerte und Komplexitäten gegeneinander abgewogen werden. Kompromisse und Projektsituation helfen dabei, die passende Lösung für den jeweiligen Kontext zu finden. Auf Basis all dieser Informationen kann letztendlich eine fundierte Entscheidung getroffen werden.

[1] vgl. Wolff, „Microservices: Weg vom Hype - rein in die Praxis!“, S. 3.
[2] siehe beispielsweise *Technology Radar - Microservice envy*.

Inhaltsverzeichnis

Abbildungsverzeichnis

Abkürzungsverzeichnis

API Application Programming Interface.

ATAM Architecture Tradeoff Analysis Method.

CD Continuous-Delivery.

CI Continuous-Integration.

EJB Enterprise JavaBean.

ESB Enterprise Service Bus.

FMEA Failure Mode and Effects Analysis.

GUI Graphical User Interface.

IDE integrated bla.

IIS Internet Information Services.

LOC Lines Of Code.

PaaS Platform as a Service.

REST Representational State Transfer.

SAAM Software Architecture Analysis Method.

SBA Service-based Architecture.

SCM Source-Control-Management.

SCS Self-contained System.

SOA Service-oriented Architecture.

UI User Interface.

Glossar

Continuous-Delivery (CD) bezeichnet eine Sammlung von Techniken, Prozessen und Werkzeugen, die den Softwareauslieferungsprozess verbessern sollen.

CRUD ist ein Akronym. Es umfasst die grundlegenden Datenbankoperationen Create, Read, Update und Delete.

DevOps beschreibt unter anderem den Zusammenschluss von Entwickler (Development) und Betrieb (Operations) zu einer Einheit mit dem Ziel, Prozesse zu verbessern.

Enterprise Service Bus (ESB) wird zur Integration von verteilter Dienste genutzt.

Erosion beschreibt den langsamen Verfall einer Softwarearchitektur.

Kohäsion In einem System mit starker Kohäsion ist jede Programmeinheit verantwortlich für genau eine wohldefinierte Aufgabe oder Einheit.

Middleware sind anwendungsneutrale Systeme, die zwischen Anwendungen vermitteln.

NoSQL bezeichnet Datenbanken, die einen nicht-relationalen Ansatz verfolgen.

Relationale Datenbank ist eine Datenbank mit einem tabellenbasierten Datenbankmodell.

Service-Discovery dient der automatischen Erkennung von Services.

Source-Control-Management (SCM) ist ein System, das zur Erfassung von Änderungen an Dokumenten oder Dateien verwendet wird.

Stateful Anwendungen beschreibt eine Zustandshaltende Anwendung, die Informationen über den eigenen Zustand zu speichern.

Timeout bezeichnet die Zeitspanne, die ein Vorgang in Anspruch nehmen darf, bevor er mit einem Fehler abgebrochen wird.

Kapitel 1

Einführung

1.1 Ausgangslage

Marc Andreessen, Mitgründer von Netscape Communications und bedeutender Investor im Internet-Startup Sektor, veröffentlichte den Artikel "Why Software is eating the world" im Wall Street Journal. Er beschreibt wie komplette Märkte mittels Software nachhaltig verändert werden. Diese These hat seit ihrer Veröffentlichung im Jahr 2011 nicht an Bedeutung verloren, im Gegenteil. Software und technologische Innovation wird immer wichtiger, um als Unternehmen bestehen zu können. In immer kürzeren Abständen erscheinen Unternehmen auf der Bildfläche, die mittels Software/Technologie bestehende Produkte und oftmals die damit verbundenen Unternehmen vom Markt drängen. Beispiele für solche disruptive Technologien/Innovationen[1] gibt es viele:

- das Auto verdrängte die Pferdekutsche
- analoge Kameras wurden von digitalen verdrängt
- Amazon beansprucht sukzessiv Anteile aus verschiedenen Märkten
- Dienste wie mytaxi oder Uber verändern die Personenbeförderung
- Streaming-Plattformen wie Netflix gewinnen immer mehr an Bedeutung

Das gleiche Phänomen kann ebenfalls in Märkten auftreten, in denen wir es uns heute noch nicht vorstellen können. Beispielsweise könnten Unternehmen wie Google oder Facebook den

[1] vgl. Christensen, *The Innovator's Dilemma: When New Technologies Cause Great Firms to Fail*.

Bankensektor innerhalb von kürzester Zeit entscheidend verändern.[2] Eine Disruption muss jedoch nicht immer so extrem sein. Es ist durchaus denkbar, dass ein Konkurrent eine neue Plattform oder ein Produkt schafft, das eine kontinuierliche Kundenabwanderung zur Folge hat.

Die klassische Industrie muss Maßnahmen ergreifen, um solchen Ereignissen nicht zum Opfer zu fallen. Das Ziel sollte sein, möglichst schnell auf Änderungen am Markt reagieren zu können und eine enge Kundenbindung zu erzeugen.[3] Bereits etablierte Unternehmen betreiben Anwendungen, die historisch gewachsen und für das Kerngeschäft kritisch sind. Diese Legacy Systeme werden den beschriebenen Anforderungen jedoch oft nicht gerecht.

Zur Lösung dieses Problems haben die Boston Consulting Group im August 2012[4] und McKinsey im Dezember 2014[5] Artikel veröffentlicht, in denen das Prinzip der "Two Speed IT" vorgestellt wird. Vor dem Hintergrund der digitalen Transformation soll sich die IT eines Unternehmens in zwei verschiedenen Geschwindigkeiten entwickeln. Auf der einen Seite soll zentrale, historisch gewachsene Software, die wichtige Daten enthält in einem langsamen, qualitativ hochwertigen, wasserfallartigem Prozess entwickelt werden. Auf der anderen Seite steht die agile IT, die Projekte autark und in kurzen Entwicklungszyklen abwickelt. Dieses Vorgehen stößt jedoch auf viel Kritik.[6,7]

Es stellt sich also die Frage: Wie versetzt man mittelfristig Unternehmen und Software in die Lage, sehr schnell auf den Markt reagieren zu können, langfristig wartbar zu sein und den Qualitätskriterien zu entsprechen? Microservices begegnen diesem Problem auf architektonischer Ebene. Sie versprechen unter anderem effizienten Umgang mit Legacy Systemen, erhöhte Reaktionsfähigkeit, Skalierbarkeit auf technischer und organisatorischer Ebene sowie eine nachhaltige Software-Entwicklung.

[2] vgl. Nestler, *Googles Weg zur Sparkasse*.
[3] vgl. Treacy und Wiersema, *Customer Intimacy and Other Value Disciplines*.
[4] Gourévitch u. a., *Two-Speed IT: A Linchpin for Success in a Digitized World*, vgl.
[5] vgl. Bossert u. a., *A two-speed IT architecture for the digital enterprise*.
[6] vgl. Nathe, *Warum die IT Sportwagen und LKW zugleich sein muss*.
[7] vgl. Hruschka und Starke, *Knigge für Softwarearchitekten: Wider die IT der zwei Geschwindigkeiten*.

1.2 Zielsetzung

Aufgrund der einleitenden Ausgangslage und dem sehr großen Interesse am Microserviceparadigma, das man an der Vielzahl an publizierten Artikeln, Konferenzvorträgen und veröffentlichten Frameworks beobachten kann, stehen viele Unternehmen vor der Frage: "Macht die Einführung von Microservices in unserem Projektkontext Sinn?" Laut Experten wie Eberhard Wolff und Stefan Toth sollte Softwarearchitektur kein Selbstzweck sein. Bei jeder Architekturentscheidung müssen Vorteile, Nachteile und Alternativen gegeneinander abgewogen werden.[8]

Diese Arbeit hat zum Ziel, genau diesen Entscheidungsprozess zu unterstützen. Es wird ein Vorgehen entwickelt, das bei der Entscheidungsfindung bezüglich der Einführung oder Migration von Microservices im Einsatzkontext unterstützt. Das Resultat ist ein breites Verständnis der eigenen Projektziele, wie eine Microservice-Architektur diese Ziele unterstützten kann, mit welchen Herausforderungen zu rechnen ist, welche Kompromisse eingegangen werden können und wie sich die aktuelle Projektsituation auf diese Entscheidung auswirkt. Letztendlich kann auf Basis der gewonnenen Erkenntnisse eine fundierte Entscheidung getroffen werden.

1.3 Lösungsansatz

Die zentrale Fragestellung dieser Arbeit ist, ob die Einführung von Microservices innerhalb eines bestimmten Projektkontextes sinnvoll ist. Da es sich hierbei um ein Softwarearchitekturparadigma handelt, muss zunächst der Begriff Softwarearchitektur geklärt werden.

Martin Folwer beschreibt Softwarearchitektur folgendermaßen: "To me the term architecture conveys a notion of the core elements of the system, the pieces that are difficult to change. A foundation on which the rest must be built."[9] Es handelt sich dabei also um die Kernstücke von Software, auf denen alles andere basiert. Die Schwierigkeit der Änderbarkeit definiert sich über Änderungen, die das Erreichen der Qualitätsziele (oder auch *nicht-funktionale Anforderungen*

[8] vgl. Wolff, „Microservices: Weg vom Hype - rein in die Praxis!“, S. 3.
[9] Fowler, *Is Design Dead?*

genannt) gefährden, teuer oder aufwendig sind. Aus diesem Grund sind Architekturentscheidungen von zentraler Bedeutung für den Projekterfolg.[10] Sie sollten somit immer auf Basis der definierten Ziele und dem Hintergrund der Wirtschaftlichkeit getroffen werden.

Um die zentrale Frage zu beantworten, werden in dem entwickelten Lösungsansatz zwei Subfragen untersucht. "Wie wirken sich Microservices auf die definierten Projekt-/Architekturziele aus?" - fokussiert auf den Mehrwert, den eine Microservice-Architektur verspricht und wie sich diese auf die Ziele auswirken. Die zweite Subfrage - "Ist diese Lösung kosteneffizient?" - beleuchtet die Kosten in Form von *Komplexitäten*, die zur Realisierung der *Mehrwerte* bezahlt werden. *Kompromisse* und die aktuelle *Projektsituation* können diese Komplexitäten beeinflussen. Dieser Einfluss hat jedoch wiederum Auswirkungen auf den Mehrwert.

Bei einer guten Architekturentscheidung überwiegen die Vorteile die Nachteile unter Berücksichtigung der Wirtschaftlichkeit. Abbildung 1.1 beschreibt das abstrakte Wirkungsmodell. Es stellt Beziehungen zwischen *Architekturzielen*, *Mehrwerten*, *Komplexitäten* und *Kompromissen* her. Auf Basis dieses ganzheitlichen Bildes, kann schlussendlich eine fundierte Architekturentscheidung getroffen werden.

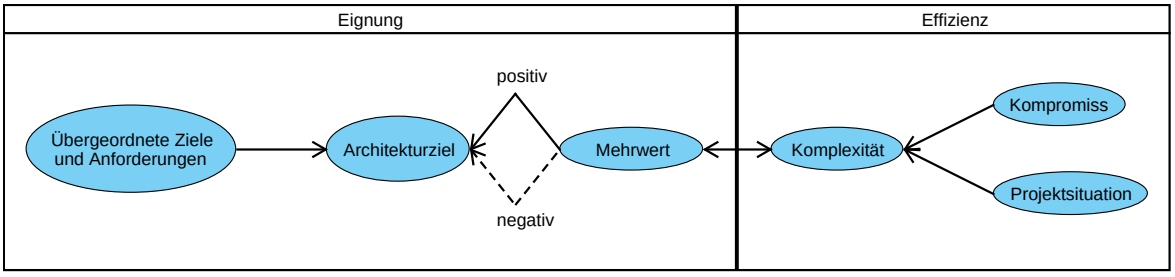

ABBILDUNG 1.1: Das entwickelte Wirkungsmodell einer Architekturentscheidung

1.4 Abgrenzung

Diese Arbeit behandelt Microservices auf einem hohen Abstraktionsgrad. Technische und organisatorische Umsetzung werden nicht behandelt. Es ist zu beachten, dass mit jeder technischen Implementierung und organisatorischen Maßnahme unterschiedliche Kompromisse eingegangen werden. Diese haben positive oder auch negative Auswirken auf den Projekterfolg.

[10] vgl. Toth, *Vorgehensmuster für Softwarearchitektur*, S. 3.

Einflüsse, die z.B. durch technologische Entscheidungen wirken, werden in dieser Arbeit nicht behandelt.

1.5 Aufbau der Arbeit

Eingangs wird in Kapitel 2 "Microservices" das Architekturparadigma Microservices aus verschiedenen Perspektiven beleuchtet. Dafür wird der Stand der Praxis gemäß der Fachliteratur wiedergegeben. Es werden zunächst verschiedene Definitionsversuche verglichen, anschließend erfolgt eine Beschreibung aus konzeptioneller, technischer und organisatorischer Sicht. Danach werden unterschiedlicher Varianten dieses Paradigmas präsentiert. Zum Schluss folgt eine Analyse der Vorteile und Herausforderungen.

In Kapitel 3 "Experteninterviews" wird die Planung und Durchführung von Experteninterviews, mit dem Ziel der Theorieentwicklung, beschrieben. Die ausgewählten Experten wurden zu ihren Projekterfahrungen mit Bezug zu Microservices anhand eines Leitfadens befragt. Die Gemeinsamkeiten in den Vorgehen der Experten ergeben ein Muster nach dem Architekturentscheidungen getroffen werden können.

In dem darauf folgenden Kapitel 4 "Methoden" werden analytische Methoden und Prozesse zur Risiko- und Architekturbewertung untersucht. Sie dienen als methodische Basis, die zur Entwicklung der hier vorgestellten Heuristik dient.

Anschließend wird die entwickelte Heuristik auf drei Abstraktionsstufen vorgestellt. Abschnitt 5.1 beschreibt das sogenannte allgemeine Wirkungsmodell. Es befindet sich auf der höchsten Abstraktionsebene und führt die Einflussfaktoren einer Architekturentscheidung und deren Wirkungsbeziehungen ein.

Das spezifische Wirkungsmodell wird in Abschnitt 5.2 vorgestellt. Es beschreibt mikroservicespezifische Mehrwerte, Komplexitäten und Architekturziele. Bei einem Modell handelt es sich um *eine* der möglichen Ausprägung des allgemeinen Wirkungsmodells. Es werden vier Ausprägungen aufgeführt.

Auf der letzten Abstraktionsebene befinden sich die detaillierten Wirkungsmechanismen. Sie werden in Abschnitt 5.3 erörtert und fokussieren vor allem auf mögliche Kompromisse und auf die Zusammenhängen aus denen sich Mehrwerte sowie Komplexitäten ergeben.

In Abschnitt 5.4 "Praktische Anwendung" wird beschrieben, wie die Heuristik an einem fiktiven Projekt angewandt wird.

Die Arbeit wird mit 6 "Fazit und Ausblick" abgeschlossen.

Kapitel 2

Microservices

Auf Fachkonferenzen ist kaum an gut besuchten Vorträgen zum Thema Microservices vorbei zu kommen und Fachzeitschriften, besonders im Java Umfeld, veröffentlichen regelmäßig Artikel rund um diesen Themenkomplex. In den Google Trends gewinnt dieses Software Architektur Paradigma seit 2014 kontinuierlich an Bedeutung, war vorher jedoch kaum von Belang. Abbildung 2.1 zeigt das relative Interesse an dem Stichwort "Microservice" von Januar 2013 bis November 2016. Die Y-Achse beschreibt die Anzahl der Google-Suchanfragen, relativ zum erfassten Höchstwert.

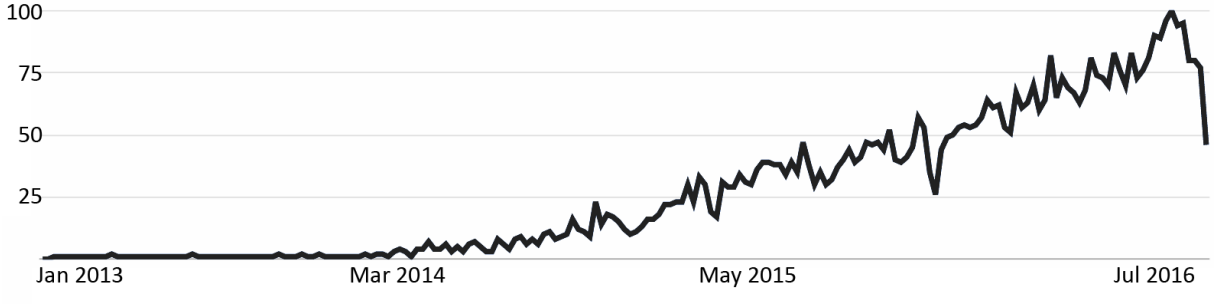

ABBILDUNG 2.1: Google Trend zu Microservice Quelle: Google Trends, *Websuche-Interesse: microservices - Weltweit, Jan. 2013 - Nov. 2016*

2.1 Definition

Trotz des hohen Interesses an Microservices gibt es bislang keine eindeutige Definition. Der wohl meist zitierte Definitionsversuch stammt von Martin Fowler und James Lewis. Sie untersuchten erfolgreich durchgeführte Projekte, die auf einer Microservice-Architektur basieren und leiteten die folgende Definition den gemeinsamen Merkmalen ab.

"The microservice architectural style is an approach to developing a single application as a suite of small services, each running in its own process and communicating with lightweight mechanisms, often an HTTP resource API. These services are built around business capabilities and independently deployable by fully automated deployment machinery. There is a bare minimum of centralized management of these services, which may be written in different programming languages and use different data storage technologies." (James Lewis, Martin Fowler)[1]

Nach Sam Newman sind Microservices "... kleine, eigenständige Services, die kollaborieren bzw. sich gegenseitig zuarbeiten ..."[2]. Für Ihn sind hochgradige Geschlossenheit und lose Kopplung die wichtigsten Attribute einer Microservice-Architektur.[3]

Eberhard Wolff beschreibt Microservice in seinen Talks immer wieder als "... einzelne Services, die unabhängig voneinander deployt werden können ..." oder als "... ein Ansatz zur Modularisierung von Software ..."[4]. Diese Definition hat einige Implikation, die er in seinem Buch weiter ausführt.[5] Die von ihm beschriebenen Aspekte finden sich ebenfalls in den Attributen der hochgradigen Geschlossenheit und der losen Kopplung wieder.

Es ist festzustellen, dass alle Definitionen Microservices als **kleine, hochgradig geschlossene und lose gekoppelte Einheiten** beschreiben.

[1] Fowler, *Microservices Guide*.
[2] Newman, *Microservices (mitp Professional): Konzeption und Design*, S. 22.
[3] vgl. ebd., S. 56.
[4] Wolff, *Microservices: Grundlagen flexibler Softwarearchitekturen*, S. 2.
[5] vgl. ebd., S. 31 - 57.

2.2 Konzept

Um das Konzept von Microservices zu verstehen, ist der Vergleich mit einem Deployment-Monolith hilfreich. Ein Deployment-Monolith ist eine Applikation, die als eine Einheit gebaut wird. Diese Einheit realisiert meist fachliche Anforderungen unterschiedlicher Geschäftsbereiche (Business Domain) (siehe Kapitel 2.3 "Business Domain"). Am Beispiel eines E-Commerce-Systems könnten diese Einkaufswagen, Bestellprozess, Benutzerverwaltung und Katalog sein.

Eine klassische Drei-Schichten-Architektur setzt sich aus den folgenden Schichten zusammen.

- Benutzerschnittstelle - Oftmals ein Graphical User Interface (GUI), bestehend aus HTML Seiten, die die Interaktion von Benutzer und System unterstützen
- Datenbank - Ein zentraler Datenspeicher, der das Persistieren von Informationen ermöglicht. Die Speicherung erfolgt oftmals in einer relationalen Datenbank.
- Serverseitige Anwendung - Diese Komponente verbindet die Benutzerschnittstelle mit der Datenbank. Sie nimmt beispielsweise HTTP Anfragen an, führt Datenbankoperationen sowie Geschäftslogik aus und erzeugt eine HTML Seite, die als Antwort an den Benutzer geschickt wird.[6]

Mittels Features wie Modulen, Klassen, Funktionen und Packages der gewählten Programmiersprache, kann die Anwendung strukturiert werden. Dennoch wird sie in einem Prozess ausgeführt. Jegliche Änderungen an einer technischen Schicht oder einer fachlichen Anforderung haben zur Folge, dass eine neue Version des gesamten Systems gebaut und ausgeliefert werden muss. Dieser Prozess kann aus vielen automatisierten oder manuellen Teilschritten bestehen.[7]

Beim Microservice-Ansatz wird der Deployment-Monolith in seine fachlichen Einzelteile zerlegt. Dies geschieht, indem die verschiedenen Business Domains als eigenständige Services implementiert werden. Fowler definiert Services als Komponenten, die in eigenständigen Prozessen ausgeführt werden, Mechanismen wie *Web Service Requests* zur Kommunikation nutzen und unabhängig voneinander in Betrieb genommen werden können.[8]

[6] vgl. Fowler, *Microservices*.
[7] vgl. Wolff, *Microservices: Grundlagen flexibler Softwarearchitekturen*, S. 3.
[8] vgl. Fowler, *Microservices*.

Abbildung 2.2 veranschaulicht die Aufspaltung von einem Deployment-Monolith nach fachlichen Aspekten in Business Domain spezifische Microservices, die in eigenen Prozessen ausgeführt werden.

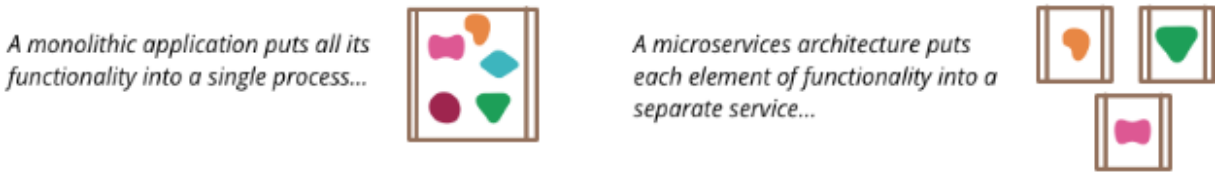

ABBILDUNG 2.2: Konzeptioneller Vergleich eines Deployment-Monolithen und Microservices aus Prozesssicht (Quelle: Fowler, *Microservices*)

Technisch können die Microservices weiterhin in die zuvor beschriebenen Schichten unterteilt sein, jedoch liegt der Fokus auf der Zerlegung des Systems auf einzelne Business Domains. Abbildung 2.3 vergleicht den technischen Aufbau von einem Deployment-Monolith und Microservices.

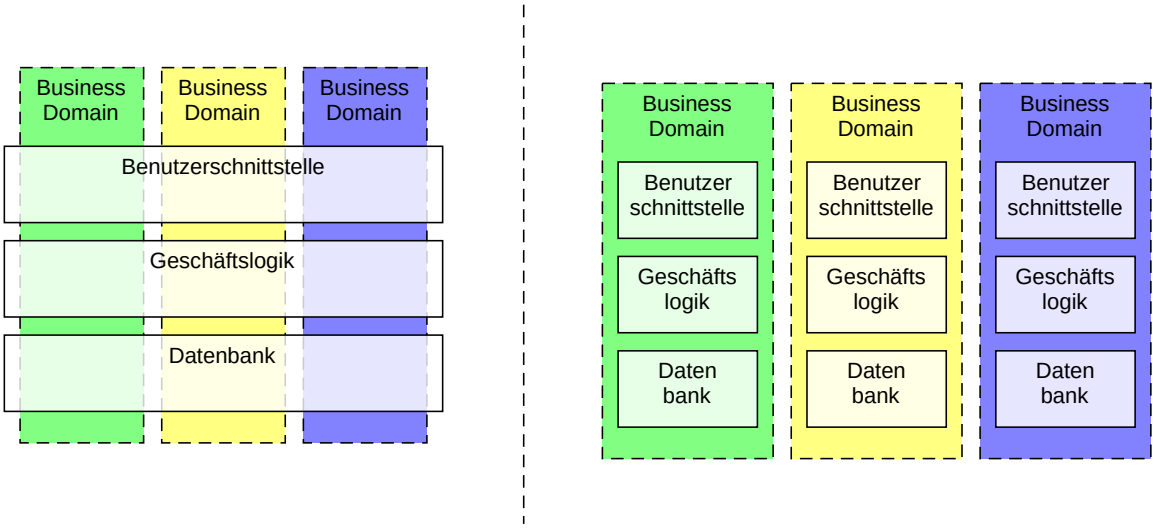

ABBILDUNG 2.3: Konzeptioneller Vergleich eines Deployment-Monolithen und Microservices aus technischer Sicht

2.3 Business Domain

Im Abschnitt 2.2 "Konzept" wurde beschrieben, dass Microservices anhand von Business Domains strukturiert werden. Sie spielen somit eine zentrale Rolle bei der Einführung von Microservice-Architekturen. Doch was ist eine Business Domain und wie identifiziert man diese? Eric Evans hat in seinem Werk "Domain-Driven Design: Tackling Complexity in the Heart of Software" das Konzept des Bounded Context eingeführt. Dieses Konzept hilft bei der Identifizierung und Abgrenzung der verschiedenen Business Domains. Formell gesehen ist ein Bounded Context der Gültigkeitsbereich eines fachlichen Modells.[9]

Das vorangegangene Beispiel des E-Commerce-Systems könnte aus drei Business Domains bestehen.

- der Registrierung
- dem Produktkatalog
- dem Bestellprozess

Die Business Domains können als unterschiedliche Bounded Context betrachtet werden. In diesen lässt sich das Modell des Benutzers anders behandeln, denn es spielt in jedem eine andere Rolle. Im Kontext der Registrierung spielen die Attribute Benutzername, E-Mail und Passwort die zentrale Rolle. Im Bounded Context des Produktkatalogs sind diese Attribute nicht von Belang, dafür kann die Eigenschaft "Vorlieben zur Angabe von Kaufangeboten" herangezogen werden. Die Rechnungs- und Lieferadresse wiederum sind wichtige Attribute für den Bestellprozess. Mit diesem Vorgehen lassen sich ebenfalls Informationen aufdecken, die zwischen den verschiedenen Bounded Context geteilt werden müssen.

Neben dem Bounded Context hat Evans viele weitere Konzepte zur fachlich getriebenen Modellierung von IT System eingeführt. Dazu zählen sieben verschiedene Interaktionsarten der Kontexte, die Published Language, mittels derer die Kontexte Informationen austauschen sowie weitere Patterns zur internen Ausgestaltung des Bounded Context.[10]

[9] vgl. Evans, *Domain-Driven Design: Tackling Complexity in the Heart of Software*.
[10] vgl. Plöd, „Microservices lieben Domain-driven Design“, S. 22 ff.

Es ist zu beachten, dass Microservices *Business Capabilities* (Fachkompetenzen) eines Bounded Context umfassen. Der Kontext sollte nicht bis auf CRUD-Operationen einzelner Entitäten heruntergebrochen werden.[11]

2.4 Technische Sicht

Im folgenden Abschnitt wird das Konzept der Microservice-Architektur aus der technischen Sicht beleuchtet. Der erste Abschnitt beschäftigt sich mit der Frage, warum Microservice-Architekturen mittels Services modularisiert werden. Der darauf folgende Teil analysiert die Kommunikationsarten der realisierten Services. Zuletzt wird die Datenhaltung der Microservices behandelt.

2.4.1 Modularisierung

Der Wunsch große Software Systeme aus kleinen Komponenten wie Lego-Steine zusammensetzen zu können, besteht schon lange. 1972 veröffentliche David Parnas das Paper "On the Criteria To Be Used in Decomposing Systems into Modules", in dem er das *Geheimnisprinzip (Information Hiding)* vorstellt. Nach diesem Prinzip werden Implementierungsdetails eines Moduls vor äußeren Zugriffen geschützt. Diese Details können komplex sein, und sie verändern sich mit hoher Wahrscheinlichkeit. Anhand definierter Schnittstellen sollen die Module dann in der Lage sein miteinander zu kommunizieren.[12]

Wird ein System mittels Interfaces, Klassen, Packages oder Ruby GEMs, Java JARs, .NET Assemblies oder Node.js NPMs modularisiert, schleichen sich im Laufe eines Projektes immer mehr ungewollte Abhängigkeiten(dependency) zwischen den Modulen ein. Für die Entwickler ist es besonders leicht ein Modul zu überarbeiten, um eine Information zu veröffentlichen. So wird eine Wartung oder Weiterentwicklung des Systems über einen längeren Zeitraum praktisch unmöglich. Denn durch diese wachsende Zahl an Abhängigkeiten der Module untereinander, steigt der Grad der Kopplung. Werden in einem eng gekoppelten System Änderungen an einem Modul vorgenommen, hat diese Änderung meist ungewollte oder unbewusste Auswirkungen auf andere Module.

[11] vgl. Morris, *How big is a microservice?*
[12] vgl. Parnas, „On the criteria to be used in decomposing systems into modules“.

In einer Microservice-Architektur wird die Software mittels Services in Komponenten unterteilt, die in separaten Prozessen ausgeführt werden, was zu einer *starken Modularisierung* führt und so die Erosion der Architektur eindämmt. Wolff beschreibt diese Art der Modularisierung als *stark*, weil Microservices über explizite Schnittstellen kommunizieren, die mittels Mechanismen wie Messages oder REST umgesetzt sind. Diese technische Hürde beugt ungewollter oder ungeplanter Kommunikation zwischen der Services bzw. Modulen vor und beschränkt die Erosion der Architektur auf den Geltungsbereich eines Service.[13]

Fowler definiert den Begriff der Komponente als eine Software-Einheit, die unabhängig ersetzt und geupgradet werden kann. Eine Bibliothek ist eine Komponente, die im Speicher aufgerufen wird. Setzt sich die Applikation aus Bibliotheken zusammen, die in einem Prozess ausgeführt werden, hat eine Änderung an einer Komponente ein erneutes Bereitstellen der gesamten Applikation zur Folge. Im Gegensatz dazu sind Services *out-of-process* Komponenten. Dies bedeutet, dass es sich um Komponenten handelt, die in unterschiedlichen Prozessen ausgeführt werden und Web Service- oder *Remote-Process* Aufrufe zur Kommunikation nutzen. Wird Software mittels Services komponentisiert (modularisiert), muss lediglich der veränderte Services erneut bereitgestellt werden. Änderungen an der Schnittstelle eines Services können zur Folge haben, dass mehrere Services koordiniert bereitgestellt werden müssen. Das Ziel einer guten Microservice-Architektur ist, diese Art der Kopplung mittels hoher Kohäsion der Services und Evolutionstechniken der Schnittstellen zu minimieren.[14]

2.4.2 Kommunikation

Microservices können auf zwei unterschiedliche Arten miteinander kommunizieren. Es wird zwischen der synchronen und der asynchronen Kommunikation unterschieden.

Bei der synchronen Kommunikation wird ein Aufruf an einen entfernten Service übermittelt. Der aufrufende Service wartet bis der Zielservice die Anfrage bearbeitet und beantwortet hat oder ein *Timeout* eintritt. Die Services kommunizieren direkt miteinander, möglicherweise über den Umweg eines Service-Discovery. Es handelt sich um eine *point-to-point* Verbindung, denn ein Service weist einen anderen an etwas zu tun. Kritisch betrachtet verstößt dies gegen das

[13] vgl. Wolff, *Microservices: Grundlagen flexibler Softwarearchitekturen*, S. 3 f.
[14] vgl. Fowler, *Microservices*.

Prinzip der eindeutigen Verantwortlichkeit und wirkt sich negativ auf die lose Kopplung der Services aus. Die übermäßige Delegation an eine Vielzahl von Services kann zu einer Hairball-Architektur oder einem "Big Ball of Mud" führen, bei der die Abhängigkeiten der Module nicht mehr nachvollziehbar sind. Diese Art der Kommunikation hat jedoch auch Vorteile: Sie ist leicht verständlich und weicht nicht allzu stark von der Komponenten-Kommunikation innerhalb eines Deployment-Monolithen ab. Außerdem ist für die Implementierung synchroner REST-Aufrufe mittels des HTTP-Protokolls keine weitere Infrastruktur nötig.[15]

Kommunizieren die Services eines Microservice-Architektur hingegen asynchron, wartet der Aufrufer nicht auf die Antwort des entfernten Services. Die Antwort, oder ob die Anfrage erfolgreich bzw. fehlerhaft abgeschlossen wurde, sind für ihn möglicherweise nicht von Interesse. Diese Art der Kommunikation eignet sich besonders gut bei zeitintensiven Aufgaben, bei denen es von Nachteil, wäre die Verbindung über einen längeren Zeitraum aufrecht zu erhalten, oder bei Szenarien, in denen geringe Latenzzeiten erforderlich sind. Im Kontext von Microservices wird die Kommunikationsart oftmals mittels *Asynchronous Messaging* implementiert. Das Konzept sieht vor, dass die Services sich an einer *Pipe* registrieren. Über diesen Kommunikationskanal können Ereignisse propagiert werden. Tritt ein Ereignis ein, das für den registrierten Service von Interesse ist, reagiert dieser, indem er seine Arbeit aufnimmt und ggf. weitere Ereignisse erzeugt. Fowler ist hier besonders wichtig, dass die Pipe keine Geschäftslogik beinhaltet. Sie soll lediglich die Nachrichten weiterleiten. Er spricht von "smart endpoints and dumb pipes".[16,17]

2.4.3 Datenhaltung

Im Kapitel 2.3 "Business Domain" wurde beschrieben, dass Microservices anhand des Bounded Context strukturiert werden können. Neben der Geschäftslogik umfasst dieser Kontext auch die Daten einer Domäne. Eine Entität kann in zwei Kontexten aus unterschiedlichen Attributen bestehen oder es wird je nach Kontext anders interpretiert. Auf einer abstrakten Ebene beschrieben bedeutet dies, dass sich das Modell der Welt zwischen den Systemen unterscheidet.

[15] vgl. Williams, *Is REST Best in a Microservices Architecture?*
[16] vgl. Fowler, *Microservices*.
[17] vgl. Newman, *Microservices (mitp Professional): Konzeption und Design*, S. 70.

Bei einem klassischen Deployment-Monolithen, der eine relationale Datenbank zur Datenspeicherung nutzt, greifen die verschiedenen Module typischerweise auf gemeinsame Tabellen zu. Beispielsweise nutzen Registrierung und Bestellprozess die Benutzertabelle, um Daten des Benutzers abzufragen. Dabei können Attribut des Benutzer, wie etwa das Alter, entweder für beide Services von Interesse sein, oder nur für einen. Dies führt zu einer sehr engen Kopplung der Module. Möchte Modul A beispielsweise den Namen eines Attributs ändern, muss das Datenbankschema der Tabelle angepasst werden. Diese Änderung führt zu einem Kompatibilitätsproblem in Modul B, denn dieses greift auf eine veraltete Version des Schemas zu. Noch komplexer wird es, wenn ein Attribut zwischen den Modulen eine unterschiedliche Semantik aufweist. Das Attribut Titel aus der Benutzertabelle könnte für einen Service "Position" bedeuten - Sachbearbeiter oder Bereichsleiter, für einen anderen verweist es auf die "Ansprache" - Frau oder Herr.

Neben der Dezentralisierung des konzeptionellen Designs, wird die Datenhaltung ebenfalls verteilt realisiert, um einen hohen Grad an loser Kopplung zu erreichen. Dies wird realisiert, indem jeder Microservice, falls nötig, auf seinen eigenen Datenspeicher zugreift. Um ein Problem bestmöglich zu lösen, können sich die verwendeten Technologien unterscheiden. Denkbar ist, dass Microservice A eine relationale Datenbank und Microservice B eine NoSQL-Datenbank verwendet.[18,19]

2.5 Organisatorische Sicht

In diesem Abschnitt werden wichtige organisatorische Eigenschaften von Microservice-Architekturen dargestellt. Zunächst wird beschrieben, auf welche Weise sich Conways Gesetz auf die Architektur von Software auswirkt und wie Microservices sich dieses Gesetz zu Nutze machen. Im zweiten Teil werden die Begriffe Mikro- und Makroarchitektur definiert.

[18] vgl. Fowler, *Microservices*.
[19] vgl. Newman, *Microservices (mitp Professional): Konzeption und Design*, S. 119 - 126.

2.5.1 Fachliche Teams

"Any organization that designs a system (defined broadly) will produce a design whose structure is a copy of the organization's communication structure." (Melvyn Conway, 1967)

Nach dem oft zitierten Gesetz von Conway produzieren Unternehmen Systemdesigns, die ihre Kommunikationsstruktur widerspiegeln. Dabei muss diese Struktur nicht zwangsläufig dem Organigramm entsprechen. Conway meint damit viel mehr die realen Kommunikationswege von Personen und Teams.

Bei dem Versuch große Applikationen handhabbar zu machen, fokussieren sich Organisationen oftmals auf die Strukturierung nach technischen Schichten eines Systems. Dies führt dazu, dass es Teams für das User Interface (UI), das Backend, die Datenbank und den Betrieb gibt. Um die Kommunikationswege der Kollegen zu optimieren, wird ein Team bevorzugt an einem Standort angesiedelt. Dieses Vorgehen hat den Vorteil, dass es für jede technische Schicht Experten gibt, die aufgrund der örtlichen Nähe Erfahrungen austauschen und Hilfestellung geben können. Anforderungen, die sich auf eine technische Schicht beschränken, können schnell realisiert werden. Diese Art der Anforderung ist jedoch eher selten. Oftmals wirkt sich eine Änderung auf mehrere oder sogar alle technischen Schichten aus. Dies hat einen hohen Kommunikationsaufwand zwischen den Teams zur Folge. Je größer der örtliche oder zeitliche Abstand zwischen den Teams ist, desto größer auch der entstehende Overhead bei der Realisierung. Wird der Aufwand als zu hoch eingeschätzt, könnte ein Team versuchen die Anforderung ausschließlich in der jeweiligen technischen Schicht zu lösen. Dies führt dazu, dass sich die Geschäftslogik auf alle Schichten verteilt, eine Erosion der Architektur eintritt und die Wartbarkeit des Systems nachteilig beeinflusst wird.

Mittels Microservices wird das System entlang der Geschäftsbereiche in Services unterteilt. Zur Entwicklung eines solchen Service sind alle technischen Schichten und organisatorischen Prozesse notwendig. Microservices machen sich Conways Gesetz zu nutze und strukturieren die Teams entlang dieser Services. Dies bedeutet, dass ein Team die Verantwortung für den gesamten Lebenszyklus eines oder mehrerer Services übernimmt und *Cross Fuctional* aufgestellt sein muss. Die Teams bestehen dementsprechend aus Personen, die Kenntnisse in den

Bereichen UI, Backend, Datenbank, Betrieb und Projektmanagement haben müssen. Auf diese Weise verkürzen sich die Kommunikationswege bei der Realisierung neuer Features.[20]

Ein weiterer Vorteil ist, dass mittels Microservices nicht nur das System aus technischer Sicht skaliert werden kann, sondern auch das Unternehmen aus organisatorischer Sicht. Es wird nicht länger an einem großen Projekt gearbeitet, sondern an vielen kleinen, die ein gemeinsames Ziel verfolgen. Entsteht der Bedarf für einen neuen Service, kann ein Team die Verantwortung für diesen übernehmen oder es wird ein neues Team geschaffen. Neben der *vertikalen Skalierung* der Teams wird das Risiko eines großen Projektes auf viele kleine verteilt. Große Vorhaben werden so potenziell risikoärmer, reaktionsfähiger, schneller, innovativer und für Entwickler interessanter.[21]

2.5.2 Mikro-/Makroarchitektur

Bei der Entwicklung eines Deployment-Monolithen wird sich im Normalfall auf einen Technologie Stack beschränkt. Außerdem ist es aufgrund von Inkompatibilitäten gegebenenfalls unmöglich, die Version eines verwendeten Frameworks zu ändern. Nutzen Microservices ein technologisch unabhängiges Kommunikationsprotokoll, werden die Teams in die Lage versetzt, jeden Service mittels einer anderen Technologie zu realisieren. Auf diese Weise kann ein Problem mit der passenden Lösung bewältigt werden. Ist Java, C++ oder Node.js zur Lösung eines Problems besser geeignet als andere, steht der Verwendung dieser Technologie aus technischer Sicht nichts im Weg. Besteht eine Anforderung an die Datenbank, die mittels einer NoSQL Datenbank besser umgesetzt werden kann als mit Hilfe einer relationalen Datenbank, ist dies ebenfalls möglich. Dieser Spielraum wird als *Mikroarchitektur* bezeichnet. Die Teams dürfen eigenständig Entscheidungen treffen, um so individuelle Lösungen zu entwickeln. Sie lässt Raum für Innovation, die dann wiederum in einen Standard überführt werden können.

Die Freiheit aus einer großen Menge an Möglichkeiten wählen zu können und ein stark polyglottes System zu entwickeln, bedeutet nicht, dass man das immer tun sollte. Auch in einer Microservice-Architektur ist es wichtig harte Regeln zu definieren um Anforderungen wie Kommunikation, Monitoring und Logging effizient lösen zu können. Auf der anderen Seite soll den

[20] vgl. Wolff, *Microservices: Grundlagen flexibler Softwarearchitekturen*, S. 273 - 280.
[21] vgl. Toth, „Evolution statt Diktatur“, S. 12.

Teams so viel Freiheit wie möglich zugesprochen werden, damit Anforderungen bestmöglich umgesetzt werden können.[22]

Die *Makroarchitektur* eines Microservice-Projektes besteht aus diesen harten Regeln. Sie sind eine Art der Standardisierung und lassen sich nachträglich nur schwer ändern. Dementsprechend sollte die Makroarchitektur schlank gehalten werden und nichts vorgeben, was sich schnell oder oft ändert. Neben den harten Regeln kann die Makroarchitektur auch Vorschläge beinhalten. Diese Vorschläge sind Ideen oder konkrete Implementierungen, die einen präferierten Weg aufzeigen. Sie können aus Prinzipien, Bibliotheken oder Konzepten bestehen. Durch die Verbreitung bewährter Lösungen, die einen Mehrwert bieten, bewegt man Entwickler dazu ähnliche Probleme mit ähnlichen Lösungen zu bewältigen.[23,24]

2.6 Varianten

Wie man dem Kapitel 2.1 "Definition" entnehmen kann, gibt es keine eindeutige Definition was Microservices sind. Die Definitionsversuche bestehen bislang aus Konzepten und Eigenschaften, die Microservice-Architekturen aufweisen. Bei der Implementierung dieses Paradigmas gibt es viele Entscheidungen und Kompromisse, die getroffen werden müssen. Aus diesem Grund entstehen verschiedene Ausprägungen des Microservice-Paradigmas. Unterschiede sind vor allem in der Größe (und somit auch der Menge an Deployments) der Services, der Integration und der Skalierbarkeit der Teams zu beobachten.[25] Es gibt jedoch auch Konstanten, die alle Microservice-Architekturen aufweisen. Es handelt sich immer um ein verteiltes System, bestehend aus einzeln deploybaren Komponenten, das entlang der Fachlichkeit modularisiert wird. Jede der nachfolgend vorgestellten Varianten geht unterschiedliche Kompromisse ein, um die Geschäfts- bzw. Architekturziele des jeweiligen Kontextes bestmöglich erreichen zu können.

[22] vgl. Fowler, *Microservices*.
[23] vgl. Toth, „Evolution statt Diktatur“, S. 12 f.
[24] vgl. Zörner, „Bring your own Architecture“, S. 16 f.
[25] vgl. Wolff, *Microservices 101: Varianten, Herausforderungen, Erfahrungen aus der Praxis*.

2.6.1 Developer Anarchy

In diesem von Fred George entwickelten Stil wird den Entwicklern die größtmögliche Freiheit gegeben. Die Entwickler arbeiten ohne jegliches Management und übernehmen die gesamte Verantwortung für den erfolgreichen Abschluss eines Projektes. Dieses wird in Form einer "selbst organisierenden Anarchie" realisiert. Jeder Entwickler hat die Freiheit selbst zu entscheiden, welche Programmiersprache oder Frameworks zur Realisierung der Anforderung genutzt werden sollen. Ein Beispiel für ein Projekt in dem dieses Vorgehen funktioniert, ist der Webauftritt der britischen Zeitung *The Guardian*.[26] In einem solchen Szenario sind die entwickelten Services besonders klein. Laut Wolff sind Services, die aus 10 - 100 Zeilen Code bestehen, durchaus realistisch. Aufgrund der großen Anzahl an Services sollte die Integration mittels eines asynchronen Kommunikationskanals erfolgen.[27]

2.6.2 Layered

In einem geschichteten Ansatz (Layered) sind die Services wesentlich größer als in der 2.6.1 "Developer Anarchy" und implementieren komplexere Geschäftsfunktionalitäten. Das Ziel ist die Entwicklung von Microservices, die wiederverwendbar sind. Dazu werden sie in Schichten unterteilt. Die Präsentationsschicht besteht aus Services, die für verschiedene Endgeräte optimiert sind. Darunter folgen 1-n Schichten, die fachliche Services beinhalten. Bei einem Aufruf wird die Anfrage in einer Baumstruktur ggf. bis auf die unterste Schicht an Microservices weitergereicht.[28] Ein Team übernimmt typischerweise die Verantwortung für einen oder mehr Services und folgt den Prinzipien von DevOps. Vereinfacht bedeutet dies, dass das Team für die Planung, Entwicklung, Betrieb und Wartung der Services zuständig ist. Unternehmen wie Netflix haben diese Variante sehr erfolgreich realisiert und folgen ihr weiter. Netflix hat zudem die Besonderheit, dass sie technisch hauptsächlich auf Java setzen. Die Entwickler haben die Freiheit jegliche Technologie zu nutzen, jedoch haben sie auch die Konsequenzen ihrer Entscheidungen zu tragen. Lösungen für querschnittliche Anforderungen wie Logging, Security

[26] vgl. Jardine, *"Do what you want": building great products through anarchy*.
[27] vgl. Wolff, *Microservices 101: Varianten, Herausforderungen, Erfahrungen aus der Praxis*.
[28] vgl. Wolff, „Self-contained Systems: Microservices-Architektur mit System“.

oder Monitoring, werden durch spezielle Teams entwickelt und als Java Bibliotheken bereitgestellt. Anderen Teams steht es frei diese zu nutzen.[29]

2.6.3 Self-contained Systems

Ein Self-contained System (SCS) oder System-of-Systems besteht aus autonomen Webanwendungen - im Folgenden als Teilsystem bezeichnet -, die zu einem Gesamtsystem integriert werden. Idealerweise wird jedes von einem eigenen Team betreut werden. Jedes dieser Teilsysteme kann eine Drei-Schicht-Architektur, bestehend aus Präsentation, Geschäftslogik und Persistenz, aufweisen. Typischerweise werden die Teilsystem über das UI integriert. Im einfachsten Fall verweisen Links von einem Teilsystem auf das andere und bilden so das Gesamtsystem. Neben der UI-Integration können die Teilsystem HTML-Snippets andere Teilsystem integrieren (Transklusion) oder Informationen idealerweise mittels Messaging austauschen. Um die Anwendungen bestmöglich voneinander zu entkoppeln, sollte die Kommunikation hauptsächlich asynchron erfolgen und jedes Teilsystem auf seine eigene Datenbank zugreifen.[30] Das Versandhandelsunternehmen Otto realisiert Microservices in dieser Variante.[31]

2.6.4 Service-oriented Architecture

Bereits die Beschreibung einer Service-oriented Architecture (SOA) ist nicht einfach. Wie bei Microservices auch, fehlt es der SOA an einer *eindeutigen* Definition. Viele verschiedene Personen haben ein ganz unterschiedliches Bild von SOA. Für manche ist der Kern von SOA, dass die Software mittels Webservices bereitgestellt wird. Für andere liegt der Fokus auf Services, die Geschäftslogik realisieren. UI-Aggregatoren aggregieren die Geschäftslogik mehrerer Services und stellen sie dem Benutzer bereit. Für wieder andere bedeutet SOA, dass mittels asynchroner Kommunikation, Dokumente zwischen Systemen verteilt werden.[32] Dementsprechend wurden in den vergangenen Jahren viele, sehr unterschiedliche Architekturen entworfen, die

[29] vgl. Wolff, *Microservices 101: Varianten, Herausforderungen, Erfahrungen aus der Praxis*.
[30] vgl. innoQ, *Self-Contained Systems - Assembling Software from Independent Systems*.
[31] vgl. Steinacker, *Why Microservices?*
[32] bgl. Fowler, *Microservices*.

von ihren Schöpfern als SOA-Architekturen bezeichnet werden. Eines der Ziele dieser Varianten kann die Erhöhung der Wiederverwendbarkeit von Programmteilen sein. Beispielsweise sollten zwei oder mehr Programme für Endanwender die gleichen Services nutzen können.[33]

Fowler berichtete davon, dass die meisten SOA Projekte, die er kennenlernen durfte aus kleinen *technisch orientierten* Services bestehen, die mittels eines Enterprise Service Bus (ESB) miteinander in Verbindung gesetzt wurden. Im Laufe der Zeit wurden diese ESBs jedoch mit immer mehr Logik ausgestattet, was zu einem sehr komplexen und schwer wartbaren System führt. Dies ist das Komplement zum Prinzip von *dumb pipes and smart endpoints*. Aufgrund der unterschiedlichen Auffassung, wie eine SOA-Architektur aussieht, gibt es jedoch auch Unternehmen, die eine uns heute als Microservice bekannten Architektur unter dem Namen von SOA implementiert haben.[34,35]

Laut Newman ist der Microservice Ansatz "... aus praktischen Anwendungen hervorgegangen und baut dabei auf unser besseres Verständnis der Systeme und Architekturen auf, um SOA richtig zu implementieren.".[36] Er und Fowler sehen Microservices als eine Teilmenge des Verständnisses von SOA an. SOA bedeute so viel, dass praktisch alles sein kann. Nach Fowler werden sinnvolle Konzepte von SOA unter dem Begriff Microservices zusammengefasst.

2.6.5 Service-based Architecture

Neal Ford betrachtet die Architektur eines Deployment-Monolithen, die von SOA und Microservices aus drei Perspektiven: Der Service-Granularität, der Datenbank und der Integrationsschicht. In seinem Ansatz wägt er die Vor- und Nachteile der verschiedenen Architekturen unter den besagten Gesichtspunkten ab und stellt so einen pragmatischen Lösungsansatz vor. Im Sinne der Service-Granularität wird das System in eine geringe Anzahl von lose gekoppelten, domänenspezifischen Services aufgespalten. Ein Service kann weiter in *in-process* Komponenten (siehe Abschnitt 2.4.1 "Modularisierung") unterteilt sein. Diese Macroservices umgehen Probleme der Orchestrierung und Transaktion. All diese Services nutzen eine Datenbank und

[33] vgl. Newman, *Microservices (mitp Professional): Konzeption und Design*, S. 30.
[34] vgl. Fowler, *Microservices Guide*.
[35] vgl. Fowler, „Microservices“.
[36] Newman, *Microservices (mitp Professional): Konzeption und Design*, S. 30.

teilen den Datenbestand. Dies führt zu wenigen Netzwerkaufrufen zur Kommunikation der Services und ist besonders dann sinnvoll, wenn das Refactoring der Datenbank nicht sinnvoll oder unmöglich ist. Die Integration der UI und der Macroservices erfolgt mittels einer intelligenten Middleware. Diese erlaubt das Transformieren von Daten zwischen inkompatiblen Schnittstellen und bildet einen Punkt zum Ausführen von Logik, über alle Services hinweg.

In seinem Vortrag beleuchtete er die Vor- wie auch die Nachteile einer Service-based Architecture (SBA). Zu nennen ist der Entwicklungsaufwand eines größeren Macroservice, die Planung der Delivery-Pipeline, die Vermischung der Bounded Context von Services, den teuren Änderungen des Datenbankschemas und der hohen Komplexität einer intelligenten Middleware. Dennoch sei diese Architektur für Migrationsprojekte gut geeignet, ein guter Kompromiss in vielerlei Hinsicht und Domain-Orientiert, ohne sehr kleine Microservices zu entwickeln.[37,38]

2.7 Vorteile und Herausforderungen

Gegenüber einem Deployment-Monolithen versprechen Microservices verschiedene Vorteile, bergen aber auch einige Herausforderungen, die nicht vernachlässigt werden dürfen. Diese lassen sich auf einer technischen, organisatorischen, architektonischen und betrieblichen Ebene einordnen. In den folgenden Kapiteln werden diese zusammengefasst.

2.7.1 Technische Vorteile

Durch die besondere Art der Modularisierung mittels Microservices ergeben sich einige technische Vorteile, die in den folgenden Abschnitten beschrieben werden.

Starke Modularisierung

Wie bereits in Kapitel 2.4.1 "Modularisierung" erläutert, handelt es sich bei Microservices um ein Konzept zur Modularisierung von Software. Der Aufruf eines anderen Moduls bzw. Microservice ist nur mittels verteilter Kommunikation über das Netzwerk möglich. Dies stellt für den

[37] vgl. Ford, „Comparing Servicebased Architectures“.
[38] vgl. Fletcher, *Service-Based Architecture as an Alternative to Microservice Architecture*.

Entwickler, die eine ungeplante Abhängigkeit von Modul A zu Modul B einführen will, eine technische sowie organisatorische Hürde. Er muss einen entfernten Aufruf starten, der eine explizite Schnittstelle bedarf. Um diese Schnittstelle zu implementieren, ist jedoch ggf. die Kommunikation mit einem anderen Team nötig.

Im Vergleich dazu sind die verschiedenen Module eines Deployment-Monolith direkt aufruf- und änderbar. So können sich unerwünschte Abhängigkeiten sehr schnell einschleichen. Mit der Zeit erodiert die Architektur des Systems. Microservices modularisieren die Anwendung in einer starken Weise, da das Überschreiten der Grenzen zwischen den Modulen schwierig ist. Wolff beschreibt diese Grenzen als Firewalls, die den Zerfall der Architektur aufhalten.[39,40]

Austauschbarkeit

Während der Planung und Implementierung von Software, werden viele unterschiedliche Aspekte berücksichtigt. Die Ablösung des entwickelten Systems am Ende des Lebenszyklus wird jedoch oftmals wenig beachtet oder ganz ignoriert. Heute stehen jedoch viele Unternehmen vor der Aufgabe, historisch gewachsene Systeme, die geschäftskritische Prozesse unterstützten, ganz oder teilweise ablösen oder erneuern zu müssen. Änderungen werden an solchen Legacy Systemen nur widerwillig gemacht, denn die steigende Komplexität und die Erosion der Architektur führen zu einem gestiegenen Risiko von Ausfällen.

Ein Vorteiler von Microservice-Architekturen ist, dass sie dieses Risiko senken. Die Auswirkungen von Änderungen können aufgrund der losen Kopplung und der hochgradigen Geschlossenheit wesentlich präziser eingeschätzt werden. Wird ein Microservice zu einem Legacy System, kann er mit geringerem Risiko ganzheitlich ausgetauscht oder erneuert werden. Bei diesem Austauschen könnten beispielsweise veraltete Technologien ersetzt werden. Selbst wenn der Austausch fehlschlagen sollte, lässt sich ein Totalausfall verhindern, denn nur ein Teil des verteilten Systems ist davon betroffen. In einigen der vorgestellten Microservice-Varianten lassen sich die Services zudem schnell und risikoarm neu implementieren, denn sie haben eine besonders kleine Codebasis.[41,42]

[39] vgl. Wolff, *Microservices: Grundlagen flexibler Softwarearchitekturen*, S. 59 f.
[40] vgl. Newman, *Microservices (mitp Professional): Konzeption und Design*, S. 28 f.
[41] vgl. Wolff, *Microservices: Grundlagen flexibler Softwarearchitekturen*, S. 60 f.
[42] vgl. Newman, *Microservices (mitp Professional): Konzeption und Design*, S. 29.

Skalierbarkeit

Die Skalierung von monolithischen Anwendungen muss ganzheitlich erfolgen. Ist nur ein kleiner Teil des Systems unzureichend leistungsfähig, muss das System als Ganzes skalieren, da es ein untrennbarer Teil des Deployment-Monolithen ist. Wird das System in kleinere Services zerschlagen, kann gezielt die Leistungsfähigkeit der betroffenen Komponenten gesteigert werden. Ist ein System im Einsatz, das Ressourcen nach Bedarf bereitstellt, können Bestandteile des Systems automatisch skalieren. Anfallende Kosten können so wesentlich effektiver gesteuert werden. Ein weiterer Vorteil ist, dass einzelne Services an verschiedenen Stellen im Netzwerk bereitgestellt werden können, die Übertragungsstrecke zwischen Service und Aufrufer wird verkürzt. Das Resultat kann eine verkürzte Antwortzeit sein.

Die Skalierbarkeit welche mittels out-of-process Komponenten erreicht wird, ist jedoch ein zweischneidiges Schwert. Einerseits sorgen sie dafür, dass neue Instanzen bei bedarf hinzugefügt werden können, andererseits nutzen sie Netzwerkaufrufe. Diese sind deutlich langsamer als Aufrufe, die mittels des internen Speichers realisiert werden.[43,44]

Robustheit

In einer Microservice-Architektur sind die Komponenten des Systems voneinander isoliert. Die Services werden auf eigenen Maschinen betrieben. Stürzt ein Service oder eine Maschine ab, sind die anderen davon nicht betroffen. Im Gegensatz dazu kann der Gesamtausfall eines Deployment-Monolithen von einer unkritischen Komponente verursacht worden sein.

Es ist unabdinglich ein Verständnis für die Fehlerquellen eines verteilten Systems, Ausfälle und deren Auswirkung zu entwickeln, um die Robustheit einer Microservice-Architektur zu steigern. Netzwerke und Maschinen fallen aus, und jeder integrierten Microservice steigert prinzipiell die Wahrscheinlichkeit von Teilausfällen, da mehr Services miteinander kommunizieren und mehr Maschinen betrieben werden.[45,46]

[43] vgl. Wolff, *Microservices: Grundlagen flexibler Softwarearchitekturen*, S. 64 f.
[44] vgl. Newman, *Microservices (mitp Professional): Konzeption und Design*, S. 26 f.
[45] vgl. Wolff, *Microservices: Grundlagen flexibler Softwarearchitekturen*, S. 65 f.
[46] vgl. Newman, *Microservices (mitp Professional): Konzeption und Design*, S. 26.

Technologische Wahlfreiheit

Microservices bieten die Freiheit unterschiedliche Technologien nutzen zu können. Dank einer technologieunabhängigen Kommunikation der Services, kann ein Microservice mit Java und ein anderer mit C++ umgesetzt werden. Für ein bestimmtes Problem, das ein Microservice löst, kann demnach das passende Werkzeug ausgewählt werden. Dies ist eine Option und keinesfalls verpflichtend, denn mit steigender Anzahl an Programmiersprachen steigt die Komplexität des Systems. Die technologische Wahlfreiheit kann sich jedoch auch im Kleinen abspielen. Innerhalb eines Deployment-Monolithen können meist nicht mehrere Versionen eines Frameworks genutzt werden. Microservices brechen dieses enge Korsett auf und ermöglichen Versionsunterschiede zwischen den Services.

Ein weiterer großer Vorteil ist, dass mittels Microservices neue Technologien mit geringem Risiko ausprobiert werden können. Schlägt solch ein Experiment fehl, sind Ausfälle isoliert und im schlimmsten Fall kann der Service innerhalb kurzer Zeit von einer auf bewährten Technologien basierenden Implementierung abgelöst werden.[47,48]

Continous-Delivery

Continuous-Delivery (CD) bringt Software automatisiert und regelmäßig in Produktion. Ein Deployment-Monolith kann aus Millionen von Codezeilen bestehen. Wird auch nur eine Zeile geändert, muss die Applikation als Ganzes erneut in Produktion gebracht werden. Dies kostet allerdings viel Zeit und kann erhebliche Auswirkungen nach sich ziehen. In der Praxis werden aus diesem Grund Systeme oftmals nur in großen Zeitabständen aktualisiert, was dazu führt, dass sich die Modifikationen ansammeln. Mit steigender Größe des tatsächlichen Versionsunterschiedes steigt das Risiko einer fehlerhaften Bereitstellung. Es entsteht ein Teufelskreis.

Microservices bieten den Vorteil, dass sie einzeln deploybar sind. Das Optimum ist, dass mittels CD eine vollautomatisierte Bereitstellung der Services erfolgt. Die schnelle Auslieferung von Funktionalität ist dabei nicht das einzige Ziel. Regelmäßige Deployments und zeitnahes

[47] vgl. Wolff, *Microservices: Grundlagen flexibler Softwarearchitekturen*, S. 66 f.
[48] vgl. Newman, *Microservices (mitp Professional): Konzeption und Design*, S. 24 ff.

Feedback an die Entwickler, ob die implementierten Funktionalitäten in der Betriebsumgebung funktionieren, ermöglichen kürzere Releasezyklen.

Es ist zu beachten, dass Microservice-Architekturen die Realisierung von CD vereinfachen. Umgekehrt ist CD eine Voraussetzung, um die Stärken von Microservices voll ausschöpfen zu können.[49,50]

Umgang mit Legacy

Microservice-Architekturen bieten einen einfacheren Weg Legacy Anwendungen um neue Funktionalität zu erweitern. Während bei einem Deployment-Monolith das Legacy System selbst unter großem Risiko angepasst werden muss, kann ein Service einer Microservice-Architektur die Altanwendung indirekt erweitern. Anstatt neue Geschäftslogik in das Altsystem zu integrieren, wird es um einen unabhängigen Service erweitert. Dazu muss die Legacy Anwendung Schnittstellen zur Kommunikation bereitstellen. So kann es um Funktionen erweitert oder etappenweise ersetzt werden. Eine andere Möglichkeit ergibt sich, wenn ein Proxy-Microservice eingesetzt werden kann. Dieser nimmt die Anfragen des Benutzers entgegen, bearbeitet sie selbst, leitet sie an einen anderen Microservice oder an das Legacy System zur Bearbeitung weiter. Nicht jede Lösung muss zwangsläufig dem Idealbild eines Microservice entsprechen. Es ist durchaus denkbar, dass das Legacy System und der Microservice während der Migrationsphase die gleiche Datenbank nutzen und die Ergebnisse beider Komponenten konsistent bleiben. Microservice bieten eine ganze Reihe an Möglichkeiten wie Legacy Anwendungen angebunden oder migriert werden können.[51]

2.7.2 Organisatorische Vorteile

Die technischen Vorteile von Microservice-Architekturen bilden das Fundament, aus dem Nutzen für die Organisation ziehen kann.

[49] vgl. Wolff, *Microservices: Grundlagen flexibler Softwarearchitekturen*, S. 63 f.
[50] vgl. Newman, *Microservices (mitp Professional): Konzeption und Design*, S. 27 f.
[51] vgl. Wolff, *Microservices: Grundlagen flexibler Softwarearchitekturen*, S. 61 f.

Autonome Teams

Nutzt ein Unternehmen die hohe technische Unabhängigkeit von Microservices aus und macht es sich das Gesetz von Conway (siehe Kapitel 2.5.1 "Fachliche Teams") zunutze, können die Teams selbstständig agieren. Die technische Basis und organisatorische Maßnahmen, erlauben den Teams mit geringer Koordination zu arbeiten. Sie sind ermächtigt, eigenständig Entscheidungen zu treffen. Sie sind für den gesamten Lebenszyklus eines Service verantwortlich. Angefangen bei der Anforderungsermittlung, dem Design, über die Implementierung, dem Betrieb, bis hin zu Wartung des Service handeln die Teams eigenverantwortlich. Dies schließt ebenfalls alle Probleme mit ein, wie beispielsweise den Ausfall eines Service im Produktivbetrieb. Entscheidungen, die z.B. das Logging betreffen, können dennoch zentral getroffen werden (siehe Kapitel 2.5.2 "Mikro-/Makroarchitektur"). Diese Art der Organisationsform unterscheidet sich stark von der klassischen Arbeitsweise, in der alle Architekturentscheidungen von einer zentralisierten Architektengruppe getroffen werden.

Am Ende all dieser Maßnahmen steht die Skalierbarkeit von agilen Prozessen. Arbeit, z.B. in Form von Features, kann besser parallelisiert und auf die Teams verteilt werden. Kommt das Unternehmen an einen Punkt, an dem die bestehenden Teams voll ausgelastet sind, kann ein Neues gebildet werden.[52,53]

Kleine Projekte

Microservice-Architekturen erlauben letztendlich, ein großes Projekt in viele kleine zu zerlegen, die aus Sicht des Projektmanagements weniger Koordinationsaufwand benötigen als Großprojekte. Dies hat den Vorteil, dass die kleineren Projekte besser planbar, genauer schätzbar und der Verlust durch Fehlentscheidungen geringer ist. Hinzu kommt, dass der Kommunikationsaufwand bei Großprojekten unverhältnismäßig ansteigt. Alle diese Faktoren senken das Risiko eines Fehlschlags. Das gesenkte Risiko zusammen mit der technischen und organisatorischen Flexibilität führen dazu, dass Entscheidungen schneller und einfacher getroffen werden können und dass das Gesamtprojekt agiler wird.[54,55]

[52] vgl. Wolff, *Microservices: Grundlagen flexibler Softwarearchitekturen*, S. 67 - 71.
[53] vgl. Wolff, „Schein und Sein“, S. 9.
[54] vgl. Wolff, *Microservices: Grundlagen flexibler Softwarearchitekturen*, S. 69.
[55] vgl. Toth, „Evolution statt Diktatur“, S. 12.

2.7.3 Technische Herausforderungen

Die meisten technischen Herausforderungen sind eine Folge davon, dass es sich bei Microservices um ein verteiltes System handelt. Einige, der bisher bekannten, werden in den folgenden Abschnitten erläutert.

Code-Abhängigkeiten

Eines der wichtigsten Ziele, die es mit Microservices zu erreichen gilt, ist die Möglichkeit sie unabhängig voneinander deployen zu können. Erst dann können viele der Vorteile genutzt werden. Binärabhängigkeiten können dieses Ziel zunichtemachen. Ein Beispiel für solche Abhängigkeiten sind Bibliotheken, mit deren Hilfe der Versuch unternommen wird, einmal geschriebene Funktionalität wiederzuverwenden. Nutzt man das gleiche Prinzip im Microservice-Kontext, werden die Services eng miteinander gekoppelt. Sind mehr als ein Service von solch einer Abhängigkeit betroffen, führen Änderungen an dieser zentralen Komponente oftmals dazu, dass alle betroffenen Services erneut bereitgestellt werden müssen. Im schlimmsten Fall kann es sogar dazu führen, dass die Services in einer bestimmten Reihenfolge gestartet werden müssen. Dies wirkt konträr zu der Idee von unabhängig deploybaren Einheiten.

Aus diesem Grund präferieren Microservice-Architekturen den Shared-Nothing-Ansatz. Code-Redundanzen werden bewusst in Kauf genommen, um eine enge technische und organisatorische Kopplung zu vermeiden.

In manchen Fällen kann es dennoch sinnvoll sein, Bibliotheken in mehreren Services zu nutzen. Der wiederverwendete Code sollte jedoch keine Businesslogik enthalten, denn änderte sich die Logik in einem Kontext, wirkt sich die Änderung auch auf die anderen aus. Letzten Endes müssen diese wieder koordiniert bereitgestellt werden. Bibliotheken, die jedoch technische Probleme wie Logging oder Monitoring lösen, können durchaus sinnvoll sein. Es ist jedoch darauf zu achten, dass sie bei Änderungen abwärtskompatibel bleiben. So sind die Nutzer nicht gezwungen, zeitgleich reagieren zu müssen.[56]

[56] vgl. Wolff, *Microservices: Grundlagen flexibler Softwarearchitekturen*, S. 73 - 76.

Unzuverlässige Kommunikation

Wie bereits mehrfach erwähnt, kommunizieren Komponenten einer Microservice-Architektur über das Netzwerk miteinander. Dies ist im Vergleich zur Kommunikation von klassischen Modulen, die den Speicher als Kommunikationsmedium nutzen, äußerst unzuverlässig. Denn Netzwerke können ausfallen oder verzögern die Antwort. Ein technischer Lösungsansatz kann die Verwendung von hochverfügbarer Hardware sein. Damit ist das Problem jedoch nicht mit absoluter Sicherheit gelöst. Microservices versuchen solche Ausfälle mittels Kompensation zu umfahren. Dazu muss jedoch die Qualität der bereitgestellten Dienstleistung eingeschränkt werden. Dies wird am folgenden Beispiel von Wolff sehr deutlich: Wenn der Geldautomat den Kontostand des Kunden nicht mehr abfragen kann, gibt es zwei Möglichkeiten. Der Geldautomat kann die Auszahlung verweigern. Das ist zwar sicher, verärgert aber den Kunden und senkt den Umsatz. Oder das Geld wird bis zu einer Obergrenze ausgezahlt. Letztendlich ist die Entscheidung für eine Lösung fachlicher Natur. Bei diesen Problemen wird die Schwelle von einem technischen zu einem fachlichen Problem überschritten.[57]

Technologie Pluralismus

Die technologische Wahlfreiheit (siehe Kapitel 2.7.1 "Technologische Wahlfreiheit") von Microservices ist Segen und Fluch zugleich. Mit jeder eingeführten Technologie steigt die Komplexität des Gesamtsystems, bis sie nicht mehr von einem Entwickler oder einem Team beherrschbar ist. Im Kontext von Microservices ist dies jedoch meist nicht notwendig, da ein Team nur seinen Service verstehen muss. Wenn dieses Verständnis jedoch aus auch nur einer von vielen Perspektive erforderlich ist, wird es zu einem Problem (siehe Kapitel 2.5.2 "Mikro-/Makroarchitektur"). Zu nennen wären hier z.B. Kosten, rechtliche Risiken oder Betriebsaspekte. Wenn niemand das Gesamtbild überblickt, wird es ohne eine gemeinsame technische Basis praktisch unmöglich sein eine Vielzahl an unterschiedlichen Technologien zu betreiben.[58]

[57] vgl. Wolff, *Microservices: Grundlagen flexibler Softwarearchitekturen*, S. 76 f.
[58] vgl. ebd., S. 77.

2.7.4 Architektonische Herausforderungen

Architektonische Herausforderungen finden auf einer höheren Abstraktionsebene statt. Die Probleme betreffen oftmals die Gesamtheit aller Services, der Beziehung von Architektur und Organisation sowie querschnittliche Aspekte.

Überblick bewahren

Um einen Überblick über alle Module eines Deployment-Monolith und deren Beziehungen zu bekommen, können verschiedene Werkzeuge eingesetzt werden. Diese lesen den Quelltext ein und generieren Schaubilder, die Module und ihre Beziehungen visualisieren. So ist es möglich die aktuelle Architektur gegen die geplante abzugleichen und ggf. Änderungen vorzunehmen. Solche Werkzeuge für Microservices fehlen jedoch noch.[59]

Wechselbeziehung von Architektur und Organisation

Microservices machen sich das Gesetz von Conway (siehe Kapitel 2.5.1 "Fachliche Teams") zu Nutze und stellen die Organisation so auf, dass sie sich vorteilhaft auf die Architektur auswirkt oder eine Microservice-Architektur überhaupt erst ermöglicht. Die Umstellung der Architektur kann also eine Umstellung der Organisation nach sich ziehen. Diese sind nach aussage von Experten (siehe Kapitel 3.5.1 "Stefan Toth") oftmals die Problematischsten, da die Unternehmenspolitik eine tragende Rolle spielt. Laut Conway ist dies aber nicht nur ein Problem von Microservices, sondern betrifft alle Projekte gleichermaßen. Unternehmen sind sich diesem Gesetz jedoch oft nicht bewusst und können daher die organisatorischen Probleme bei Änderungen der Architektur nicht abschätzen.[60]

[59] vgl. Wolff, *Microservices: Grundlagen flexibler Softwarearchitekturen*, S. 77.
[60] vgl. ebd., S. 78.

Fachlicher Schnitt

Der fachliche Schnitt von Microservices hat signifikante Auswirkungen auf die Qualität der Software, die Organisation, die unabhängige Arbeit der Teams und somit auf die Produktivität. Ein falscher Schnitt kann zur Folge haben, dass Stories oder Features sich überwiegend auf mehr als einen Service beziehen, daraus resultiert ein hoher Kommunikationsaufwand der Teams und zwangsläufig koordinierte Deployments - das Gegenteil der Ziele von Microservice-Architekturen.[61]

Restrukturierung

Restrukturierung (Refactoring) bezeichnet in der Software-Entwicklung Strukturverbesserung von Quelltext unter Beibehaltung des beobachtbaren Programmverhaltens.[62] Mittels zahlreicher Werkzeuge und modernen integrated blas (IDEs) ist das Verschieben von Funktionalität eines Moduls in ein anderes besonders einfach möglich. Bei der Restrukturierung von Microservices wirkt sich die starke Modularisierung jedoch negativ auf den Arbeitsaufwand aus. Das Verschieben von Businesslogik aus einer Deployment-Einheit in eine wird nicht durch Werkzeuge unterstützt. Dies kann z.B. aufgrund eines falschen fachlichen Schnittes erforderlich sein. Unter Umständen müssen Schnittstellen angepasst werden, verwendete Bibliotheken der Services sind nicht kompatibel oder die Programmiersprachen unterscheiden sich.[63]

Evolutionäre Architektur

Der erste Entwurf einer Softwarearchitektur ist oftmals nicht optimal. Im Laufe des Projekts lernt das Team die Domäne und ihre Anforderungen besser kennen. Daraus resultiert, dass die nachfolgenden Entwürfe besser werden. *Innerhalb* der Services bietet eine Microservice-Architektur den Vorteil, dass diese besonders leicht änderbar ist. Anders verhält es sich mit der Architektur des *Gesamtsystems*. Setzt man auf eine suboptimale Makroarchitektur (siehe Kapitel 2.5.2 "Mikro-/Makroarchitektur"), werden Services falsch geschnitten oder wirken sich

[61] vgl. Wolff, *Microservices: Grundlagen flexibler Softwarearchitekturen*, S. 78.
[62] vgl. Steinacker, *Refactoring*.
[63] vgl. Wolff, *Microservices: Grundlagen flexibler Softwarearchitekturen*, S. 78 f.

die Kommunikationswege der Services nachteilig aus, sind Änderungen am Zusammenspiel der Services nur schwer umsetzbar.[64]

2.7.5 Betriebliche Herausforderungen

Aufgrund der Bereitstellung vieler kleiner Services ergeben sich ganz besonders Herausforderungen für den Betrieb des Systems.

Infrastruktur

Eine wesentliche Herausforderung für Microservice-Architekturen ist der Aufbau einer Infrastruktur, die es ermöglicht eine Vielzahl an Services bereitzustellen und zu betreiben. Um die Services voneinander zu isolieren und technologische Wahlfreiheit zu gewährleisten, sollte jeder Microservice idealerweise auf einem eigenen Host betrieben werden. Die dafür notwendige Anzahl an Systemen lässt sich mit Hardware-Servern nicht bewältigen. Es muss also eine Infrastruktur geschaffen werden, mit der jeder Service mittels einer dedizierten Continuous-Delivery-Pipeline in einer virtuellen Maschine bereitgestellt werden kann.

Die Herausforderung besteht demnach darin eine Infrastruktur zu schaffen, die eine Vielzahl an virtuellen Maschinen erzeugt und startet, um die Services bereitzustellen. Diese sollten aber auch wieder heruntergefahren werden, um die nicht benötigten Ressourcen freigeben zu können. Außerdem müssen eine große Anzahl an Continuous-Delivery-Pipelines zur automatisierten Inbetriebnahme jedes Service erzeugt und gewartet werden. Dies bedeutet auch, dass es entsprechende Testumgebungen und Automatisierungsscripte geben muss.[65]

[64] vgl. Wolff, *Microservices: Grundlagen flexibler Softwarearchitekturen*, S. 79.
[65] vgl. ebd., S. 80.

Monitoring

Eine weitere Herausforderung für den Betrieb von Microservices stellt das Monitoring dar. Bei einem Deployment-Monolithen ist es üblich, dass sich der Administrator auf dem System einloggt und das Problem mit entsprechenden Werkzeugen untersucht. Dazu werden unter anderem Log-Dateien und Informationen über die Auslastung des Servers ausgelesen. Bei einem verteilten System ist dies jedoch nicht mehr so einfach möglich, denn dann muss jeder Service auf jedem Server untersucht werden. Es ist also ein zentrales Monitoring aller Services nötig, um effektiv auf Probleme reagieren zu können. Dazu reicht es nicht aus, nur Festplattenzugriffe, Netzwerkauslastung und Log-Dateien zu überwachen. Anwendungsmetriken sollten den Einblick in den Service ermöglichen, um diesen optimieren zu können.[66]

Neben dem Überwachen der Log-Dateien ist es wichtig, kaskadierende Serviceaufrufe zu überwachen. Wird eine Anfrage von einem Service entgegengenommen und an N andere weitergegeben, ist es für die Fehleranalyse von großem Vorteil diese nachvollziehen zu können. Das gleiche gilt bei Programmfehlern für den Stacktrace, er sollte über alle Services hinweg bis zum Ursprung des Problems reichen.

2.7.6 Weitere Herausforderungen

Neben den bis hierher vorgestellten Herausforderungen gibt es noch weitere. Zu nennen sind u.a. verteilte Transaktionen, Datenpersistenz über Servicegrenzen hinweg, Versionierung der APIs, Service-Discovery und Security. Außerdem erfordert die Entwicklung von Microservices fähige Entwickler, was zu einem Problem im Bereich des Recruiting werden kann. Aufgrund des Umfangs und des zeitlichen Rahmens können diese Herausforderungen nicht näher untersucht werden.

[66] vgl. Wolff, *Microservices: Grundlagen flexibler Softwarearchitekturen*, S. 80.

2.8 Fazit

Microservices sind ein Modularisierungsansatz mit dem Ziel der losen Kopplung und hochgradiger Geschlossenheit der Komponenten. Dazu wird das System in einzeln deploybare Services zerlegt. Hauptaugenmerk ist dabei die fachliche Orientierung der Komponenten. Dieses Architekturparadigma macht sich das Gesetz von Conway zu Nutze, dieses besagt, dass die Softwarearchitektur den Kommunikationswegen der Organisation entsprechen wird. Im Umkehrschluss haben Microservices einen großen Einfluss auf die Kommunikationswegen der Teams und somit auf die Organisationsstruktur.

Aufgrund der unscharfen Definition von Microservices, divergenten Ausgangslagen sowie unterschiedlich oder abweichend gewichteten Projektzielen, haben sich verschiedene Varianten von Microservices entwickelt. Jede dieser Varianten hat unterschiedliche Vorteile, die sie für den Einsatz in einem bestimmten Kontext verschieden attraktiv machen.

Die technische Unabhängigkeit der Services hat zur Folge, dass die Teams weitestgehend autark arbeiten können. Es ist nur wenig zentrale Koordination notwendig. Die technische und organisatorische Agilität macht diesen Architekturansatz besonders attraktiv. Damit die vorgestellten Vorteile erreicht werden können, gilt es Herausforderungen aus verschiedenen Bereichen zu bewältigen. Die technischen Herausforderungen sind hauptsächlich eine Folge davon, dass es sich bei Microservices um ein verteiltes System handelt. Die größten Schwierigkeiten liegen jedoch im fachlichen Schnitt der Komponenten und der Infrastruktur.

Kapitel 3

Experteninterviews

Zur Entwicklung einer Heuristik zur Gebrauchstauglichkeit von Microservice-Architekturen darf auf die in realen Projekten gewonnenen Erfahrungen, Eindrücke, Einflussfaktoren und Resultate nicht verzichtet werden. Um Gemeinsamkeiten des Entscheidungsprozesses in Bezug zu Microservice-Architekturen aufzudecken und daraus eine Heuristik ableiten zu können, sind Befragungen mehrerer Experten durchgeführt worden. Aus diesem Grund wurden Interviews durchgeführt. Sie fokussieren aktuelle Microservice-Projekte, organisatorische und fachliche Aspekte, Vor- und Nachteile dieser, sowie den Entscheidungsprozess und dessen Einflussfaktoren.

3.1 Einleitung

Zur Theoriegewinnung wird eine Sonderform des qualitativen Interviews, das leitfadengestützte Experteninterview, angewandt. Anders als bei anderen Formen des Interviews, stellt das Experteninterview das Wissen des Experten, nicht die Person als solche, in den Mittelpunkt des Interviews.[1] Der Fokus liegt darauf, das Wissen der Experten zu einem bestimmten Untersuchungsbereich abzufragen, um daraus Rückschlüsse ziehen zu können. In diesem Kontext

[1] vgl. Meuser und Nagel, „ExpertInneninterviews - vielfach erprobt, wenig bedacht : ein Beitrag zur qualitativen Methodendiskussion“, S. 445 f.

werden Personen, die über spezifisches Fachwissen verfügen, als Experten bezeichnet und vom Forschenden selbst zu solchen ernannt.[2]

Das leitfadengestützte Experteninterview erlaubt eine strukturierte Durchführung des Interviews, bietet jedoch die Flexibilität, während des Interviews auf interessante Aspekte des Gespräches eingehen zu können. Aufgrund der nicht bindenden Reihenfolge und Formulierung der Fragen ist es möglich, eine natürliche Gesprächsatmosphäre zu schaffen, in der der Experte möglichst frei sprechen kann.[3] Trotz dieser Freiheit lassen sich die Aussagen der befragten Experten miteinander vergleichen, da der Leitfaden dazu führt, dass inhaltlich die gleichen Fragen gestellt werden.

Die Auswertung und Durchführung von Leitfadengesprächen stellt wesentlich höhere Anforderungen an den Forscher als standardisierte Fragebögen mit geschlossenen Fragen. Zum einen werden Forschungsfragen teilweise während des Gesprächs zu Interviewfragen formuliert. Zum anderen existieren aufgrund der explorativen Natur dieses Vorgehens nur wenige Antwortvorgaben. Dies führt dazu, dass sich die Dokumentation und Auswertung ungleich aufwendiger und komplexer gestaltet.

3.2 Expertenauswahl

Die Definition von Experten nach Meuser und Nagel führte zur Auswahl der Interviewpartner. Experte ist demnach ein Person welche "... in irgendeiner Weise Verantwortung trägt für den Entwurf, die Implementierung oder die Kontrolle einer Problemlösung oder wer über einen privilegierten Zugang zu Informationen über Personengruppen oder Entscheidungsprozesse verfügt." Weiterhin wurde bei der Auswahl der Experten Wert daraufgelegt, dass diese ein möglichst breites Spektrum an Erfahrungen auf dem Gebiet von Microservices und Softwarearchitektur aufweisen. Diese Voraussetzungen führen dazu, dass ein möglichst umfassendes Verständnis für die Beweggründe für oder gegen eine Microservice-Architektur entwickelt werden kann.

Die Interviews wurden mit den folgenden drei Experten durchgeführt:

[2] vgl. Meuser und Nagel, „ExpertInneninterviews - vielfach erprobt, wenig bedacht : ein Beitrag zur qualitativen Methodendiskussion“, S. 443.

[3] vgl. ebd., S. 448 f.

Stefan Toth arbeitet als Entwickler, Softwarearchitekt und Berater bei der embarc GmbH. Seine Schwerpunkte liegen in der Konzeption und der Bewertung mittlerer bis großer Softwarelösungen sowie der Verbindung dieser Themen zu agilen Vorgehen. Er ist Autor zahlreicher Artikel und des Buchs „Vorgehensmuster für Softwarearchitektur“.[4]

Eberhard Wolff ist Fellow bei innoQ und arbeitet seit mehr als fünfzehn Jahren als Architekt und Berater, oft an der Schnittstelle zwischen Business und Technologie. Er ist Autor zahlreicher Artikel und Bücher, u.a. zu Continuous Delivery und Microservices und trägt regelmäßig als Sprecher auf internationalen Konferenzen vor. Sein technologischer Schwerpunkt sind moderne Architektur- und Entwicklungsansätze wie Cloud, Continuous Delivery, DevOps, Microservices und NoSQL.[5]

Peter Böhm ist leitender IT-Architekt bei Barmenia Versicherungen mit langjähriger Erfahrung im Bereich der Softwarearchitektur, der Automatisierung und der Komplettmodernisierung von gesetzten IT-Landschafen. Außerdem war er Sprecher auf den Softwareforen Leipzig.[6]

3.3 Leitfragen

Der entworfene Leitfaden besteht aus Schlüsselfragen, die mit einer Reihe an Eventualfragen vertieft werden können. Auf diese Weise konnte eine freie Gesprächsatmosphäre geschaffen werden, deren Verlauf sich jedoch an den Schlüsselfragen orientiert. So wird die Gefahr, dass "... die Fragen des Leitfadens der Reihe nach ‚abgehakt’ werden, ohne dass dem Befragten Raum für seine (möglicherweise auch zusätzlichen) Themen und die Entfaltung seiner Relevanzstrukturen gelassen wird“[7] umgangen. Dennoch führt dies zu einer Vergleichbarkeit der Aussagen der Interviewpartner.

Die Leitfragen zielen darauf ab, die Projekterfahrungen der Experten abzurufen. Aufgrund der limitierten Zeit, die für die Durchführung des Interviews zur Verfügung stand, thematisierte das

[4] *Stefan Toth, Autor bei JAXenter*.
[5] *Eberhard Wolff, Autor bei JAXenter*.
[6] vgl. *Kunlaboro – operatives DevOps bei den Barmenia-Versicherungen*.
[7] Friebertshäuser und Prengel, *Handbuch Qualitative Forschungsmethoden in der Erziehungswissenschaft.* S. 377.

Interview jeweils ein oder zwei Projekte, die der Experte durchgeführt hat. Dank dieser Abgrenzung konnte sich der Experte auf konkrete Ereignisse und Beobachtungen stützen und wurde nicht dazu verleitet, während des Interviews Theorien entwickeln zu müssen.

Allen Experten wurden die folgenden Schlüsselfragen gestellt:

1. Bei welchem Projekt haben Sie zuletzt Microservices eingeführt?
2. Wie ist das Team organisiert?
3. Welche Technologien wurden eingesetzt?
4. Wie wird die Software betrieben?
5. Wie sind Sie zu der Entscheidung gekommen, dass eine Microservice-Architektur angewandt werden soll?
6. Welche Vorteile, die Microservices mit sich bringen, sind für dieses Projekt die größten und warum?
7. Welche Herausforderungen, die Microservices mit sich bringen, sind für dieses Projekt die größten und warum?
8. Wenn Sie das Projekt erneut beginnen könnten, was würden Sie jetzt anders bzw. gleich angehen?
9. Möchten Sie abschließend noch Aspekte nennen, die aus Ihrer Sicht im Zusammenhang mit der Entscheidungsfindung für bzw. gegen Microservices betrachtet werden sollten?

Der vollständige Leitfaden mit einer verschriftlichten Einleitung, allen Schlüssel- und Eventualfragen, ist dem Anhang A "Leitfaden für Experteninterviews" zu entnehmen.

3.4 Durchführung

Die Interviews wurden nach Absprache telefonisch am 15.08.2016 mit Herrn Toth und am 22.08.2016 mit Herrn Wolff durchgeführt. Das Interview mit Herrn Böhm wurde am 24.08.2016 in der Zentrale der Barmenia Versicherungen durchgeführt. Nach Zustimmung des Experten wurden die Interviews zur späteren Auswertung und Verwendung in dieser Arbeit digital aufgenommen. Die geplante Gesamtdauer der Interviews betrug 90 Minuten, bestehend aus einer 30-minütigen Einführung und dem 60-minütigen Hauptteil.

Aufgrund des Formats der Interviews und des sich sehr stark unterscheidenden Projektkontextes, konnte kein vollständig vergleichbarer Datenbestand erhoben werden. Jedoch gewährt gerade dieser Umstand einen tiefen Einblick in die Praxis der Experten und der sich stark unterscheidenden Einsatzgebiete und Ziele von Microservice-Architekturen.

3.5 Zusammenfassungen

Nach Durchführung der Interviews wurden die digitalen Aufnahmen paraphrasiert. Die Meinungen, Urteile, Beobachtungen, Deutungen der Experten wurden textgetreu und in eigenen Worten verschriftlicht. Weiter wurden die Aussagen thematischen Einheiten, den Schlüsselfragen, zugeordnet.[8] Es folgen Zusammenfassungen der paraphrasierten Interviews.

3.5.1 Stefan Toth

Projektkontext

Herr Toth unterstützte ein Unternehmen aus der Logistik-Branche, das Lagerhäuser mit Geräten und Software ausrüstet. Gegenstand des Projektes war eine zentrale Software, die große Mengen an Betriebs- und Verkaufsdaten verarbeitet, um festzustellen wo, welches Equipment genutzt wird, wie sich die Abnutzung verhält und wann Wartungen fällig sind. Dazu wurden Live-Daten aus den Lagerhäusern mit SAP-Daten abgeglichen. Im Zentrum stand die Verwaltung großer Flotten von Flurförderfahrzeugen und anderen Gerätschaften, die Verarbeitungsgeschwindigkeit von großen Datenmengen sowie die Möglichkeit neue Schnittstellen in kurzer Zeit anbinden zu können. Zusätzlich sollten neue Technologien nicht mehr nach dem Big-Bang-Prinzip eingeführt werden.

[8] vgl. Meuser und Nagel, „ExpertInneninterviews - vielfach erprobt, wenig bedacht : ein Beitrag zur qualitativen Methodendiskussion“, S. 456 f.

Organisation

Das Team bestand aus sieben bis acht Entwicklern, die ein agiles Vorgehensmodell angewandt hat. Es gab ein Backlog, das durch einen Product Owner befüllt wurde. Außerdem gab es eine Umgebung, in die öfter deployt wurde. In Richtung des Kunden geschah dies jedoch nur alle vier Monate. Ein Lead-Entwickler stellte die Schnittstelle zum Management dar. Rollen wie Operations, UI- oder Backendentwickler waren keine dedizierten Rollen, es gab jedoch Kollegen, die sich auf diesen Gebieten besser auskannten als andere. Nach der Umstrukturierung der Software haben ein bis zwei Entwickler die Verantwortung für einen Service übernommen. Die Verteilung der Verantwortlichkeiten begründete sich meistens auf dem Fachwissen der Entwickler. Ein Entwickler, der bereits für einen bestimmten Teil des Systems verantwortlich gewesen ist, hat auch die Verantwortung für den entsprechenden Microservice übertragen bekommen.

Technologien

Aufgrund der Historie, des Know-Hows und politischer Gründe, wurde PHP eingesetzt. Dies war eine Herausforderung, da in diesem Bereich noch nicht so viele Werkzeuge angeboten werden. Aus diesem Grund wurden im Bereich der Datenbank und Kommunikation, Technologien aus dem Java Umfeld integriert. Dazu zählen Apache ActiveMQ, Apache Kafka und Elasticsearch. Außerdem Konzepte wie Rest-Calls und Service-Discovery eingeführt. Die bereits genutzten Technologien im Front- und Backend wurden größtenteils übernommen. Das Team hatte bisher keine Erfahrungen im Microservice-Umfeld sammeln können, daher wurde der Lead-Entwickler damit beauftragt, das Wissen mittels Pair Programming bei der Erstellung von Services in das Team zu tragen.

Entscheidungsfindung

Die Idee Microservices zu nutzen, entstand in dem Team selbst. Herr Toth wurde damit beauftragt, den ersten Architekturentwurf der Entwickler zu prüfen. Dieser war aufgrund mangelnder Erfahrung jedoch mit vielen versteckten Nachteilen bespickt. Nicht alle Qualitätsziele hätten

erreicht werden können. Das Management war von dem Aufwand überrascht, den man im Bereich von Discovery, Security und dem Schnitt investieren musste. Aufgrund der Qualitätsziele erwies sich die Umsetzung einer Microservice-Architektur jedoch durchaus sinnvoll.

Laut Herr Toth ist jedoch in anderen Projekten das größte Problem, dass man auf DevOps und CD umstellen muss, um die Vorteile von Microservices voll ausnutzen zu können. Viele Unternehmen wollen darauf jedoch nicht warten, daher geht man oft die Entkopplung der Architektur und die Umstellung auf diese Prinzipien gleichzeitig an. Man nimmt einen anfänglichen Mehraufwand für Koordination und Kommunikation in Kauf, um diesen Aufwand dann im Laufe der Zeit zu verringern.

Vorteile

Viele der Ziele, die eine Microservice-Architektur zu erreichen versucht, sind langfristig. Es ist jedoch noch zu früh, um beurteilen zu können, ob diese erreicht werden oder worden sind.

Herausforderungen

Große Herausforderungen ergeben sich aufgrund der gewählten Programmiersprache PHP. Der Umstieg auf eine andere Plattform war für den Auftraggeber jedoch kein alternative, denn dies würde mit einer Umschulung aller Entwickler und Produktionsstillstand einhergehen. Zudem waren Windows-Server im Einsatz, die zu dieser Zeit noch nicht von Docker unterstützt wurden. Daraus resultiert, dass erst die Server umgestellt werden mussten, um Technologien aus dem Microservice-Umfeld nutzen zu können. In großen Projekten kommt es jedoch viel öfter zu organisatorischen bzw. politischen Problemen als zu technischen. Besonders dann, wenn es darum geht, ob Rollen erhalten bleiben oder Teams eigenverantwortlich arbeiten sollen. Diese Verantwortungs- oder Machtverschiebung kann in einigen Personen Existenzängste hervorrufen, die dann versuchen einen solchen Prozess zu blockieren.

3.5.2 Eberhard Wolff

Projektkontext

Herr Wolff begleitete ein Projekt im E-Commerce Bereich. Es handelt sich um einem Online Shop, über den digitale Assets eingestellt und verkauft werden. Das System ist an den eigentlichen Shop und an ein System zur Erstellung der Güter angebunden. Das Hauptziel bestand darin, die agile Entwicklung voranzubringen.

Organisation

Das Team besteht aus 50 bis 100 Personen und umfasst Entwickler verschiedener Dienstleister, die auch einen Betriebsschwerpunkt haben. Durch die Aufteilung der verschiedenen Dienstleister ergibt sich die Aufteilung der Teams, die nach Systemgrenzen unterteilt sind. Ein Team übernimmt die Verantwortung für Services, die entweder eine Schnittstelle zum Altsystem darstellen oder das Altsystem ersetzen. Scrum wird als agiles Vorgehensmodell genutzt.

Technologien

Im Wesentlichen handelt es sich um ein Projekt, das auf Java und relationalen Datenbanken basiert. Diese Entscheidung basiert auf einer Richtlinie, die durch das Altsystem beeinflusst wurde. Es waren keine Erfahrungen im Bereich von Microservices im Team vorhanden.

Betrieb

Im Zuge der Entwicklung der Microservices wurde der Delivery-Prozess weitestgehend automatisiert. Hierbei handelt es sich um Docker Container, die sich in der Cloud befinden. Eine Testumgebung befindet sich ebenfalls in dieser Cloud.

Entscheidungsfindung

Die Entscheidung Microservices zu nutzen, wurde bereits geklärt und durch das Management verabschiedet. Herr Wolff wurde zur Realisierung herangezogen. Der Ausgangspunkt solcher Projekte kann sich jedoch stark unterscheiden. In manchen Projekten wurde bereits die Entscheidung zugunsten von Microservices getroffen. Herr Wolff unterstützt diese Projekte bei der Umsetzung einer geeigneten Variante. In anderen Projekten wird er zu Architekturdiskussionen eingeladen, die in einer entsprechenden Entscheidung bezüglich der Zielarchitektur resultieren. Grundsätzlich lassen sich die angestrebten Ziele in organisatorische und technische unterteilen. Ausgehend von diesen Zielen werden technische Lösungen und organisatorische Maßnahmen entwickelt, die typischerweise zu einer der Varianten von Microservice-Architekturen führen. Das Hauptargument für Herrn Wolff für Microservices ist die Risikominimierung durch kleine Deployment-Einheiten. Im Gegensatz dazu steht das Big-Bang Deployment eines Monolithen. Herausforderungen können gezielt umgangen werden, indem z.B. alle Services in *einem* Application Server betrieben werden. Dies verringert die Komplexität im Betrieb, man verliert jedoch die Ausfallsicherheit. Für welche Varianten man sich auch entscheidet, es ist wichtig, begründete Architekturentscheidungen zu treffen. Ein Gesichtspunkt dieser Entscheidung ist die Kosteneffizienz.

Vorteile

Der größte Vorteil von Microservice liegt laut Wolff in der Unterstützung von agilen Prozessen, indem Releasezyklen verkürzt wurden. Außerdem kann das System dank Microservices stückweise ersetzt werden. Zur Bewertung weiterer Vorteile befindet sich das Projekt in einem zu frühen Stadium.

Herausforderungen

Der Aufbau der notwendigen Infrastruktur und der Grad der Automatisierung waren in diesem Projekt die größten Herausforderungen. Das resultiert aus einer Unterschätzung des Aufwandes zur Umstellung der Infrastruktur und sich ändernden Anforderungen im Projekt. Besonders

der organisatorische Impact und die Infrastruktur sind Gebiete denen man große Beachtung schenken muss. Beispielsweise wäre eine Platform as a Service (PaaS) Lösung in diesem Projekt interessant gewesen.

3.5.3 Peter Böhm

Projektkontext

Herr Böhm berichtete von einem Projekt der unabhängigen Versicherungsgruppe Barmenia mit dem Namen Partneranlage im Antragsprozess(PAA). Bei den Partnern handelt es sich um Vertriebspartnern wie Makler, Mehrfachagenten oder Finanzdienstleistungsunternehmen die den wesentlichen Vertriebskanal des Unternehmens darstellen. Das Ziel dieses Projektes ist ein vereinheitlichter automatisierter Antragsprozess mit zusätzlichen Prüfungen der Eingangsdaten zur Verbesserung der Datenqualität und zur Vermeidung von Dubletten im Partnersystem. Zu diesem Zweck werden die vom Partner hinterlegten Daten mittels verschiedener Regeln maschinell geprüft. Diese Prüfung kann jedoch in eine manuelle Prüfung münden. Neben den bereits erwähnten fachlichen Zielen, wurde dies Projekt zur Evaluation von Microservice-Architekturen durchgeführt. Es galt zu prüfen, wie sich Microservices in die bestehende Infrastruktur einbinden lassen und inwiefern die Barmenia dieses Paradigma für sich nutzen kann. Das Projekt ist losgelöst vom Altsystem, wurde jedoch in die bestehenden Abläufe integriert.

Organisation

Aufgrund des Evaluations-orientierten Charakters dieses Projektes, wurde es von lediglich einem Entwickler innerhalb von ca. 60 Personentagen umgesetzt. Der Entwickler ist sehr erfahren und hat ebenfalls Kenntnisse im IT-Betrieb. Das Projekt ist agil durchgeführt worden. Anstehende Arbeitspakete wurden mit dem Vorgesetzten besprochen und auf einem Kanban-Board dokumentiert. Vorschläge bezüglich den zu verwendenden Technologien konnte der Entwickler in die Architekturdiskussionen mit einbringen.

Technologien

Die Barmenia homogenisiert derzeit die verwendeten Programmiersprachen. Aus diesem Grund wird Java als Programmiersprache für alle Services genutzt. Es wird Spring Boot mit einem Tomcat Server eingesetzt, da sich diese Technologie sehr gut in die bestehende Infrastruktur einbinden lässt. Als Datenbank dient MongoDB. Zur asynchronen Kommunikation wird eine MySQL Datenbank eingesetzt. Querschnittliche Anforderungen wie Security werden mittels eines eigens von der Barmenia entwickelten Frameworks abgedeckt. Dieses wird bereits in anderen Projekten erfolgreich eingesetzt. Aufgrund des Umfangs des Projektes wurde bewusst auf Techniken wie Service-Discovery verzichtet.

Der Entwickler hatte bislang nur theoretische Erfahrungen mit Microservice-Architekturen sammeln können. Dazu studierte er die aktuelle Literatur und besuchte entsprechende Veranstaltungen. Diese Person ist es jedoch gewohnt sich in neue Themengebiete einzuarbeiten und war vorher bereits in ein großes Infrastruktur Projekt involviert.

Betrieb

Es wird eine Puppet Infrastruktur zur Provisionierung eingesetzt. Die Software wird über die CD-Pipeline in den Nexus eingestellt. Dies ist ein Repository-Manager, der für die Verteilung, Organisation und Verwaltung von unternehmensinternen Maven-Artefaken zuständig ist. Die Software muss "kontrolliert überführt" werden. Das bedeutet, dass erst nach einer manueller Freigabe einer Version, die nach dem Vier-Augen-Prinzip erfolgt, die Software bereitgestellt werden darf. Die Bereitstellung wird mittels eigen geschriebener Werkzeuge vollautomatisiert durchgeführt. Infrastructure as Code ist ein wesentlicher Bestandteil dieser Automatisierung. Die Abdeckung der maschinellen Tests ist relativ hoch, jedoch gibt es dazu keine harten Regeln. Die Verantwortung darüber hat der Projektleiter selbst.

Entscheidungsfindung

Die Barmenia kommt aus einer gestandenen Service Landschaft. Es wurde eine „implementierte“ SOA-Architektur umgesetzt, wobei keine Webservices, sondern transaktional gekoppelte Enterprise JavaBeans (EJBs) genutzt werden. Aufgrund der Automatisierung und durch den Versuch die Application Server zu wechseln, der wegen der Inkompatibilität der Security Transaktionskontexte nicht funktionierte, rückte die Entkopplung der Software vom Server in den Fokus. Die Grundidee von nicht transaktional gekoppelten Services, die asynchron kommunizieren, passt an vielen Stellen gut zu den Anforderungen der Barmenia. Denn auch Versicherungen müssen in immer kürzeren Abständen neue Funktionalität nach außen bringen. Aufgrund des vorher abgeschlossenen Projektes, mit dem Services automatisiert deployt werden können, war die Barmenia in einer guten Ausgangslage um diesen Architekturansatz zu testen.

Vorteile

In diesem Projekt geht es um die Austauschbarkeit von sich ändernden Funktionalitäten. Wenn sich Prozesse oder Regeln ändern, ist es von Nachteil wenn das System als Ganzes neu deployt werden muss und so Ausfallzeiten entstehen. Aufgrund der asynchronen Kopplung der Microservices passten sie in diesem Kontext gut zum Problem. Das Gesamtsystem arbeitet weiter, aber einzelne Services lassen sich dennoch updaten und die Reihenfolge der Prüfung kann neu organisiert werden.

Herausforderungen

Herr Böhm sieht die Probleme zum einen in der Verteilung der Daten. Sollten beispielsweise Kundendaten über X Services verteilt sein, müssen bei einem Update auch X Services davon erfahren und es durchführen. Der Aufwand wird somit potenziert.

Dies betrifft auch Updates der verwendeten Betriebssysteme. Tritt ein Fehler im Betriebssystem auf, auf dem die virtuellen Maschinen basieren. Muss mit steigender Anzahl an Services eine steigende Anzahl an Systemen aktualisiert bzw. neu konfiguriert werden. Weiter, so Herr

Böhm, wird es in Zukunft problematisch qualifiziertes Personal zu finden und Personen von den Vorteilen einer Microservice-Architektur zu überzeugen. Nach Einschätzung von Herrn Böhm denken Entwickler synchron. Dieses mentale Modell auf asynchron umzustellen, könnte ein größeres Problem werden.

3.6 Fazit

In dem von Herrn Toth vorgestellten Projekt überwiegen die technischen Vorteile einer Microservice-Architektur. Herausforderungen ergeben sich größtenteils durch die Umstellung auf DevOps und CD und sind oftmals politischer bzw. organisatorischer Natur. Um eine Entscheidung bezüglich Microservices treffen zu können, muss der Kontext ganzheitlich betrachtet werden. Die Qualitätsziele des Projektes spielen eine zentrale Rolle, jedoch darf der Aufwand für die Implementierung nicht unterschätzt werden.

Aufgrund der organisatorischen Vorteile einer Microservice-Architektur ist in dem von Herrn Wolff beschriebenen Projekt die Entscheidung für Microservices gefallen. Auch für ihn stellen die Umstellung auf DevOps-Prinzipien und CD die größte Herausforderung dar. Laut Herr Wolff können einige Probleme mittels Kompromissen entschärft werden. Solche Zugeständnisse verringern jedoch nicht nur die Komplexität eins Problems, sie wirken sich auch auf das angestrebte Ziel aus. Das Deployment vieler Services auf *einem* gemeinsamen Application Server verringert zwar die Komplexität im Betrieb, es mindert aber auch die Isolation der Services. Die Entscheidung für Microservices darf keine Bauchentscheidung oder aufgrund des aktuellen Trends getroffen werden. Architektur muss eine vernünftige Basis zur Erfüllung der Qualitätsziele finden.

Für Herrn Böhm sind Microservice keine Lösung für jedes Problem. In dem von ihm beschriebenem Projekt haben die Anforderungen sehr gut mit den Vorteilen von Microservices übereingestimmt. Auf der anderen Seite konnten Komplexitäten durch Kompromisse umgangen werden. Er betonte, dass die schweren Probleme vor allem im Bereich der Infrastruktur, Automatisierung und Organisation angesiedelt sind. Außerdem bedarf es vieler verschiedener Fertigkeiten eines Entwicklers, um den neuen Anforderungen einer verteilten Architektur gerecht zu werden. Dies erschwert das ohnehin schon herausfordernde Recruitment von geeignetem Personal.

Die Experteninterviews zeigen, dass die Entscheidung für oder gegen eine Microservice-Architektur verschiedene Dimensionen hat. Zum einen stellen die Anforderungen und Ziele des Projektes tragende Entscheidungsfaktoren dar. Die zentrale Frage, die es zu beantworten gilt, ist: Passen die Vorteile einer Architektur zu den Anforderungen des Projektes? Zum anderen müssen Architekturentscheidungen auch unter Berücksichtigung der Kosteneffizienz getroffen werden. Möglicherweise passen Anforderungen und Vorteile zusammen, dann stellten sich jedoch die Fragen: Welchen Preis gilt es zu zahlen? Ist dieser angemessen oder verhandelbar?

Kapitel 4

Methoden

Im folgenden Kapitel wird zunächst die Failure Mode and Effects Analysis (FMEA) beleuchtet. Tracy P. Omdahl beschreibt dieses Verfahren als ein "formalisiertes Verfahren zur Definition, Identifizierung und Behebung bekannter und/oder potentieller Störungen, Probleme oder Fehler in Systemen, Designs, Prozessen und/oder Diensten, bevor sie den Kunden erreichen".[1] Im Zentrum dieser Methode stehen die Wirkungsbeziehungen von Eintrittswahrscheinlichkeit, Schadenausmaß und Entdeckungswahrscheinlichkeit eines Fehlers. Um die zentrale Frage dieser Arbeit zu beantworten, ist es essenziell, die Einflussfaktoren zu erfassen und ein Verständnis für deren Wirkungsbeziehungen zu entwickeln.

Im darauffolgenden Abschnitt wird Architecture Tradeoff Analysis Method (ATAM) untersucht. "ATAM ist eine Methode zur Evaluierung von Software Architekturen relativ zu den zu erreichenden Qualitätsattributen. ATAM Evaluationen decken Risiken der Softwarearchitektur auf, die das Erreichen der Unternehmensziele hindern."[2] Die Frage, ob eine Microservice-Architektur zur Lösung eines bestehenden Problems angemessen ist, kann im weiteren Sinne als eine "auf den Kopf gestellte ATAM-Analyse" betrachtet werden. Es wird nicht die Architektur in Frage gestellt, diese steht fest. Viel mehr stellt sich die Frage: Zu welchem Problem passt das Microserviceparadigma? Aus diesem Grund können Aspekte von ATAM der Beantwortung der zentralen Fragestellung dienen.

[1] vgl. Omdahl, *Reliability, availability, and maintainability (RAM) dictionary*.
[2] Carnegie Mellon University, *Architecture Tradeoff Analysis Method*, (Übersetzung durch den Autor).

4.1 Failure Mode and Effects Analysis

Die FMEA wurde in den 1940er Jahren vom US Militär entwickelt. Die NASA nutzte dieses Modell in den 1960er Jahren zur Qualitätssicherung bei den Apollo-Projekten. Im darauffolgenden Jahrzehnt wurde die FMEA von Ford in die Automobilindustrie überführt. Ab den 1980er Jahren war sie im gesamten Industriezweig im Einsatz.[3] Zu Beginn wurde sie zur Entdeckung von Fehlern bei physikalischen Gütern oder Prozessen eingesetzt. Heute wird sie unter anderem auch zur Identifikation von Schwachstellen und unternehmensrelevanten Bedrohungen bei IT Systemem eingesetzt.[4]

4.1.1 Ziel

Die FMEA hat die Minimierung von Risiken der Produktentwicklung oder Prozessanalyse zum Ziel. Um dieses Ziel zu erreichen, müssen die folgenden Fragen beantwortet werden: "Welche Probleme könnten auftreten?", "Wie hoch ist die Wahrscheinlichkeit, dass dieses Problem eintritt?, "Wie hoch ist das Schadensausmaß?" und "Wie können diese Probleme verhindert werden?". FMEA formalisiert die Suche nach potentiellen Fehlern in Prozessen oder Produkten und bildet so die Basis für ein proaktives Risikomanagement.[5]

4.1.2 Vorgehen

Typischerweise wird die FMEA von einem interdisziplinären Team durchgeführt. Hierzu kommen Mitarbeiter eines Unternehmens mit verschiedenen Funktionen zusammen. Der Analyseprozess selbst wird anhand von Formblättern oder entsprechender Software unterstützt, die die folgenden Elemente enthalten:

- Abgrenzung des betrachtenden Elementes
- Beschreibung der Funktion
- Analyse der potenziellen Fehlerarten, Fehlerursachen und Fehlerfolgen
- Beschreibung der Kontrollmaßnahmen

[3] Ben-Daya, *Handbook of maintenance management and engineering*, vgl. S. 75.
[4] Prokein, *IT-Risikomanagement - Identifikation, Quantifizierung und wirtschaftliche Steuerung*, vgl. S. 25.
[5] Ben-Daya, *Handbook of maintenance management and engineering*, vgl. S. 75 f.

- Eine Risikoeinschätzung
- Abstellmaßnahmen

Zu jedem potentiellen Fehler erfolgt eine Bewertung des Schadensausmaßes **S** (Severity), der Eintrittswahrscheinlichkeit **O** (Occurrence) und der Entdeckungswahrscheinlichkeit des Fehlers oder dessen Folgen **D** (Detection). Das Produkt aus S und O ergibt die Risikoklassifizierung **SO**. Ein seltenes aber teures Risiko wird ähnlich einem häufigen aber billigem klassifiziert. Die Risiko-Prioritätszahl **RPN** (Risk Priority Number) bildet wiederum das Produkt aus SO und D.[6] Die RPN priorisiert die verschiedenen Risiken untereinander. Abbildung 4.1 veranschaulicht den Zusammenhang der Faktoren. Risiken, deren Entdeckungswahrscheinlichkeit niedrig ist (der Aufwand zur Entdeckung sehr hoch), werden dementsprechend niedriger priorisiert als Risiken die schnell entdeckt bzw. gelöst werden können.

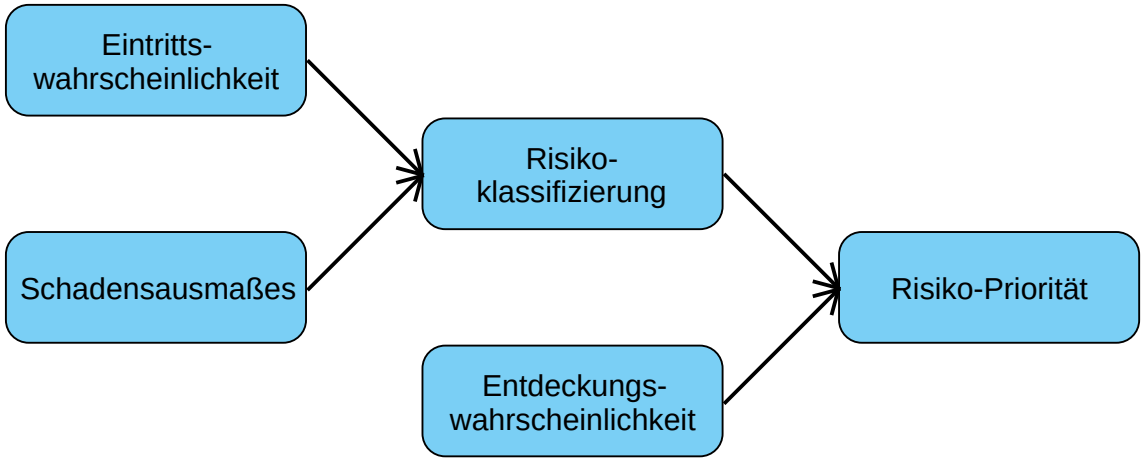

ABBILDUNG 4.1: Berechnung der Risiko-Prioritätszahl (Quelle: Bente, *Grundlagen des Strategischen IT-Managements*, Vorlesung WS 2015/16, TH Köln)

Die Analyse kann auf zwei verschiedenen Datenbeständen fußen. Die erste Möglichkeit, um Fehler identifizieren zu können, sind historische Daten. Möglicherweise wurde bereits eine Analyse für ähnliche Produkte durchgeführt. Es könnten aber auch Kundenbeschwerden oder Bug Reports herangezogen werden. Die zweite Möglichkeit ist die Identifizierung von Fehlern anhand von mathematischen Modellen, Simulationen oder Expertenwissen.[7]

[6] vgl. Bente, *Grundlagen des Strategischen IT-Managements*, S. 10 - 14.
[7] vgl. Stamatis, *Failure Mode and Effect Analysis - FMEA from Theory to Execution*, S. 11.

4.2 Architecture Tradeoff Analysis Method

Die Methode wurde von Rick Kazman am Software Engineering Institute - Carnegie Mellon University entwickelt und ist eine Weiterentwicklung der Software Architecture Analysis Method (SAAM)[8].

Diese Bewertungsmethode generiert konkrete Qualitätsaussagen (o.a. Qualitässzenarien) die sich aus den Zielen der Architektur und den Rahmenbedingungen des Projektes ableiten lassen. Spielt man die Architekturentscheidungen anhand dieser Szenarien durch, werden potentielle Risiken, Probleme und Schwächen der Entscheidung deutlich und können so mit alternativen Lösungsansätzen verglichen werden.

Szenarienbasierte Techniken zählen zu den ältesten, populärsten und meist verbreitetsten Instrumenten der Zukunftsforschung. Szenarios werden beispielsweise regelmäßig zur Planung und Vorbereitung von Unternehmensstrategien eingesetzt. [9]

Langfristige Architekturentscheidungen können dank des Zukunftsforschungs-orientierten Hintergrundes von szenarienbasierten Methoden evaluiert werden.

4.2.1 Ziel

ATAM ist eine Methode zur Identifizierung von Risiken in einem komplexen Software System. Mit Hilfe dieser Methode können Auswirkungen von Architekturentscheidungen mit Fokus auf die Qualitätsanforderungen des Projektes bewertet werden.[10] Neben der Risikoanalyse, die bereits in einem frühen Stadium eines Projektes durchgeführt werden kann, hat ATAM weitere positive Auswirkungen auf das Projekt. Bereits sehr früh in der Evaluation werden Qualitätsziele definiert und priorisiert. Diese Ziele werden oft nicht oder nur vage definiert. Szenarien stellen einen expliziten Maßstab dar, um die Zielerreichung zu messen. Außerdem wird die Dokumentation der Architektur verbessert, da sie dem Vorgehen als Basis dient. Aufgrund der intensiven Kommunikation von Stakeholdern entwickelt sich ein besseres Verständnis für die

[8] Kazman, Abowd u. a., „SAAM: A Method for Analyzing the Properties of Software Architectures“
[9] vgl. Babar und Gorton, „Experiences from Scenario-Based Architecture Evaluations with ATAM“, S. 214 f.
[10] vgl. Kazman, Klein u. a., *ATAM: Method for Architecture Evaluation*, S. 2.

Anforderungen. Oft werden durch die Evaluation bislang unbekannte Anforderungen an das System deutlich.[11]

4.2.2 Vorgehen

Eine Evaluation mittels ATAM unterteilt sich in drei Aktivitäten: Vorbereitung, Evaluation und der Präsentation der Resultate. Die Evaluation wird in acht Schritten durchgeführt, dazu kommen verschiedene Stakeholder in einem Workshop zusammen. Für die vorliegende Arbeit ist besonders die Erstellung des Quality Attribute Utility Trees von Interesse. Eine vollständige Beschreibung ist in Anhang B "Architecture Tradeoff Analysis Method - Vorgehen im Detail" zu entnehmen.

Zur Identifizierung, Priorisierung und Präzisierung der wichtigsten Qualitätsziele bedient sich ATAM einer Technik namens Utility Tree.

Die Wurzel des Baumes bildet die *utility*, die die Güte des Systems beschreibt. Auf der ersten Ebene werden die Qualitätsziele aus Schritt zwei aufgeführt, denn die Güte des Systems besteht aus ihnen. Dabei handelt es sich um unternehmerischen Einflussfaktoren, wie die wichtigsten Funktionen, technische, politische oder historische Rahmenbedingungen, Unternehmensziele und Anforderungen durch Stakeholder. Typischerweise werden Attribute wie Performance, Modifiability, Security, Usability oder Availability genannt. Die Teilnehmer des Workshops können jedoch auch eigene Ziele nennen, denn es kommt vor, dass unterschiedliche Benutzergruppen verschiedene Begriffe für die gleiche Eigenschaft nennen oder Qualitätsattribute erst in dem jeweiligen Kontext, aber nicht in allen anderen, sinnvoll sind.

Auf der dritten Ebene werden die Qualitätsziele explizit gemacht. Qualitätsziele wie Performance werden in *Data Latency* und *Transaction Throughput* zerlegt. Auf der darauffolgenden Ebene werden die Zielattribute mit Szenarien konkretisiert, um priorisiert und analysiert werden zu können. Data Latency könnte mit den Szenarien "Senkung der Dauer des Datenspeicherns auf unter 20 Millisekunden" und "Ausliefern eines 20 Frame/Sekunden Videos in Echtzeit". Diese Szenarien sind explizit genug, um getestet zu werden. In den nachfolgenden Schritten wird für jedes Szenario geprüft, wie die Architektur darauf reagiert oder ob sie das Ziel erreicht.

[11] vgl. Babar und Gorton, „Experiences from Scenario-Based Architecture Evaluations with ATAM“, S. 227.

Da auf diese Weise eine ganze Reihe an Szenarien entstehen kann, müssen diese priorisiert werden. Dies geschieht auf zwei Ebenen. Zunächst einigen sich die Entscheider darauf, wie wichtig ein Attribut ist. Hier bietet sich eine Bewertung nach hoch H (high), mittel M (medium) und gering L (low). Anschließend bewerten die Architekten wie schwer ein Ziel erreichbar ist. Szenarien welche mit H,H gekennzeichnet sind, sollten während der Analyse im Fokus stehen.[12]

Abbildung 4.2 zeigt beispielhaft den Auszug eines Utility Trees.

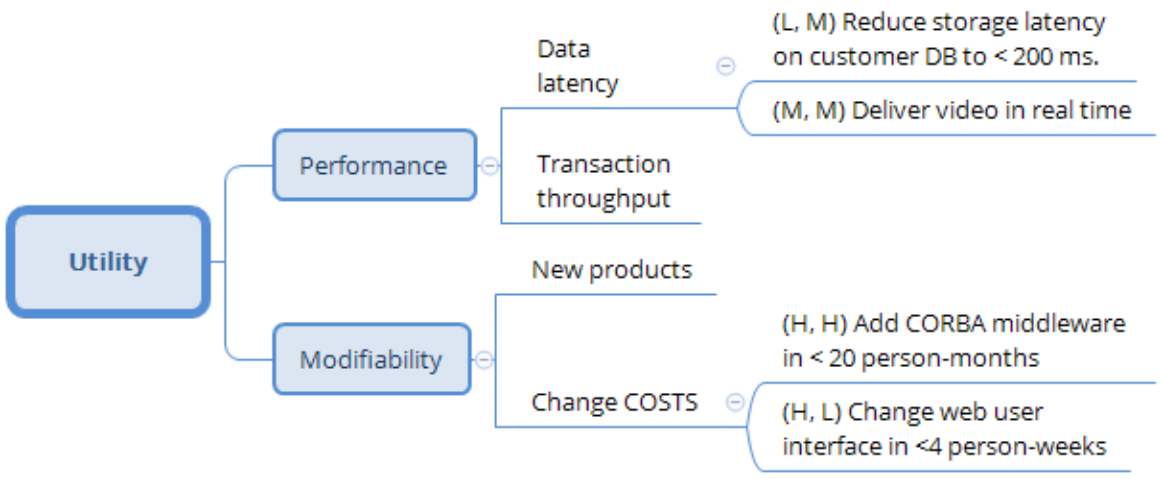

ABBILDUNG 4.2: Auszug eines möglichen Utility Trees (Quelle: Kazman, Klein u. a., *ATAM: Method for Architecture Evaluation*, S. 17)

Im nächsten Schritt begründet der Architekt oder das Team für jedes Szenario, wie die Architektur dieses unterstützt. Auf dem Weg werden alle relevanten Architekturentscheidungen dokumentiert. Das Abarbeiten der Szenarien führt zu Diskussionen in denen Risiken (Risks), Nicht-Risiken (Nonrisks), Punkte, die besonders beachtet werden müssen (Sensitivity Points) und Kompromisse (Tradeoffs) erfasst und katalogisiert werden können.[13]

[12] vgl. Kazman, Klein u. a., *ATAM: Method for Architecture Evaluation*, S. 29.
[13] vgl. ebd., S. 29 - 33.

4.3 Fazit

Im Zentrum der FMEA steht ein Wirkungsmodell welcher zur Identifizierung und Priorisierung potentieller Risiken sowie zur Entwicklung von Gegenmaßnahmen dient. Das Produkt (RPN) der verschiedene Faktoren hilft Entscheidern, den Fokus auf die Risiken zu legen, deren Beseitigung am wirtschaftlichsten ist. Zur Durchführung einer Eignungsprüfung von Microservice-Architektur im Projektkontext lässt sich diese Methode jedoch nur indirekt adaptieren. Die Entscheidung, eine Architektur zu implementieren, kann nicht nur anhand eines Produktes getroffen werden. Es benötigt viel mehr ein ganzes Spektrum an bewerteten Faktoren, wie Vor- und Nachteile unter Berücksichtigung der Projektziele sowie den entstehenden Kosten. Erst bei einer Gegenüberstellung dieser Faktoren kann eine fundierte Entscheidung getroffen werden. Dennoch trägt ein Wirkungsmodell dieser Aspekte zur Problemlösung bei. Es stellt Beziehungen zwischen den Einflussfaktoren einer Architekturentscheidung her und hilft so Entscheidungen und deren Auswirkungen festzustellen.

ATAM ist eine vielseitige Methode. Zum einen hilft sie, Qualitätsziele zu definieren, die mittels konkreter Szenarien explizit beschrieben werden. Dies hilft dem Architekten die Bedürfnisse der Stakeholder zu verstehen und so eine zielgerichtete Architektur zu entwerfen. Zum anderen werden Architekturentscheidungen kritisch hinterfragt und Lösungsalternativen miteinander verglichen. Zur Durchführung einer Eignungsprüfung von Microservice-Architektur im Projektkontext ist diese Methode jedoch nur bedingt von Nutzen. Zum einen sollen keine Implementierungsdetails bzw. angewandte Pattern miteinander vergleichen werden. Im Kontext von Microservices ist dies Teil der Mikroarchitektur. Die Entscheidung, ob eine Microservice-Architektur umgesetzt werden soll, muss auf einer höheren Abstraktionsebene getroffen werden. Zum anderen hat ATAM das Ziel Qualitätsziele zu erfassen, um für diese eine passende Architektur zu entwickeln. Bei der Entscheidung für bzw. gegen Microservices wird die Frage jedoch auf den Kopf gestellt. Es gilt herausfinden, ob das Projekt für eine bestehende(geplante) Architektur geeignet ist.

Kapitel 5

Ergebnisse

In diesem Kapitel werden die gewonnenen Erkenntnisse aus Literatur, untersuchten Methoden und Experteninterviews vorgestellt. In Abschnitt 5.1 "Allgemeines Wirkungsmodell" wird das erarbeitete Wirkungsmodell einer Architekturentscheidung dargestellt. Dieses Modell kann auf Entscheidungen, unabhängig der vorliegenden oder geplanten Architektur, angewandt werden. Es definiert grundlegende Einflussfaktoren einer Architekturentscheidung sowie deren Wirkungsbeziehungen.

Aufbauend auf diesem Modell wurde ein spezifisches Wirkungsmodell abgeleitet. Abschnitt 5.2 "Spezifisches Wirkungsmodell" beleuchtet vier Probleme und beschreibt, wie spezifische Mehrwerte einer Microservice-Architekturen diese lösen. Es folgt eine Beschreibung der zugehörigen Komplexitäten und Architekturziele.

In Abschnitt 5.3 "Detaillierte Wirkungsmechanismen" werden die zuvor vorgestellten spezifischen Mehrwerte und Komplexitäten im Detail betrachtet. Es wird erläutert, aus welchen Vorteilen und Herausforderungen sich diese ergeben. Außerdem werden mögliche Kompromisse und Projektsituation beschrieben, mit denen Einfluss auf die Komplexität genommen werden kann.

Abgeschlossen wir das Kapitel mit einem praktischen Anwendungsbeispiel im Kontext eines fiktiven Projektes.

Es ist zu beachten, dass es sich bei den hier vorgestellten Konzepten um Modelle der Realität handelt. Aus diesem Grund wurden sie in der Form vereinfacht, dass nicht alle Wirkungsbeziehungen erfasst wurden. Es sei darauf hingewiesen, dass ganz besonders das spezifische Wirkungsmodell und das detaillierte Wirkungsmodell keinen Anspruch auf Vollständigkeit erheben. Wie der Abschnitt 5.4 zeigen wird, ist die Erweiterung des Modells vorgesehen und erwünscht.

5.1 Allgemeines Wirkungsmodell

Um die zentrale Fragestellung dieser Arbeit beantworten zu können, ob die Einführung von Microservices in einem konkreten Projektkontext sinnvoll ist, wurde ein allgemeines Wirkungsmodell zu Architekturentscheidungen entwickelt. Dieses Modell stellt in der höchsten Abstraktionsschicht das systematische Zusammenspiel von Einflussfaktoren einer Architekturentscheidung aus zwei Sichten dar. Zum einen beschreibt die Eignung, welche Architekturziele es zu erreichen gilt und wie sie von einer Architektur unterstützt werden. Zum anderen beleuchtet die Effizienz, wie Einfluss auf die Komplexität genommen werden kann. Abbildung 5.1 "Darstellung des entwickelten allgemeinen Wirkungsmodells" stellt diese zwei Sichten, deren Faktoren und Beziehungen dar.

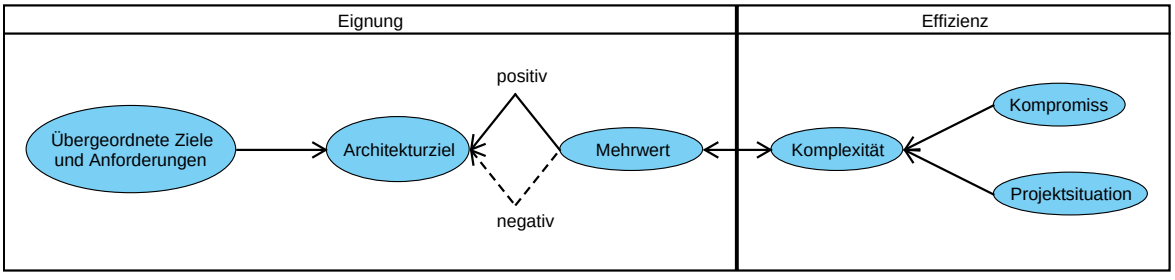

ABBILDUNG 5.1: Darstellung des entwickelten allgemeinen Wirkungsmodells

5.1.1 Begriffsdefinition

In dem folgenden Abschnitt werden die wichtigsten Einflussfaktoren des allgemeinen Wirkungsmodells definiert und ausgeführt.

5.1.1.1 Architekturziel

Architekturziel werden oftmals auch als auch nicht-funktionale Anforderungen oder Qualitätsziele bezeichnet. Sie beschreiben einen Zustand, den die Software erreicht oder eine Eigenschaft, die sie erfüllen soll. Die ISO 25010 fasst Qualitätsmerkmale für Software in einer Norm zusammen. Nach dieser Norm können Merkmale wie Fehlertoleranz, Robustheit oder Anpassbarkeit als Architekturziele dienen.

Merkmale wie Time-to-Market und Langlebigkeit umfasst diese Norm nicht, dennoch sind es valide Architekturziele. Es ist wichtig, die Ziele genau zu spezifizieren, denn innerhalb eines Unternehmens oder Projektkontextes kommt es immer wieder vor, dass Stakeholder unterschiedliche Begriffe für das gleiche Ziel verwenden. ATAM nutzt Szenarien zur Formalisierung. Bei diesem Vorgehen steht unter anderem die Messbarkeit im Vordergrund.

5.1.1.2 Mehrwert

In dieser Arbeit beschreibt der Mehrwert einer Architektur, einen übergeordneten Vorteil gegenüber anderen Architekturen. Beispielsweise haben Microservice-Architekturen den Vorteil, dass für jeden Service eine andere Technologie gewählt werden kann. Der Mehrwert spiegelt sich jedoch in einer erhöhten Flexibilität der Architektur wieder. Die Summe mehrerer Vorteile kann zu einem Mehrwert werden.

Einen Mehrwert zeichnet weiterhin aus, dass der Ertrag die Leistung übersteigt. Ob dies für einen Mehrwert zutrifft, muss individuell geprüft werden. Der Wert eines Ertrags und der Aufwand, um diesen zu einzufahren, können sich stark unterscheiden. Der Wert von technologischer Wahlfreiheit kann für ein Unternehmen Wertvoller sein als für andere. Das Gleiche gilt für den Aufwand, der investiert werden muss. Unter bestimmten Gegebenheiten fällt dieser höher oder geringer aus.

5.1.1.3 Komplexität

Eine Komplexität wird als die Summe aller Herausforderungen definiert, die es zu überwältigen oder handhaben gilt, um einen Mehrwert zu erreichen. Beispielsweise muss ein Fahrradfahrer

die Komplexität des Gleichgewichts handhaben, um nicht vom Fahrrad zu stürzen. Dazu muss er unter anderem sein Gewicht verlagern und Arm- sowie Beinbewegungen koordinieren.

5.1.1.4 Kompromiss

Ein Kompromiss ist ein beidseitiges Zugeständnis, das eingegangen wird. Auf der einen Seite mindern Kompromisse die Komplexität, indem Herausforderungen abgeschwächt oder sogar umgangen werden. Auf der anderen Seite mindern sie jedoch auch den Mehrwert, der betroffenen Komplexität (siehe Abschnitt 5.1.3 "Effizienz").

5.1.2 Eignung

Im Zentrum der Eignungsbestimmung eines Architekturstils stehen die Architekturziele, die erreicht werden sollen. Zum einen haben alle befragten Experten wiederholt darauf hingewiesen, dass Architektur kein Selbstzweck ist, viel mehr führen Ziele und Anforderungen zu einer adäquaten Lösung (siehe Kapitel 3 "Experteninterviews"). Aber auch Evaluierungsmethoden, wie ATAM (siehe Kapitel 4.2 "Architecture Tradeoff Analysis Method") und SAAM bauen auf einer Definition von Architekturzielen auf.

Architekturziele ergeben sich meist aus übergeordneten Zielen oder Beschränkungen. Die folgende Liste gibt einen Überblick der möglichen Quellen, aus denen Architekturziele abgeleitet werden können:

- wichtige funktionale Anforderungen
- technische, betriebswirtschaftliche, ökonomische, politische oder gesetzliche Einschränkungen
- Unternehmensziele und dem Kontext
- wichtige Stakeholder
- architektonische Einflussfaktoren (die wichtigsten Qualitätsattribute)

Der Einbezug von Management und Stakeholdern ist demnach wichtiger Bestandteil einer Zieldefinition.[1]

[1] vgl. Kazman, Klein u. a., *ATAM: Method for Architecture Evaluation*, S. 26.

Die Definition von Zielen und Anforderungen ist eine wichtige Voraussetzung, um eine adäquate Lösung zu finden. Das Modell zeigt, dass nicht jedes Architekturziel von den Mehrwerten eines Architekturstils unterstützt wird. Sie können sich positiv, negativ oder überhaupt nicht auf Architekturziele auswirken. Die Eignungsprüfung stützt sich auf einer möglichst hohen Übereinstimmung von angestrebten Architekturzielen, die von Mehrwerten eins Architekturstils unterstützt werden. Beeinflussen die Mehrwerte eines Architekturstils die erhobenen Ziele überwiegend positiv, fällt auch die Eignungsprüfung positiv aus.

5.1.3 Effizienz

In dieser Arbeit wird Effizienz als Relation zwischen Kosten und Nutzen definiert. Die Kosten, im Kontext der Effizienzbestimmung eines Architekturstils, entstehen in Form von Komplexitäten. Dies bedeutet, dass ein Mehrwert nicht kostenlos entsteht, er wird direkt oder indirekt mittels einer Komplexität bezahlt. Beispielsweise bieten Microservice-Architekturen den Mehrwert, dass fachliche Komponenten nach Bedarf skaliert werden können. Diese Skalierbarkeit wird jedoch mit der Komplexität eines verteilten Systems erreicht. Im Gegensatz dazu bieten monolithische Architekturen den Mehrwert, dass bei geringer Komplexität des Projektes die Produktivität des Teams tendenziell höher ist, als die eines Teams, das eine Microservice-Architektur umsetzt. Solche Deployment-Monolithen sind mit einer geringeren technischen Komplexität umzusetzen, aber schwerer vor Erosion zu schützen.

Eingeführte Komplexitäten lassen sich in vielen Fällen beeinflussen. Die Experten berichteten von Kompromissen und Projektsituationen, die dazu führten, dass eine Komplexität gemindert und teilweise sogar umgangen wurde. Der Unterschied zwischen den beiden Einflussfaktoren - Kompromiss und Projektsituation - ist jedoch enorm. Kompromisse wirken in der Regel sowohl komplexitäts- als auch mehrwertmindernd. Dies ist auf die wechselseitige Beziehung von Mehrwerten und Komplexitäten zurückzuführen. Im Gegensatz dazu beeinflusst die zu jenem Zeitpunkt aktuelle Projektsituation die Komplexitäten insofern, als dass sie leichter oder auch schwerer bewältigt werden können.

Als Beispiel für einen Kompromiss kann das Deployment mehrerer Microservices auf einem *gemeinsamen* Application Server dienen. Die Komplexität im Betrieb sinkt, dafür wird jedoch der Mehrwert der Isolation ebenfalls gemindert. Die Microservices sind nicht mehr vollständig

voneinander isoliert, ein fehlerhafter Service kann den gesamten Application Server und somit alle Services zum Absturz bringen. Ein Beispiel für eine Projektsituation, die eine Komplexität beeinflusst, kann ein stark ausgeprägtes CD oder automatisierte Deployments sein. Die Inbetriebnahme vieler Microservices wird dann tendenziell kein großes Problem darstellen.

Der beidseitige Minderungseffekt eines Kompromisses ist nicht immer negativ, denn ein Mehrwert einer Architektur muss nicht zwangsläufig für jedes Projekt ein Mehrwert sein. Die Effizienz einer Lösung kann durchaus gesteigert werden, indem bewusst auf zwecklosen Mehrwert verzichtet wird. Der Fokus der Effizienzbestimmung liegt in der Bestimmung der tatsächlich nötigen Komplexitäten, sodass der Nutzen den Kosten überwiegt oder zumindest ein Gleichgewicht entsteht.

5.2 Spezifisches Wirkungsmodell

Bei den folgenden spezifischen Wirkungsmodellen handelt es sich um *mögliche* Ausprägungen des abstrakten Wirkungsmodells. Sie sind einer mittleren Abstraktionsebene zuzuordnen und stellen jeweils eine von möglicherweise vielen weiteren Ausprägungen dar. Diese Modelle wurden vor dem Hintergrund einer Microservice-Architektur erstellt. Sie beschreiben den Ursprung für den Bedarf des jeweiligen Mehrwerts im Detail und wie der Mehrwert spezifische Architekturziele unterstützt wird. Die Mehrwerte und Komplexitäten dieses Stils werden in Kurzform wiedergegeben. Details, wie sich diese zusammensetzen, können dem Kapitel 5.3 "Detaillierte Wirkungsmechanismen" entnommen werden. Auf Kompromisse und Projektsituationen wird in diesem Kapitel nicht eingegangen, da sich diese auf konkrete Herausforderungen einer Komplexität beziehen (siehe Kapitel 5.1.1.3 "Komplexität"). Sie können ebenfalls 5.3 entnommen werden.

5.2.1 Skalierbarkeit von agilen Prozessen - Organisation

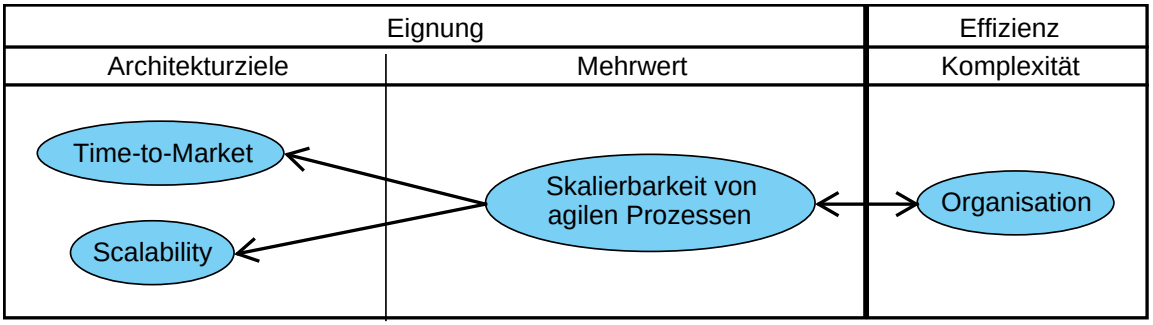

ABBILDUNG 5.2: Darstellung eines spezifischen Wirkungsmodells mit Fokus auf die Skalierbarkeit von agilen Prozessen mit organisationaler Komplexität

Aus organisatorischer Sicht stellt die Skalierbarkeit von agilen Prozessen einen der wesentlichen Vorteile einer Microservice-Architektur dar. In diesem Zusammenhang bedeutet Agilität, dass Anforderungen, die sich stetig ändern können, in einem iterativen und kommunikationsintensiven Prozess erfasst und realisiert werden. Dieses Vorgehen ist erprobt und hat sich in den letzten Jahren als ein erfolgreiches Vorgehensmodell für Softwareprojekte durchgesetzt. Von Skalierung wird dann gesprochen, wenn innerhalb eines festen Zeitraums mehr Anforderungen mittels mehr Personal umgesetzt werden. Dies bedeutet, dass die Größe oder die Anzahl der Teams steigt. Die Praxis zeigt jedoch, dass bei steigender Anzahl von Teammitgliedern die Produktivität ab einer bestimmten Teamgröße nicht weiter steigt. Einer der Gründe dieses Umstandes ist der steigende Kommunikationsaufwand zwischen den Teammitgliedern. Potenziell sind alle Beteiligten miteinander vernetzt. Mit der Formel n*(n-1)/2 lässt sich das Wachstum der Kommunikationswege abbilden.

Untersuchungen belegen, dass sich bei einem Team aus fünf Beteiligten ein Gleichgewicht zwischen Teammitgliedern und Kommunikationswegen ergibt.[2]

[2] vgl. *Die optimale Teamgröße*.

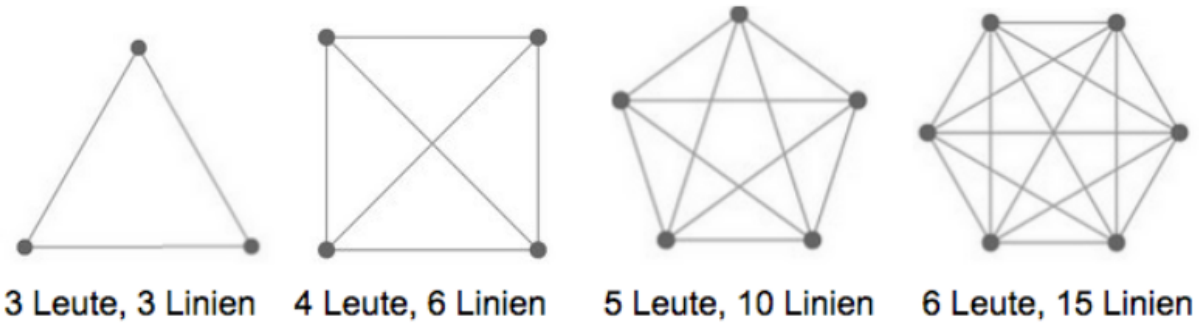

ABBILDUNG 5.3: Darstellung der Relation von Teamgröße zu Kommunikationswegen (Quelle: *Die optimale Teamgröße*)

Mehrwert

Microservice-Architekturen begegnen diesem Problem auf zwei Ebenen. Technisch wird das System in autarke Komponenten unterteilt. Diese sind möglichst auf keine anderen Komponenten angewiesen. Diese technische Basis ist die Voraussetzung für autonome Teams, deren Größe und Kommunikationswege ausgeglichen sind. Sie sind in der Lage eigenverantwortlich Entscheidungen zu treffen und selbstständig und zu arbeiten. Die Synergie dieser beiden Konzepte ermöglicht eine Skalierung der agilen Prozesse.

Architekturziele

Dieser Mehrwert führt zum einen zu einer Skalierbarkeit von agilen Prozessen. Diese Skalierbarkeit löst das Problem, dass Teams mit steigender Anzahl an Mitgliedern, aufgrund des gestiegenen Kommunikationsbedarfs, an Produktivität verlieren. Übersteigt die Arbeitslast die Kapazitäten des Unternehmens, wird ein neues Team eingeführt, die die Verantwortung für neue oder bestehende Services übernimmt. Zum anderen wird eine kürzere Time-to-Market erreicht. Das Gesamtprojekt wird durch viele kleine Projekte und kurzer Entwicklungszyklen agiler und kann schneller auf Veränderungen auf dem Markt reagieren. Projekte können in kurzer Zeit auf den Weg gebracht und neue Geschäftsbereiche mit geringem Risiko erprobt werden.

Komplexität

Der beschriebene Mehrwert ergibt sich aus einem Schnitt der Module nach fachlichen Aspekten. Er erfordert ein besonders gutes Verständnis für die Domäne und kann, wenn falsch durchgeführt, schwerwiegende Folgen haben. Außerdem müssen eine Reihe an organisatorischen Maßnahmen getroffen werden, um die Teams voneinander zu entkoppeln und selbstständiges sowie eigenverantwortliches Arbeiten zu ermöglichen.

5.2.2 Änderungsgeschwindigkeit - Automatisierung

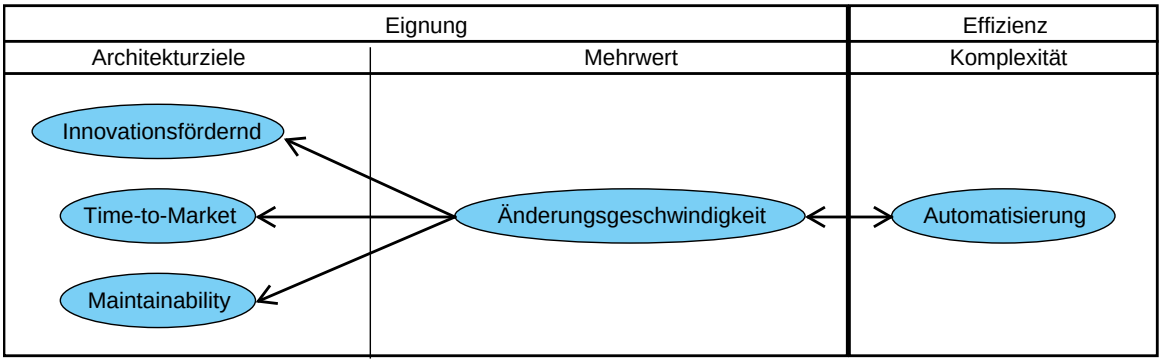

ABBILDUNG 5.4: Darstellung eines spezifischen Wirkungsmodells mit Fokus auf die Änderungsgeschwindigkeit durch die Automatisierung von Prozessen

Neben den in Abschnitt 5.2.2 "Änderungsgeschwindigkeit - Automatisierung" beschriebenen Herausforderungen, haben über Jahre gewachsene Projekte mit einer immer größer werdenden Codebasis, steigender Komplexität und einer Erosion der Architektur zu kämpfen. Dies sind meist Folgen von Anpassungen, die das Entwicklungsteam aufgrund von Zeitdruck einbaut, obwohl es weiß, dass die ideale Lösung bzw. die Architektur ganz anders aussehen müsste. Sollte die Architektur nicht regelmäßig geprüft und überarbeitet werden, steigt die Komplexität und der Kopplungsgrad der Bestandteile eines Systems. Diese sich aufbauende Schuld wird auch als technische Schuld bezeichnet. Sie hindern ein Team daran, Fehler schnell zu finden und Änderungen kostengünstig durchzuführen. Ein Projekt mit einer hohen technischen Schuld leidet typischerweise an Qualitätsverlust und immer längeren Entwicklungszyklen.[3]

[3]vgl. Lilienthal, *Wie Sie technische Schulden in Architekturen abbauen lassen*.

Mehrwert

Die erhöhte Änderungsgeschwindigkeit einer Microservice-Architektur zum einen durch autonome Teams erreicht. Die können Anforderungen schnell planen und umsetzten. Auf lange Sicht ändert sich die Änderungsgeschwindigkeit voraussichtlich nur geringfügig. Die starke Modularisierung sorgt dafür, es wird den Entwicklern wesentlich schwerer gemacht ungewollte Architekturentscheidungen zu treffen und so den Grad der Kopplung zwischen den Komponenten zu erhöhen. Ein weiterer Grund ist die kognitive Entlastung der Entwickler.

Architekturziele

Die hohe Änderungsgeschwindigkeit wird zum großen Teil mittels bereits lange bekannter Prinzipien erreicht. Information Hiding und die Modularisierung nach Fachlichkeit helfen die Wartbarkeit des Systems sicherzustellen und so Änderungen schnell realisieren zu können. Ein großer Vorteil ergibt sich vor allem in einer kurzen Time-to-Market. Änderungen, ob Fehlerbehebung oder neue Anforderungen, können schnell in Produktion gebracht werden. Dies erlaubt schnell und in hoher Qualität auf die Anforderungen des Marktes reagieren zu können. Entwicklungen schnell, mit geringem Risiko in Produktion zu bringen und diese gegebenenfalls zügig rückgängig machen zu können, schafft Raum für Innovationen. Unternehmen wie Netflix nutzen diese Innovationskraft, um den Markt nachhaltig zu verändern oder sogar einen neuen zu schaffen.

Komplexität

Diese Art der Änderungsgeschwindigkeit setzt eine Kultur der Automatisierung voraus. Mit steigender Anzahl an Services erhöht sicher unter anderem der Aufwand zum Bereitstellen und Testen der Services. Werden solche Prozesse nicht automatisiert, kann die Änderungsgeschwindigkeit drastisch abnehmen.

5.2.3 Flexibilität - Betrieb

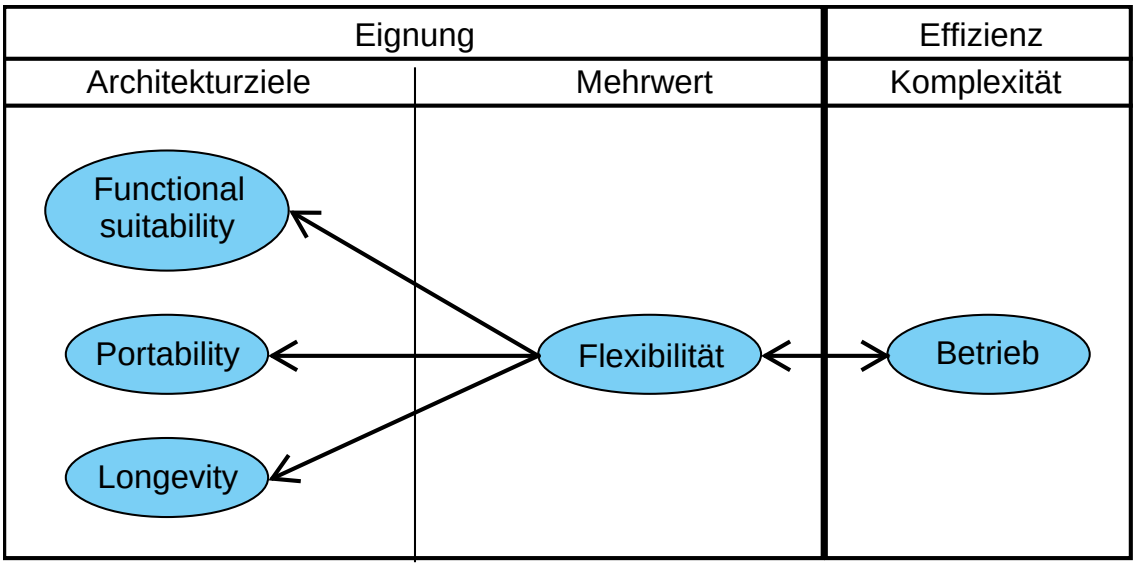

ABBILDUNG 5.5: Darstellung eines spezifischen Wirkungsmodells mit Fokus auf die Flexibilität und den Komplexitäten des Betriebs

Monolithische Architekturen, die als eine Einheit deployt werden, sind aus technischer Sicht relativ starre Konstruktionen. Die Entscheidung für einen Technologie Stack muss in jedem Fall vor der Entwicklung der ersten Komponente und in manchen Fällen schon während des Designs getroffen werden. Diese Entscheidung ist die Basis, auf der alle zukünftigen Entscheidungen basieren werden. Sie betrifft nicht nur das zu entwickelnde Produkt, sondern auch Werkzeuge, wie Continuous-Integration (CI), CD, Server, IDE und viele weitere werden von dieser Entscheidung beeinflusst. Wird im Laufe der Zeit aufgrund neuer Anforderungen festgestellt, dass ein Projekt besser mit Java als mit PHP umzusetzen wäre, muss ein kostenintensives Projekt zur Migrierung des gesamten Systems initiiert werden. Die andere Möglichkeit besteht darin, neue Anforderungen in einem neuen Produkt zu entwickeln und diese zu integrieren. Dies wird jedoch zu anderen Problemen führen, da das ursprüngliche Produkt nicht als verteiltes System konzipiert wurde. Änderungswünsche müssen jedoch nicht immer so gravierend sein. Bereits das Java Upgrade von einer Version auf eine andere oder die Aktualisierungen von verwendeten Frameworks kann ein Unternehmen vor kostenintensive Herausforderungen stellen, da diese systemweite Auswirkungen nach sich ziehen können.

Mehrwert

Microservice-Architekturen bieten einen hohen Grad an Flexibilität. Sie ermöglichen es, ein polyglottes System zu erstellen, in dem die Programmiersprache, verwendete Datenbanken und Frameworks gezielt für ein Problem ausgewählt werden. Dank der schrittweisen Migrationsfähigkeit dieses Architekturstils, kann der Umstieg von einem Deployment-Monolith hin zu einer Microservice-Architektur nach Bedarf durchgeführt werden. Das ganze System muss nicht in einem Schritt umgestellt werden.

Architekturziele

Die beschriebene Flexibilität einer Microservice-Architektur führt zu einer erhöhten Lebensdauer der entwickelten Systeme. Einzelne Services können im Laufe der Zeit effizient ausgetauscht oder auf eine neue Technologie portiert werden. Außerdem kann für ein spezifisches Problem, wie beispielsweise der Suche nach Produkten, eine angemessene Technologie auf Ebene der Mikroarchitektur ausgewählt werden, ohne durch vorangegangene Entscheidungen eingeschränkt zu werden. Wie die hohe Änderungsgeschwindigkeit auch, wirkt sich die Flexibilität positiv auf die Experimentierbereitschaft aus. Neue Technologien können erprobt werden und in Innovationen münden. Erfolglose Experimente können risikolos wieder verworfen werden. Nicht weniger wichtig ist, dass das Arbeiten mit neuen Technologien für viele Entwickler sehr reizvoll ist. Diesen Umstand kann der Arbeitgeber nutzen, um attraktiver für Bewerber zu werden.

Komplexität

Der hohe Grad an Flexibilität stellt ganz besondere Anforderungen an den Betrieb des Systems. Die Infrastruktur muss ggf. in der Lage sein, die vielen verschiedenen Technologien zu betreiben. Außerdem stellt das Monitoring und Logging eines verteilten, polyglotten Systems eine große Herausforderung dar. Es müssen Informationen ggf. unterschiedlicher Plattformen an einem zentralen Punkt zusammengefügt werden, um ein ganzheitliches Bild des Systems und dessen Zustands zu bekommen.

5.2.4 Elastische Skalierbarkeit der Anwendung - Verteiltes System

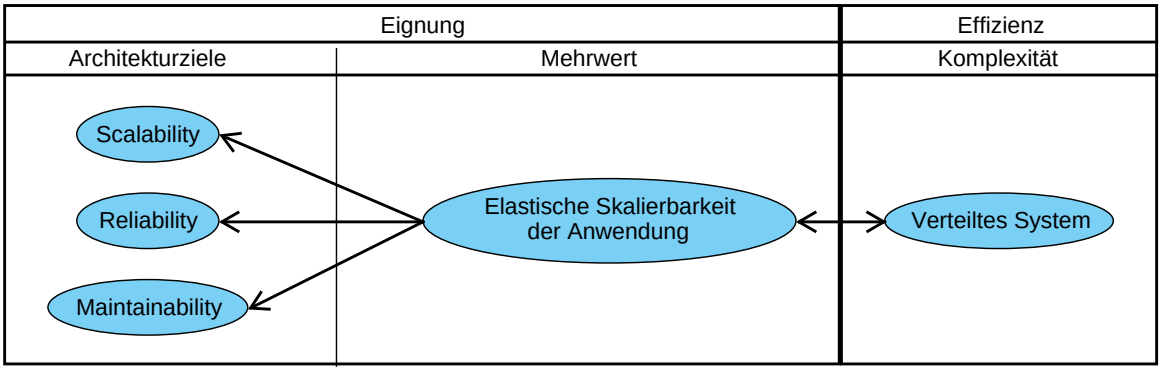

ABBILDUNG 5.6: Darstellung eines spezifischen Wirkungsmodells mit Fokus auf die elastische Skalierbarkeit der Anwendung und der Komplexität eines verteilten Systems

Steigen die Benutzerzahlen eines Softwaresystems an, kann sich das Unternehmen hinter dem System glücklich schätzen. Das Angebot wird gut angenommen und Erfolg stellt sich ein, wie auch immer dieser definiert wird. Dieser Zuwachs kann langsam aber stetig oder explosionsartig erfolgen. In manchen Fällen bleibt er bestehen, in anderen geht er wieder stark zurück. Onlineshops für Saisonprodukte wie Karnevalsverkleidungen, sind ein gutes Beispiel für einen rapiden Zuwachs und Abfall von Nutzeraktivitäten innerhalb von kurzer Zeit. Nicht selten stoßen Systeme dabei an ihre Belastungsgrenzen. Die Antwortzeiten nehmen stark zu, es entstehen scheinbar willkürliche Fehler oder das System bricht unter dem Druck gänzlich zusammen. Um diesem Druck gerecht zu werden, investiert das Unternehmen in neue Hardware. Der Server bekommt schnellere CPUs, mehr Arbeitsspeicher und neue Festplatten. Diese Art der Skalierung nennt man vertikaler Skalierung. Jede Art von Software ist auf diese Weise skalierbar, es braucht kein Quelltext angepasst zu werden. Der Nachteil ist zum einen, dass diese Skalierung eine natürliche Grenze hat. Der Rechner kann an einer bestimmten Grenze nicht weiter aufgerüstet werden. Zum anderen bindet sich das Unternehmen an die Hardware. Ein Karnevalsshop wird die Rechenleistung jedoch nur für zwei von zwölf Monaten benötigen, verschwendet also sehr viele Ressourcen.

Mehrwert

Im Gegensatz zu der zuvor beschriebenen vertikalen Skalierung, sind der horizontalen Skalierung aus Sicht der Hardware keine Grenzen gesetzt. Die Leistungssteigerung wird durch das Hinzufügen von Knoten bzw. Rechnern erreicht. Einige der Kernkonzepte von Microservice-Architekturen unterstützen diese Art der Skalierung besonders gut.

Architekturziele

Der Mehrwert der Skalierbarkeit kann ein angestrebtes Architekturziel sein. Neben dieser erreichen korrekt umgesetzte Microservice-Architekturen eine hohe Zuverlässigkeit des Systems. Diese setzt sich unter anderem aus Erreichbarkeit, Fehlertoleranz und Wiederherstellbarkeit einer Software zusammen. All dies sind Ziele, die eine Microservice-Architektur positiv beeinflusst. Die Skalierung aufgrund von Last oder das Deployment in unterschiedlichen räumlichen Regionen, um die Software näher an den Benutzer zu bringen, sind Maßnahmen für eine erhöhte Erreichbarkeit. Ein gutes Monitoring und das automatische Bereitstellen von Services beschleunigt die Wiederherstellung einer Anwendung nach einem Zusammenbruch des gesamten oder Teilen des Systems. Der nächst logische Schritt ist, dass sich das System selbst wiederherstellt oder in den angestrebten Zustand versetzt. Es wird von "Self-Healing Systems"[4] gesprochen, bei denen unter anderem die Architektur des Systems eine wichtige Rolle spielt. Eines der Kernkonzepte von Microservice-Architekturen ist das "Design for failure"[5]. Das System wird aufgrund der Schwächen einer verteilten Anwendung (siehe Abschnitt 2.7.3 "Unzuverlässige Kommunikation") von Anfang an für eine hohe Fehlertoleranz konzipiert.

Die lose Kopplung und die starke Modularisierung bilden die technische Basis für sehr viele Aspekte von Microservice-Architekturen. Eines der wohl wichtigsten Ziele ist, die Wartbarkeit von Software zu verbessern. Die Erosion der Architektur wird eingedämmt und die Bildung eines "Big Ball of Mud", den Brian Foote und Joseph Yoder als "a haphazardly structured,

[4] vgl. Farcic, „Self-Healing Systems: Wie man Microservices-basierten Systemen die Selbstheilung beibringt“.
[5] vgl. Fowler, *Microservices*.

sprawling, sloppy, duct-tape-and-baling-wire, spaghetti-code jungle"[6] beschreiben, wird in sofern unterbunden, als dass jede neue Abhängigkeit von Modulen einer Architekturdiskussion benötigt.

Komplexität

Die Komplexität ergibt sich in Form eines verteilten Systems mit einer ganzen Reihe an verschiedenartigen Aspekten. Zu nennen sind vor allem die unzuverlässige Kommunikation über das Netzwerk, die Datenarchitektur, Transaktionen und die Realisierung von Integrationstests.

5.3 Detaillierte Wirkungsmechanismen

In diesem Kapitel werden die bisher vorgestellten spezifischen Wirkungsmodelle auf der niedrigsten Abstraktionsebene betrachtete. Es folgt eine detaillierte Beschreibung, aus welchen Elementen sich die Mehrwerte und Komplexitäten zusammensetzen. Außerdem werden Vorschläge für mögliche Kompromisse und deren Auswirkung gemacht, sowie Projektsituationen beschrieben, die eine Komplexität beeinflussen könnten.

5.3.1 Skalierbarkeit von agilen Prozessen - Organisation

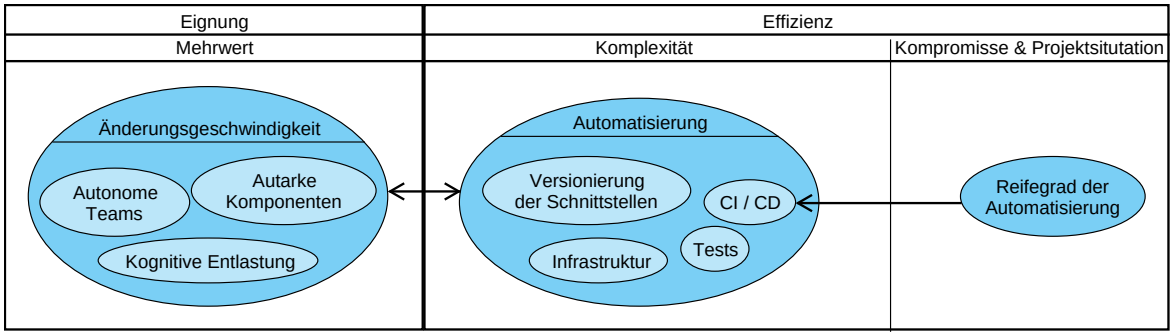

ABBILDUNG 5.7: Darstellung eines detaillierten Wirkungsmechanismus mit Fokus auf die Skalierbarkeit von agilen Prozessen, organisationaler Komplexität und möglichen Kompromissen

[6] Foote und Yoder, *Big Ball of Mud*, S. 2.

Mehrwert

Microservice-Architekturen begegnen dem Problem des steigenden Bedarfs und Aufwands von Kommunikation in wachsenden Teams auf zwei Ebenen. Technisch wird das System in autarke Komponenten unterteilt. Diese sind möglichst auf keine anderen Komponenten angewiesen. In der Realität lässt sich ein komplexes System jedoch nicht ausschließlich aus solchen Komponenten erstellen. Ein Teil des Systems wird immer einen anderen benötigen, um ein komplexes Problem zu löschen. In einer Architektur, die dem Microservice-Ansatz folgt, wird dieser Informationsaustausch jedoch mittels expliziten Schnittstellen realisiert. Übernimmt also jeweils ein Team die Verantwortung für eine Komponente, beschränkt sich die Kommunikation der Teams lediglich auf die Schnittstelle. Neben der Kommunikation müssen die Komponenten auch auf der Ebene des Deployments entkoppelt werden. Ein Service sollte zu jedem Zeitpunkt und ohne die Abstimmung mit anderen Teams deploybar sein.

Diese technische Basis ist die Voraussetzung für autonome Teams, deren Größe und Kommunikationswege ausgeglichen sind. Sie sind in der Lage, eigenverantwortlich Entscheidungen zu treffen und selbstständig und zu arbeiten. Unter diesen Voraussetzungen ist es möglich, ein Projekt in viele kleine zu zerlegen. Neben den bereits beschriebenen Vorteilen solcher Projekte (siehe Kapitel 2.7.2 "Kleine Projekte") können diese effizient skalieren. Die Teams sind sowohl auf technischer als auch auf organisatorischer Ebene weitestgehend unbeeinflusst und können so kleine (Teil) Projekte selbstständig umsetzen. Entstehen mehr Projekte, können mehr Teams eingeführt werden, ohne einen Effizienzverlust zu erzeugen.

Komplexität

Die zuvor beschriebenen Vorteile ergeben sich jedoch nicht von alleine. Um Komponenten aus technischer Sicht erfolgreich voneinander zu entkoppeln, ist der Schnitt bzw. die Grenzen der Services besonders wichtig. Sind die Services falsch geschnitten, kann sich der vermeintliche Vorteil einer Microservice-Architektur schnell zu einem Nachteil entwickeln. Eines der Resultate kann steigender teamübergreifender Kommunikationsbedarf sein. Wurden die Services nicht nach ihrer Fachlichkeit geschnitten, wirken sich neue Anforderungen oftmals auf mehr als einen Service aus. Dies führt dazu, dass sich die Teams untereinander abstimmen müssen, wie und

wann ein Feature implementiert wird und wann es bereitgestellt wird. Die Teams sind nicht mehr autonom, sondern miteinander verzahnt.

Eine weitere Herausforderung stellen die Fähigkeiten und Befugnisse der Teams dar. Ein Team ist für den gesamten Lebenszyklus sowie für alle technischen Aspekte eines Service verantwortlich. Demnach müssen die Teammitglieder hoch qualifiziert und nach Möglichkeit Erfahrungen im gesamten Technologie Stack aufweisen, um ein neues Feature über alle Phasen (Analyse, Design, Implementierung, Betrieb und Wartung) und Schichten (Frontend, Backend und Datenbank) hinweg entwickeln und betreiben zu können. Außerdem müssen die Teams in die Lage versetzt werden, selbstständig einen Service in Betrieb zu nehmen. Es müssen entsprechende Befugnisse zugestanden und Tools geschaffen werden. Viele dieser Anforderungen spiegeln sich in den Werten der DevOps-Kultur wieder.

Kompromisse und Projektsituation

Nicht alle Komplexitäten lassen sich bereits zu Beginn der Entwicklung lösen. Experten eines bestimmten technologischen Bereiches können nicht von heute auf morgen zu Experten in anderen werden. Es ist jedoch sehr sinnvoll, das bestehende Wissen in einem Unternehmen zu nutzen und zu verbreiten. Bestehende, technisch orientierte Abteilungen eines Unternehmens, wie Frontend, Backend und Operations sollten in neuen, fachlich orientierten Teams organisiert werden. Dieser Umbruch darf jedoch nicht nur auf dem Papier erfolgen, auch eine räumliche Neuordnung ist von Nöten, um den gewünschten Kommunikationsfluss zu erzeugen (siehe Abschnitt 2.5.1 "Fachliche Teams"). Die Bildung von Cross-Functional Teams kann ein erster Schritt in die Richtung von autonomen Teams sein.

Ein anderer Ansatz kann sein, Development und Operations als Abteilungen getrennt voneinander zu halten, diese jedoch eng miteinander zu verbinden, um die Kommunikation und Zusammenarbeit zu steigern. Die Teams müssen den Willen verinnerlichen, der nötig ist, um die Probleme des anderen am eigenen Leibe zu erleben. Auf diese Weise kann ein tiefes Verständnis für dessen Belange geschaffen werden.[7]

[7] vgl. Brown, *What's the Best Team Structure for DevOps Success?*

5.3.2 Änderungsgeschwindigkeit - Automatisierung

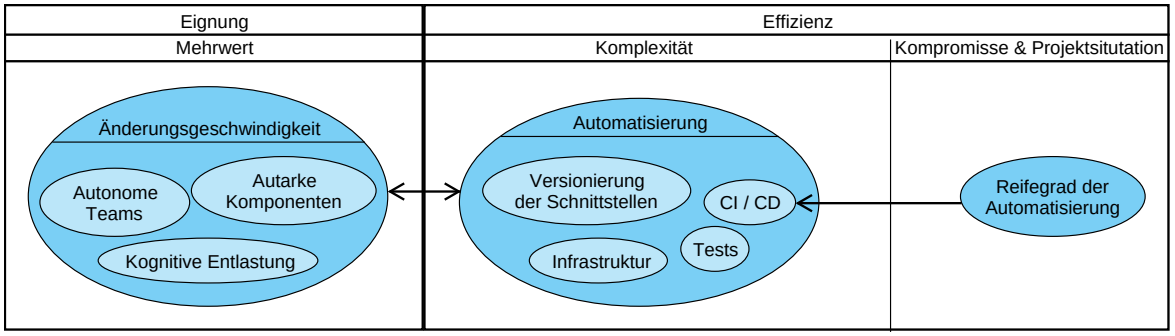

ABBILDUNG 5.8: Darstellung eines detaillierten Wirkungsmechanismus mit Fokus auf die Änderungsgeschwindigkeit durch die Automatisierung von Prozessen und möglichen Kompromissen

Mehrwert

Auch dem Problem der sinken Änderungsgeschwindigkeit aufgrund von technischen Schulden wirken Microservice-Architekturen mit autarken Komponenten entgegen. Diese Komponenten setzen eine starke Modularisierung und hochgradige Geschlossenheit voraus. Es wird den Entwicklern wesentlich schwerer gemacht, ungewollte Architekturentscheidungen zu treffen und so den Grad der Kopplung zwischen den Komponenten zu erhöhen. Dies verhindert jedoch nicht, dass unerwünschte Änderungen innerhalb eines Service eingebaut werden. Sie verhindern lediglich, dass die Makroarchitektur leichtfertig und ohne Architekturdiskussionen geändert wird. Steigen die technischen Schulden eines Service bis zu einem Grad, an dem sie nicht mehr vertretbar sind oder eine Ausbesserung nicht lohnenswert ist, kann ein Service verworfen und neu implementiert werden, ohne dass andere Services davon betroffen sind. Es wird verhindert, dass die Architektur so weit erodiert, dass die Änderungsgeschwindigkeit nicht mehr vertretbar ist und das Projekt möglicherweise komplett neu entwickelt werden muss

Ein weiterer Grund für die hohe Änderungsgeschwindigkeit einer Microservice-Architektur ist die kognitive Entlastung der Entwickler. Das Hinzufügen neuer fachlicher Aspekte oder das Verändern bestehender, beschränken sich idealerweise auf einen oder eine geringe Anzahl an Services, die mittels expliziter Schnittstellen miteinander kommunizieren. Bei der Entwicklung kann sich ein Entwickler oder das Team auf eine Fachlichkeit beschränken. Sie müssen nicht

das gesamte System verstehen, um die Auswirkung einer Änderung einzuschätzen. Diese können bei einem System mit starker Kopplung sehr vielseitig und folgenschwer sein.

Ein Team, das autonom agiert, kann Anforderungen schnell planen und umsetzten. Es fokussiert auf eine Fachlichkeit, trifft Entscheidungen, die die Mikroarchitektur etreffen selbstständig und hat seinen eigenen Zeitplan, in dem die Aufgaben priorisiert werden können. Sind Absprachen mit anderen Teams erforderlich, betreffen diese auch nur die Schnittstellen. Es kann nach dem Contract-First-Prinzip vorgegangen werden, bei dem zuerst die Schnittstelle definiert wird und danach die Realisierung erfolgt.[8] Gepaart mit Deployments von Services, die nicht koordiniert werden müssen, wird eine hohe Änderungsgeschwindigkeit des Systems erreicht.

Komplexität

Diese Art der Änderungsgeschwindigkeit setzt jedoch einiges voraus. Es müssen, wie bereits in Abschnitt 5.3.1 "Skalierbarkeit von agilen Prozessen - Organisation" beschrieben, autarke Komponenten und autonome Teams geschaffen werden. Neben diesen Komplexitäten muss eine Kultur der Automatisierung entstehen. Bei steigender Anzahl von Services können diese weder manuell getestet noch bereitgestellt werden. Es gilt unter anderem die Schnittstellen der angrenzenden Services zu testen. Denn diese können sich trotz, der fachlichen Modularisierung, ändern. Hier können Prinzipien, wie das Consumer-driven Contract Testing helfen, bei dem der Consumer eines Service(Provider) festlegt, was für Antworten er vom Producer erwartet. Ändert der Producer seine Rückgaben, kann festgestellt werden, welche Consumer von dieser Änderung betroffen sind. Aus solchen Änderungen ergibt sich das Problem, dass ein Provider unterschiedliche Versionen einer Schnittstelle bereitstellen muss, damit die Consumer zu einem geeigneten Zeitpunkt auf die neue Version der Schnittstelle umsteigen können. Den bereits ohnehin komplexen und schwer nachvollziehbaren Abhängigkeiten der Services wird eine weitere Dimension hinzugefügt.

Das manuelle Bereitstellen von einem oder einiger weniger Services ist durchaus möglich. Müssen jedoch zehn, zwanzig oder zweihundert Services betrieben werden, ist diese Aufgabe nicht mehr manuell zu bewältigen. Einige Experten führen an, dass die Einführung von CD durch

[8] vgl. Abedrabbo, *The 7 Deadly Sins of Microservices*.

Microservices erleichtert wird. Sie ist jedoch erst einmal eine Komplexität, die andere architektonische Ansätze nicht zwangsläufig umsetzen müssen. Microservice-Architekturen setzen ab einer bestimmten Größe CD, oder zumindest eine automatisierte Bereitstellung der Services, voraus. Neben dem Deployment in die Produktivumgebung muss sichergestellt werden, dass den Teams die beantragte Ressourcen zügig bereitgestellt werden. Die Entwicklung neuer Services darf nicht durch die Provisionierung von Servern beschränkt werden. Es ist nutzlos, wenn ein Team einen Service innerhalb einer Woche entwickelt, die Ressourcen aber erst nach drei Wochen bereitstehen. Dies betrifft die gesamte Infrastruktur, Source-Control-Management (SCM), Tools zum Managen von Projekten und Continuous-Delivery-Pipeline sind einige Beispiele.

Kompromisse und Projektsituation

Nach Hornauer erfolgt Automatisierung in fünf Stufen. Diese fünf Stufen könnten für den Integrations- und Deployment-Prozess folgendermaßen aussehen.[9]

- **Stufe 1: Vollständig manuell**
  Ein Entwickler oder Tester baut und prüft die Software auf seiner Entwicklungsmaschine. Das geprüfte Paket wird per Email an einen Mitarbeiter versendet, der für die Server verantwortlich ist. Dieser spielt das Paket im Optimalfall nach einer Anleitung oder im Worst-Case nach bestem Wissen und Gewissen ein.

- **Stufe 2: Assistenzsysteme**
  Ein Entwickler publiziert seine Änderungen in einem SCM System. Ein Assistenzsystem wird angewiesen die Software zu kompilieren oder sie auszuführen und führt eine geringe Anzahl an Tests durch. Der Betrieb bekommt dieses Paket manuell per Email oder FTP bereitgestellt, um es zwecks End-to-End Tests auf einem Testserver zu deployen. Nach mehrfacher Wiederholung und dem Beheben von Fehlern wird ein Paket manuell in Produktion gebracht.

[9] vgl. Hornauer, „Industrielle Automatisierungstechnik“, S. 34.

- **Stufe 3: Teilautomatisierung**
  Nach der Publizierung in einem SCM System reagiert ein Assistenzsystem, es kompiliert, integriert die Systemteile, führt Unit- und Integrationstests aus. Das geprüfte Paket wird manuell oder teilautomatisiert in Test- und Produktivumgebung deployt.

- **Stufe 4: Hochautomatisierung**
  Das System wird zum größten Teil automatisiert getestet. Nach Sichtung und einer Freigabe durch einen oder mehrere Verantwortliche wird es automatisiert in die Produktivumgebung deployt.

- **Stufe 5: Vollautomatisierung**
  Integration, Tests und Deployment erfolgen vollautomatisiert. Ein manueller Eingriff ist nur bei Komplikationen von Nöten.

Im Bereich der Softwareentwicklung könnte die sechste Stufe das automatische Zurückrollen auf eine vorherige Version oder das automatisierte Beheben von Problemen umfassen.

Ausgehend von der Stufe der Automatisierung auf der sich ein Unternehmen oder Projekt befinden, ist die Einführung von Microservice-Architekturen weniger komplex. Aus den Experteninterviews geht hervor, dass die Automatisierung aufgrund und durch Microservices vorangetrieben wird. Oft handelt es sich um einen Prozess, der zeitgleich mit den Änderungen der Architektur einhergeht. Durch die steigende Automatisierung wird der Mehraufwand, der sich dank der vielen Services ergibt, immer weiter gesenkt.

5.3.3 Flexibilität - Betrieb

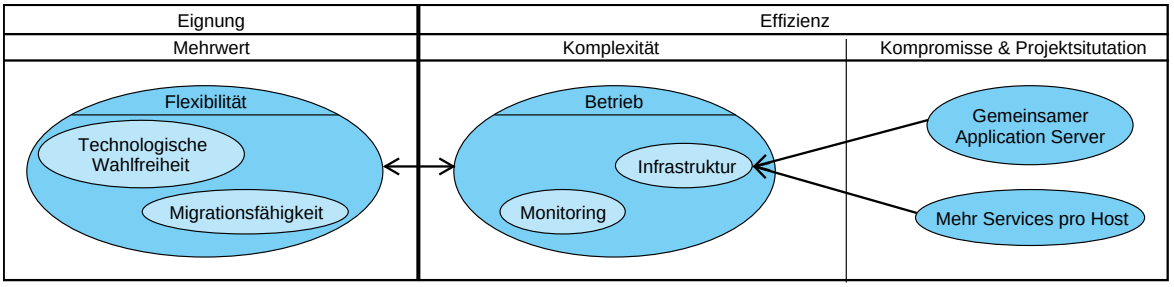

ABBILDUNG 5.9: Darstellung eines detaillierten Wirkungsmechanismus mit Fokus auf die Flexibilität, den Komplexitäten des Betriebs und möglichen Kompromissen

Mehrwert

Microservice-Architekturen bieten einen hohen Grad an Flexibilität. Sie ermöglichen es ein polyglottes System zu erstellen, in dem die Programmiersprache gezielt für ein Problem ausgewählt wird, aber auch verwendete Datenbanken und Frameworks eines Service können sich aufgrund der starken Modularisierung voneinander unterscheiden. Die Größe eines Service trägt ebenfalls zu dieser Flexibilität bei. Stellt ein Unternehmen beispielsweise das System zur Produktverwaltung um, kann der Service, der ein System mit der Produktverwaltung verbindet, relativ schnell und unter geringem Risiko verworfen und neu entwickelt werden. Das System wird aus technischer Sicht sehr viel agiler und kann so schneller und gezielter auf Ereignisse von außen und innen reagieren.

Ein weiterer wichtiger Mehrwert, der zur Flexibilität beiträgt, ist die schrittweise Migrationsfähigkeit dieses Paradigmas. Der Umstieg von einem Deployment-Monolith, hin zu einer Microservice-Architektur, muss nicht nach dem Big-Bag-Prinzip erfolgen. Die Umstellung des gesamten Systems, nach mehreren Wochen oder Monaten an Arbeit, birgt ein sehr großes Risiko. Die Anforderungen können sich in der Zeit ändern, das neue System verhält sich unerwartet, oder anders als das alte, es kommt zu Komplikationen bei der Inbetriebnahme oder die Portierung des Datenbestandes weist Fehler auf. Nach dem Microservice-Gedanken werden Teile des alten Systems herausgelöst und in einen Service überführt. Ein Proxy-Server könnte dann Anfragen, jeweils an das alte oder neue System weiterleiten, um sie dort zu bearbeiten. In der

Praxis birgt dieses Vorgehen natürlich auch einige Herausforderungen, wie die Kommunikation des alten und neuen Systems, der Handhabung des Datenbestandes oder Sicherheitsaspekte. Diese sollten keinesfalls unbeachtet bleiben.

Komplexität

Der hohe Grad an Flexibilität stellt ganz besondere Anforderungen an den Betrieb des Systems. Besteht eine Systemlandschaft aus vielen verschiedenen Technologien, muss eine Infrastruktur geschaffen werden, die es erlaubt, diese möglichst einheitlich zu betreiben. Das Betreiben eines komplexen Systems, dessen Services unterschiedlich bereitgestellt und betrieben werden müssen, ist über längere Zeit nicht wünschenswert. Mit jedem neuen Service nehmen Komplexität und die Anzahl der Fehlerquellen zu. Letztendliche nimmt die Änderungsgeschwindigkeit ab. Virtualisierungstechnologien wie Docker, bieten eine Abstraktionsschicht von Software zu Hardware bzw. Betriebssystem. Die Software wird in einem Container verpackt, die aus der Sicht des Betriebs immer gleich behandelt wird, egal welche Technologie im Container selbst genutzt wird.

Eine weitere Herausforderung stellt die Überwachung eines Systems dar, das sich aus vielen kleinen Teilen zusammensetzt. Es gibt nicht mehr den einen Punkt, an dem Speicherverbrauch, Festplattenzugriffe oder Last überwacht werden, um bei der Überschreitung eines Grenzwertes eingreifen zu können. Per Definition stürzen Deployment-Monolithen als Ganzes ab. Ist das System nicht mehr erreichbar, wird dies oftmals sehr schnell und deutlich erkennbar. Im Gegensatz dazu haben Microservices den Vorteil, dass das System die Arbeit nur teilweise verweigert und Funktionen des Systems nur eingeschränkt nutzbar sind. Festzustellen, wann ein verteiltes System in diesen Zustand verfällt, ist eine durchaus anspruchsvolle Aufgabe. Das Monitoring einer Microservice-Architektur sollte an einer zentralen Stelle die Informationen jedes Service erfassen. Informationen wie Netzwerkzugriffe, Last und Puls (Health Checks) aber auch service- bzw. anwendungsspezifische Informationen helfen dabei Systemfehler, Ressourcenengpässe und Optimierungsmöglichkeiten aufzudecken. Die heterogene Systemlandschaft stellt dabei eine besondere Herausforderung dar. Eine Monitoring-Lösung ermöglicht eine solche Vielfalt nicht zwangsläufig, daher sollte man diesem Problem in der Makroarchitektur besondere Beachtung schenken.

Ähnlich wie beim Monitoring verhält es sich mit dem Logging von Fehlern. Wird ein Fehler auf der Oberfläche erkennbar, kann die Fehlerquelle in jedem der aufgerufenen Services liegen. Es müssten also alle Log-Dateien gesichtet werden. Der Betrieb von mehr als einer Instanz pro Service verkompliziert dieses Problem noch weiter. Es bedarf einer zentralen Lösung, in der alle Log-Informationen zusammenlaufen. Aufrufe, die über mehrere Service hinweg stattfinden und kaskadierende Fehler erzeugen, sollten erkennbar und nachvollziehbar sein. Das zentrale Logging ist demnach mindestens genauso wichtig und herausfordernd wie das Monitoring einer Microservice-Architektur.

Kompromisse und Projektsituation

Um die verschiedenen Möglichkeiten zu beleuchten, wie eine Microservice-Architektur betrieben werden kann, müssen zuerst einige Begriffe geklärt werden. Im Zeitalter der Virtualisierung spricht Newman von einem Host, wenn ein einzelnes, isoliertes System gemeint ist, nämlich das Betriebssystem, auf dem Services installiert und betrieben werden können. Wird demnach ein Betriebssystem unmittelbar auf einem physikalischen Server betrieben, handelt es sich um einen Host. Werden Virtualisierungs- oder Container-Technologien eingesetzt, in denen ein abgeschottetes Betriebssystem ausgeführt wird, können mehrere Hosts auf einer physikalischen Maschine betrieben werden. Hinzu kommt das Konzept des Anwendungscontainers oder Anwendungsservers. Dieser stellt die Funktionalitäten eines Service bereits Es werden zwei Varianten unterschieden:

- **Schwergewichtig:** Es können mehrere Instanzen von Services betrieben werden. Sie optimieren den Ressourcenverbrauch und stellen eine ganze Reihe an Werkzeugen zur Verfügung. Bekannte Beispiele sind Internet Information Services (IIS) und Apache Tomcat.
- **Leichtgewichtig:** Ein Anwendungsserver ohne viele Extras. Im Kontext von Microservices werden diese vom Service selbst bereitgestellt (embedded), es wird also maximal eine Anwendung pro Server bereitgestellt. Beliebte Beispiele sind Spring Boot und Jetty.

Variante A - Anwendungsserver (siehe Abbildung 5.10 "Darstellung der drei Varianten, wie Services betrieben werden können (Quelle: Newman, *Microservices (mitp Professional): Konzeption und Design*, S. 203, 205, 208)") sieht vor, einen möglicherweise bereits bestehenden

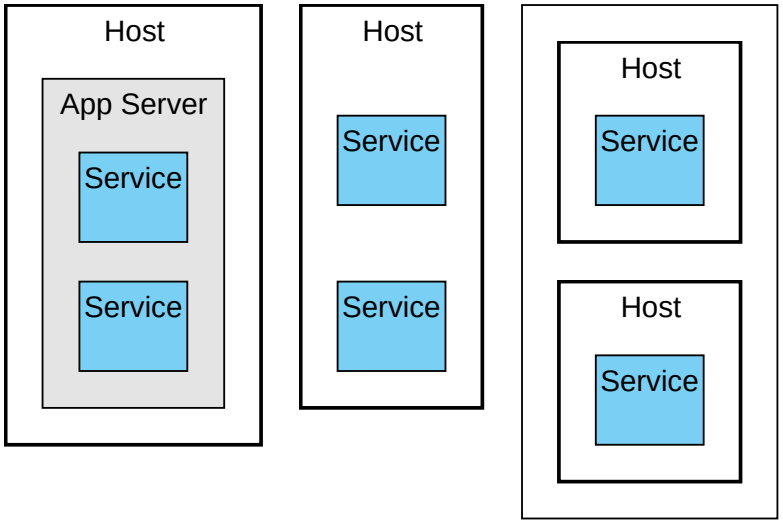

ABBILDUNG 5.10: Darstellung der drei Varianten, wie Services betrieben werden können (Quelle: Newman, *Microservices (mitp Professional): Konzeption und Design*, S. 203, 205, 208)

Anwendungsserver, der auf einem Host betrieben wird, für das Deployment aller Microservices zu nutzen. Dies kann zum einen die Komplexität zum Betreiben der Services verringern. Newman führt hier vor allem die effiziente Verwendung von Ressourcen an. Die Nachteile seien jedoch schwerwiegend und sollten dringend abgewogen werden. Zu nennen sind vor allem, dass die Technologiefreiheit aufgrund der Wahl des Anwendungsservers eingeschränkt wird, Speicherauslastungen sich auf die Funktionsfähigkeit aller Services auswirken und Stateful Anwendungen begünstigt werden, die wiederum Probleme bei der Skalierung verursachen.

Variante B - mehr-Service-ein-Host (siehe Abbildung 5.10 "Darstellung der drei Varianten, wie Services betrieben werden können (Quelle: Newman, *Microservices (mitp Professional): Konzeption und Design*, S. 203, 205, 208)") sieht den Betrieb mehrerer Services, beispielsweise mittels leichtgewichtiger Anwendungsserver, auf einem Host vor. Im Vergleich zu Variante C singt vor allem der Bereitstellungs- und Verwaltungsaufwand, der für jeden Host anfällt, der betrieben werden soll. Die Nachteile sind ähnlich denen der Variante A. Das Monitoring wird schwerer, da sich die Services Ressourcen teilen und sich diese gegenseitig beeinflussen. Das Gleiche gilt für das Deployment der Services. Außerdem behindert die Konfiguration des Servers die Eigenständigkeit der Teams. Wer ist für einen lauffähigen und korrekt konfigurierten Server verantwortlich, wenn alle Teams diesen nutzen?

Variante C - ein-Service-pro-Host (siehe Abbildung 5.10 "Darstellung der drei Varianten, wie Services betrieben werden können (Quelle: Newman, *Microservices (mitp Professional): Konzeption und Design*, S. 203, 205, 208)") bietet den höchsten Grad an Isolation der Services. Dies hat Vorteile beim Monitoring, für die Arbeitsweise der Teams, im Bereich der Wiederherstellbarkeit und senkt so die Komplexität des Gesamtsystems. Nicht jedes Unternehmen ist sofort in der Lage dieses Konzept zu übernehmen, dennoch empfiehlt Newman, sich diesem allmählich zu nähern. Als Nachteile ergeben sich vor allem, dass mehr Server verwaltet werden und der Betrieb zusätzlicher Hosts mit Kosten verbunden sein kann.

Eine weitere Variante, die nicht ungenannt bleiben soll, ist das Nutzen einer PaaS Lösung. Sie bewegt sich im Vergleich zur Variante C auf einer höheren Abstrahierungsebene. Solche Plattformen nehmen Artefakte entgegen und stellen diese automatisch in einem isolierten Host bereit. Wird ein solcher Service genutzt, kann der Verwaltungsaufwand erheblich gesenkt werden, allerdings geht damit ein Kontrollverlust einher. Funktioniert ein Service nicht so wie erwartet, gibt es oft kaum eine Möglichkeit einen Blick hinter die Kulissen zu werfen, um das Problem zu beheben.[10]

5.3.4 Elastische Skalierbarkeit der Anwendung - Verteiltes System

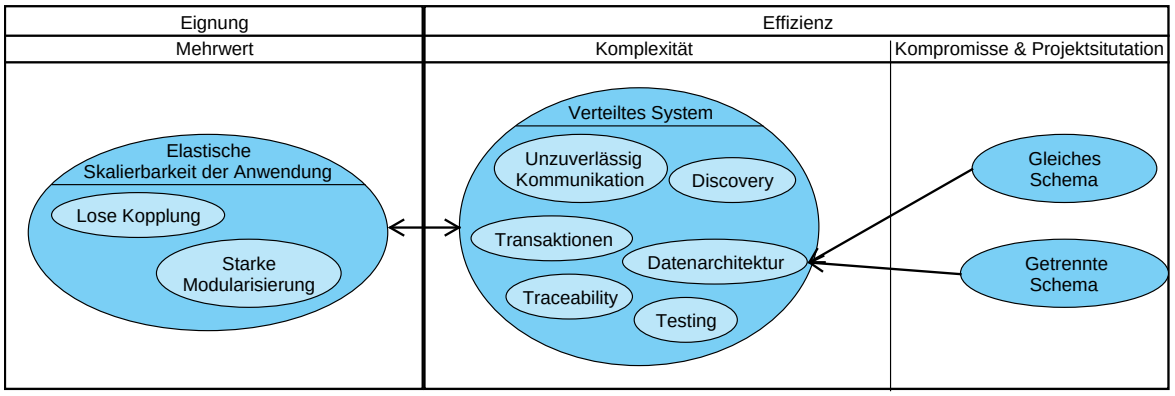

ABBILDUNG 5.11: Darstellung eines detaillierten Wirkungsmechanismus mit Fokus auf die elastische Skalierbarkeit der Anwendung, der Komplexität eines verteilten Systems und möglichen Kompromissen

[10]vgl. Newman, *Microservices (mitp Professional): Konzeption und Design*, S. 210 - 219.

Mehrwert

Im Gegensatz zu der in Kapitel 5.2.4 "Elastische Skalierbarkeit der Anwendung - Verteiltes System" beschriebenen vertikalen Skalierung, sind der horizontalen Skalierung aus Sicht der Hardware keine Grenzen gesetzt. Die Leistungssteigerung wird durch das Hinzufügen von Knoten bzw. Rechnern erreicht. Wird die Leistung nicht mehr benötigt, können Knoten entfernt werden. Die Effizienz dieser Skalierung hängt jedoch stark von der Implementierung der Software ab. Microservice-Architekturen fokussieren auf diese Art der Skalierung und gehen sogar noch einen Schritt weiter. Sie erlauben eine horizontale Skalierung von fachlichen Bereichen. Ist die Artikelsuche unter besonders hoher Last, wird dieser Service auf mehreren Knoten gleichzeitig betrieben und die Last unter diesen verteilt. Der Bestell-Service wird wesentlich weniger genutzt und daher auch auf sehr wenigen Knoten betrieben. Kleine Deploymenteinheiten, starke Modularisierung und die lose Kopplung der Services erleichtern, die Services individuell zu skalieren. Microservices werden oft im Zusammenhang mit dem Begriff Cloud-Nativ genannt. Dieser beschreibt Softwaresysteme, die für den Betrieb auf einer Cloud-Plattform, wie Amazon Web Services (AWS), ausgelegt wurden. Eine solche Plattform erlaubt Ressourcen effizient hinzuzufügen und diese wieder abzubauen.

Komplexität

Die Komplexität ergibt sich in Form eines verteilten Systems. In einer sich stetig wandelnden Umgebung, in der eine variable Anzahl an Instanzen eines Service an unterschiedlichen Knoten eines Netzwerks gestartet und gestoppt werden, muss ein Mechanismus geschaffen werden, mit Hilfe dessen die einzelnen Services miteinander kommunizieren können. REST und Messaging sind weitverbreitete Lösungen mit unterschiedlichen Vor- und Nachteilen. Auf Basis der Anforderungen kann die eine Lösung besser geeignet sein als die andere, oder es entsteht eine Mischung aus beiden. Die Implementierung einer solchen Lösung stellt im Bezug zu einem Deployment-Monolithen einen Mehraufwand dar, da dessen Komponenten direkt mittels des Speichers kommunizieren. Zusätzlich ergibt sich das Problem, dass sich die Services oder die Message Queue im Netzwerk gefunden werden muss. Das Hinterlegen fester IPs zu einem Service ist keine praktikable Lösung, da sich diese laufend ändern können und ein Load Balancing erschwert oder sogar verhindert wird.

Die Kommunikation über das Netzwerk hat den Nachteil, dass sie unzuverlässig ist. Früher oder später kommt es zu Ausfällen im Netzwerk oder Nachrichten kommen extrem verzögert am Empfänger an. Diese Fehler können kaskadieren, schwer zu reproduzieren und können den Neustart eines Service verhindern. Ein Stack, der fehlgeschlagene Anfragen zur späteren Bearbeitung speichert, kann einen gerade neu instanziierten Service durch eine Flut an Anfragen erneut zum Absturz bringen. Das Designparadigma Resilience geht davon aus, dass Kommunikationspartner nicht zur Verfügung stehen, und betrachtet diese Eventualität bereits während des Designprozesses.

Eine weitere Herausforderung stellt die Datenarchitektur dar. Welche Datenbank(en) wird genutzt? Welcher Service hält welche Daten? Wie wird mit Datenreplikation umgegangen? Wie wird Inkonsistenzen vorgebeugt oder wie werden diese behoben? Sind Inkonsistenzen überhaupt ein Problem? Müssen Ereignisse als Transaktion realisiert sein? Wie wirkt sich das CAP-Theorem auf die Architekturziele und Anforderungen aus? Die Antworten auf diese Fragen können in jedem Projekt anders ausfallen und stellen Entwickler, Architekten sowie das Management vor neue Herausforderungen. Sie sind jedoch essenziell, um eine geeignete Architektur zu entwerfen.

Auch das Testen einer Microservice-Architektur gestaltet sich bei genauerer Betrachtung komplexer als das Testen eines Deployment-Monolithen. Ganz besonders trifft dies auf die Integrationstests zu, bei denen das Zusammenspiel der verschiedenen Komponenten oder Services getestet wird. Durch die vielen Services, die sich in unterschiedlichen Geschwindigkeiten entwickeln und in unterschiedlichen Versionen betrieben werden, ist es schwer eine Umgebung zu konstruieren, in der manuelle oder automatisierte Test durchgeführt werden und zu konsistenten Ergebnissen führen. Konzepte wie asynchrone Kommunikation und dynamisches Messaging erschweren das Testen weiter. Aus solch einer dynamischen Umgebung kann schwer erfassbares Verhalten hervorgehen und das Vertrauen in die Testergebnisse der Services mindern, die in Produktion gebracht werden sollen.

Kompromisse und Projektsituation

Die Datenarchitektur stellt im Bereich von verteilten Systemen eines der größten Herausforderungen dar. Neben der Frage, welcher Service für welche Information verantwortlich ist, ergibt

sich das Problem der Datenhaltung. Nachfolgend werden drei Ansätze mit ansteigender Isolation vorgestellt. Eine detaillierte Betrachtung dieses Themenkomplexes würde jedoch den Rahmen dieser Arbeit überschreiten.

Die erste Möglichkeit sieht vor, dass mehrere Services auf die gleiche Datenbank zugreifen, die Informationen jedoch in unterschiedlichen Tabellen abgelegt werden. Tabellen übergreifende Bedingungen, beispielsweise durch Primary und Foreign Key oder Transaktionen sind schnell hinzugefügt, koppeln die Services jedoch sehr stark miteinander. Außerdem wirken sich technische Probleme sofort auf das gesamte System aus.

Eine bessere Art der Isolation stellt das Verwenden unterschiedlicher Datenbankschema dar. Die Services greifen zwar auf einen gemeinsamen Datenbankservers zu, die Daten befinden sich jedoch in unterschiedlichen Schemata zu. Diese können mit speziellen Rechten versehen werden, der den Zugriff anderer Services verweigern. Technische Probleme des Datenbankservers wirken sich noch immer auf alle Services aus, die Wahlfreiheit der Datenbanktechnologie ist eingeschränkt. Außerdem kann es zu Konsistenzproblemen zwischen den Services kommen und den Bedarf zur Replikation von Daten hervorrufen. Dennoch ist dies ein durchaus valides Vorgehen. Die Transition eines Deployment-Monolith, mit gemeinsam genutzten Tabellen (und Spalten), hin zu einer Microservice-Architektur mit getrenntem Datenbestand, stellt Architekten und Entwickler vor eine ganz besondere Herausforderung. Der Weg über separate Schemata kann hier durchaus sinnvoll sein.

Die letzte Variante basiert, wie viele andere Lösungen im Microservice Umfeld auch, auf der maximalen Entkopplung. Jeder Service nutzt einen eigenen Datenbankserver. Ausgegangen von der zuvor vorgestellten Variante ist dies der nächst logische Schritt. Dies hat Vorteile im Bereich der Skalierung und ist Bestandteil autarker Microservices.

5.4 Praktische Anwendung

In dem folgenden Kapitel wird in einem praktischen Beispiel beschrieben, wie die in dieser Arbeit entwickelte Heuristik, im Kontext eines fiktiven Projektes, angewandt werden kann. Das Unternehmen, das Projekt und die Personen von denen berichtet wird, sind frei erfunden, aber doch realitätsnah.

5.4.1 Projektkontext

Webflix, eine Online-Videothek, ist vor wenigen Jahren in das Video-on-Demand-Geschäft eingestiegen und stellt seitdem Filme und Serien per Streaming für Abonnenten zu Verfügung. Im letzten Jahr sind die Nutzerzahlen der bereitgestellten Plattform stark angestiegen, dementsprechend stieg auch der Traffic und die Last der Server, auf denen sie betrieben wird. In dieser Zeit hat das Entwicklerteam unterschiedliche Techniken zur Skalierung der monolithischen JEE-Anwendung umgesetzt, dennoch stoßen die Maßnahmen immer wieder an ihre Grenzen. Neben den technischen Problemen ist zu bemerken, dass immer mehr Konkurrenten auf den noch relativ jungen Markt drängen. Webflix versucht mit dem Angebot (funktional) der Konkurrent schritt zu halten. Dies gestaltet sich jedoch immer schwere, da die Plattform eine beachtliche Größe und Komplexität erreicht hat.

John Doe, seines Zeichens Senior Software Engineer, ist vom Webflix CTO mit der Aufgabe betraut worden, zu prüfen, ob eine Microservice-Architektur für Webflix geeignet ist. John ließt von einer Heuristik zur Eignungsprüfung von Microservice-Architekturen und beschließt diese zu verwenden.

5.4.2 Architekturziele erfassen

John beschließt, mit einem an ATAM angelehnten Verfahren, die Architektur- bzw. Qualitätsziele der neuen Plattform zu erfassen. Er lädt Vertreter der verschiedenen Abteilungen und wichtige Stakeholder zu einem Meeting ein und schildert, dass es mit der Hilfe eines Quality Attribute Utility Trees die wichtigsten Qualitätsziels des neuen Systems identifizieren, priorisieren und präzisieren will. Nach zwei Terminen hat die Gruppe, bestehend aus teilweise unterschiedlichen Teilnehmern pro Termin, einen Utility Tree nach dem Bottom-Up und Bottom-Down-Prinzip erstellt. Wie in der Heuristik vorgeschlagen, orientiert sich die Gruppe an der ISO 25010 für Softwarequalität, lässt sich aber große Freiräume, um eigene Ziele zu formulieren.

Das Resultat ist ein klares und übereinstimmendes Verständnis der gemeinsamen Ziele aller Beteiligten. John fasst die Ziele mit absteigender Priorität in Leitsätzen zusammen:

- Verfügbarkeit - Das System soll immer uneingeschränkt verfügbar sein.

- Änderbarkeit - Das System soll leicht um neue Funktionalität erweiterbar sein.
- Benutzbarkeit - Alle Geräte sollen eine optimale User Experience vermitteln.
- Überwachbarkeit - Der Zustand des Systems soll zu jeder Zeit einsehbar sein.
- Sicherheit - Sensible Daten sollen sicher sein.

5.4.3 Architekturziele mit der Heuristik abgleichen

Im nächsten Schritt gleicht John die erfassten Architekturziele mit dem spezifischen Wirkungsmodell ab. Der Reihe nach ordnet er die erfassten Architekturziele den Mehrwerten des Modells zu. Für sein Verständnis einer hohen Verfügbarkeit sind Skalierbarkeit und Verlässlichkeit unabdingbar. Aus diesem Grund ordnet er Verfügbarkeit dem Mehrwert *elastische Skalierbarkeit der Anwendung* der Heuristik zu. So geht er für alle Architekturziele vor, die er zuordnen kann. Für Architekturziele, die er nicht zuordnen kann, studiert er die detaillierten Wirkungsmechanismen. Er versucht selbst Cluster aus Vorteilen einer Microservice-Architektur zu bilden, die sowohl direkt als auch indirekt zu dem Architekturziel *Benutzbarkeit* beitragen. Den Vorteilen leitet er Herausforderungen ab, um sie in einer Komplexität zu zusammenzufassen.

5.4.4 Mehrwerte und Komplexitäten verstehen

Zu diesem Zeitpunkt hat sich John bereits ein breites Wissen zum Thema Microservices angeeignet. Jetzt ist es an ihm, den Themenkomplex zu durchdringen. Er orientiert sich an den detaillierten Wirkungsmechanismen, um Ursache und Wirkung der Vorteile und Herausforderungen zu verstehen. Am Ende begreift er, wie sich Architekturentscheidungen auf die tatsächlich implementierte Architektur auswirken und Varianten von Microservice-Architektur entstehen.

5.4.5 Kompromisse und Projektsituation ermitteln

John versteht, wie Microservice-Architekturen seine Architekturziele unterstützen und mit welchen Komplexitäten er zu rechnen hat. Er betrachtet die Herausforderungen im Detail und versucht abzuschätzen, wie schwerwiegend diese für Webflix tatsächlich sind. Dazu bedient er sich zum einen der Erkenntnisse der Heuristik, zum anderen analysiert er den IST-Zustand

des Unternehmens und der aktuellen Plattform. Für Herausforderungen, die er als sehr groß einschätzt, versucht er Kompromisse zu finden, die das Erreichen der Architekturziele nicht gefährden.

5.4.6 Entscheiden

Am Ende steht für John fest, dass eine Microservice-Architektur für die Plattform von Webflix ein geeigneter Architekturstil ist. Er hat ermittelt, wie alle Architekturziele auf der Basis von Microservices erreicht werden können. Außerdem sorgen Kompromisse und eine vorteilhafte Ausgangssituation dafür, dass die Herausforderungen mit einem vertretbaren Aufwand bewältigt werden können.

Kapitel 6

Fazit und Ausblick

Ziel der vorliegenden Arbeit war die Erstellung einer Heuristik. Sie soll Entscheidern, oder den damit beauftragten Personen, als Werkzeug und Leitfaden dienen, um bewerten zu können, ob eine Microservice-Architektur für den jeweiligen Projektkontext eine sinnvolle Lösung ist. Bereits bei der theoretischen Einführung in das Themengebiet Microservices wurde deutlich, dass es sich dabei um keine spezifische Architektur handelt. Die abweichenden Definitionen aus Literatur und Experteninterviews, sowie die in Kapitel 2.6 beschriebenen Varianten zeigen, dass es sich bei dem Microserviceparadigma viel mehr um eine lose Ansammlung von Konzepten und Prinzipien handelt. Diese umfassen sowohl technisch, wie auch organisatorische Aspekte. Je nach Anforderung kann sich diesem Werkzeugkasten bedient werden, um eine Microservice-Architektur zu entwerfen. Nur einige wenige Eigenschaften sind durchgehend in allen Varianten zu finden.

Dies zeigt, dass es nicht um eine Entscheidung für oder gegen Microservices handelt. Es muss eine ganze Reihe an Architekturentscheidungen bewertet und getroffen werden. Die Experten wurden unter anderem zum persönlichen Entscheidungsprozess befragt. Auf der Basis von FMEA haben letztendlich die Gemeinsamkeiten dieser individuellen Prozesse und das formalisierte Vorgehen von ATAM zur Entwicklung des in Kapitel 5.1 beschriebenen allgemeinen Wirkungsmodells geführt. Wünschenswert wäre es zu untersuchen, inwiefern dieses Modell zur Bewertung jeglicher Art von Architekturentscheidung angewandt werden kann.

Da dieses Modell nicht unmittelbar zur Bewertung sehr komplexer Architekturentscheidungen führen kann, wurde es mit microservicespezifischen Variablen, den Vorteilen und Herausforderungen, angereicht und auf zwei unterschiedlichen Abstraktionsebenen dargestellt. Daraus resultiert ein besseres Verständnis der komplexen Zusammenhänge.

Abschließend lässt sich nicht eindeutig klären, ob die erstellte Heuristik zur Beantwortung der zentralen Fragestellung ausreicht, da keine empirische Untersuchung durchgeführt wurde. Dies wäre eine durchaus lohnenswerte Aufgabe für zukünftige Untersuchungen. Aufgrund der nachvollziehbaren Struktur der spezifischen Wirkungsmodelle kann die Heuristik zumindest als Leitfaden dienen, um sich der Beantwortung der Frage strukturiert anzunähern.

Literatur

Abedrabbo, Tareq. *The 7 Deadly Sins of Microservices*. (Abgerufen am 19.12.2016). OpenCredo. Nov. 2014. URL: https://opencredo.com/7-deadly-sins-of-microservices/.

Babar, Muhammad Ali und Ian Gorton. „Experiences from Scenario-Based Architecture Evaluations with ATAM“. In: *Software Architecture: 4th European Conference, ECSA 2010, Copenhagen, Denmark, August 23-26, 2010. Proceedings*. Springer-Verlag Heidelberg, 2010, S. 214–229. ISBN: 978-3642151149.

Bass, Len u. a. *Software Architecture in Practice*. 2. Auflage. (Abgerufen am 24.07.2016). Milwaukee, Wisconsin: Addison-Wesley Professional, 2003. ISBN: 0321154959. URL: http://www.ece.ubc.ca/~matei/EECE417/BASS/ch11lev1sec3.html.

Ben-Daya, Mohamed. *Handbook of maintenance management and engineering*. Springer, 2009. ISBN: 978-1-8488-2472-0.

Bente, Stefan. *Grundlagen des Strategischen IT-Managements*. Vorlesung. 2014.

Bossert, Oliver u. a. *A two-speed IT architecture for the digital enterprise*. (Abgerufen am 19.10.2016). McKinsey & Company. Dez. 2014. URL: http://www.mckinsey.com/business-functions/digital-mckinsey/our-insights/a-two-speed-it-architecture-for-the-digital-enterprise.

Brown, Alanna. *What's the Best Team Structure for DevOps Success?* (Abgerufen am 14.12.2016). Puppet. Juli 2015. URL: https://puppet.com/blog/what%E2%80%99s-best-team-structure-for-devops-success.

Carnegie Mellon University. *Architecture Tradeoff Analysis Method*. (Abgerufen am 21.07.2016). Software Engineering Institute. URL: http://www.sei.cmu.edu/architecture/tools/evaluate/atam.cfm.

Christensen, Clayton M. *The Innovator's Dilemma: When New Technologies Cause Great Firms to Fail*. Harvard Business Review Press, 1997. ISBN: 978-0875845852.

Die optimale Teamgröße. (Abgerufen am 08.12.2016). Scrum Akademie. Mai 2016. URL: http://www.scrumakademie.org/die-optimale-teamgroesse/.

Eberhard Wolff, Autor bei JAXenter. (Abgerufen am 16.11.2016). JAXenter. URL: https://jaxenter.de/author/eberhardwolff.

Evans, Eric J. *Domain-Driven Design: Tackling Complexity in the Heart of Software*. 1. Auflage. Addison Wesley, 2003. ISBN: 978-0321125217.

Farcic, Viktor. „Self-Healing Systems: Wie man Microservices-basierten Systemen die Selbstheilung beibringt". In: *DevOpsCon - Berlin*. Juni 2016. URL: https://www.youtube.com/watch?v=rDtihwJjE_k.

Fletcher, Matt. *Service-Based Architecture as an Alternative to Microservice Architecture*. (Abgerufen am 11.10.2016). Okt. 2016. URL: https://www.infoq.com/news/2016/10/service-based-architecture.

Big Ball of Mud. 1999. URL: http://www.laputan.org/pub/foote/mud.pdf.

Ford, Neal. „Comparing Servicebased Architectures". In: *Über-conf*. Juli 2016. URL: http://nealford.com/downloads/Comparing_Service-based_Architectures_by_Neal_Ford.pdf.

Fowler, Martin. *Is Design Dead?* (Abgerufen am 30.10.2016). Mai 2004. URL: http://martinfowler.com/articles/designDead.html.

– *Microservices*. (Abgerufen am 18.12.2016). Juli 2005. URL: http://martinfowler.com/bliki/ServiceOrientedAmbiguity.html.

– *Microservices*. (Abgerufen am 10.08.2016). März 2014. URL: http://martinfowler.com/articles/microservices.html.

– „Microservices". In: *GOTO Conference*. Nov. 2014. URL: https://www.youtube.com/watch?v=wgdBVIX9ifA.

– *Microservices Guide*. (Abgerufen am 14.07.2016). URL: http://martinfowler.com/microservices/#what.

Friebertshäuser, Barbara und Annedore Prengel. *Handbuch Qualitative Forschungsmethoden in der Erziehungswissenschaft*. 1997.

Google Trends. *Websuche-Interesse: microservices - Weltweit, Jan. 2013 - Nov. 2016*. URL: https://www.google.de/trends/explore?cat=5&date=2013-01-01%202016-11-23&q=microservices.

Gourévitch, Antoine u. a. *Two-Speed IT: A Linchpin for Success in a Digitized World*. (Abgerufen am 18.12.2016). The Boston Consulting Group. Aug. 2012. URL: https://www.bcgperspectives.com/content/articles/it_performance_it_strategy_two_speed_it/.

Hornauer, W. „Industrielle Automatisierungstechnik“. In: *Mitteilungen aus der Arbeitsmarkt- und Berufsforschung*. (Abgerufen am 01.12.2016). W. Kohlhammer GmbH, 1968, S. 34. URL: http://doku.iab.de/mittab/1968/1968_02_MittAB_Ulrich.pdf.

Hruschka, Peter und Gernot Starke. *Knigge für Softwarearchitekten: Wider die IT der zwei Geschwindigkeiten*. (Abgerufen am 20.10.2016). JAXenter. Okt. 2016. URL: https://jaxenter.de/knigge-bimodale-it-48295.

innoQ. *Self-Contained Systems - Assembling Software from Independent Systems*. (Abgerufen am 06.10.2016). URL: http://scs-architecture.org/index.html.

Jardine, Alastair. *"Do what you want": building great products through anarchy*. (Abgerufen am 05.10.2016). The Guardian. Feb. 2015. URL: https://www.theguardian.com/info/developer-blog/2015/feb/09/do-what-you-want-building-great-products-through-anarchy.

Kazman, Rick, Abowd u. a. „SAAM: A Method for Analyzing the Properties of Software Architectures“. In: *Proceedings of the 16th International Conference on Software Engineering*. 1994, S. 81–90.

Kazman, Rick, Mark Klein u. a. *ATAM: Method for Architecture Evaluation*. Techn. Ber. Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pennsylvania, 2000.

Kunlaboro – operatives DevOps bei den Barmenia-Versicherungen. (Abgerufen am 16.11.2016). Softwareforen Leipzig. URL: http://www.softwareforen.de/portal/de/unternehmen/aktuellenachrichten_5/aktuellenachrichtendetail_69632.xhtml.

Lilienthal, Carola. *Wie Sie technische Schulden in Architekturen abbauen lassen*. (Abgerufen am 10.12.2016). JAXenter. Juni 2016. URL: https://jaxenter.de/wie-sie-technische-schulden-in-architekturen-abbauen-41042.

Meuser, Michael und Ulrike Nagel. „ExpertInneninterviews - vielfach erprobt, wenig bedacht : ein Beitrag zur qualitativen Methodendiskussion“. In: *Qualitativ-empirische Sozialforschung : Konzepte, Methoden, Analysen*. (Abgerufen am 11.07.2016). Westdeutscher Verlag, 1991, S. 441–471. ISBN: 3-531-12289-4. URL: http://www.ssoar.info/ssoar/bitstream/handle/document/2402/ssoar-1991-meuser_et_al-expertinneninterviews_-_vielfach_erprobt.pdf?sequence=1.

Morris, Ben. *How big is a microservice?* (Abgerufen am 15.09.2016). März 2015. URL: http://www.ben-morris.com/how-big-is-a-microservice/.

Nathe, Elmar. *Warum die IT Sportwagen und LKW zugleich sein muss*. (Abgerufen am 20.10.2016). Computerwoche. Juni 2016. URL: http://www.computerwoche.de/a/warum-die-it-sportwagen-und-lkw-zugleich-sein-muss,3312726.

Nestler, Franz. *Googles Weg zur Sparkasse*. (Abgerufen am 17.10.2016). Frankfurter Allgemeine Finanzen. Juni 2014. URL: http://www.faz.net/aktuell/finanzen/banklizenz-fuer-internetgeschaefte-googles-weg-zur-sparkasse-13005707.html.

Newman, Sam. *Microservices (mitp Professional): Konzeption und Design*. 1. Auflage. Frechen: mitp, 2015. ISBN: 978-3-95845-081-3.

Omdahl, Tracy P., Hrsg. *Reliability, availability, and maintainability (RAM) dictionary*. ASQC quality press, 1988.

Parnas, David. „On the criteria to be used in decomposing systems into modules“. In: *Communications of the ACM* Vol. 15 (1972), S. 1053–1058.

Plöd, Michael. „Microservices lieben Domain-driven Design“. In: *Entwickler Magazin Spezia* Vol. 9 (2016), S. 22–25.

Prokein, Oliver. *IT-Risikomanagement - Identifikation, Quantifizierung und wirtschaftliche Steuerung*. 1. Auflage. Milwaukee, Wisconsin: Gabler Verlag, 2008. ISBN: 978-3-8349-0870-4.

Stamatis. *Failure Mode and Effect Analysis - FMEA from Theory to Execution*. 2. Auflage. Milwaukee, Wisconsin: ASQ Quality Press, 2003. ISBN: 0-87389-598-3.

Stefan Toth, Autor bei JAXenter. (Abgerufen am 16.11.2016). JAXenter. URL: https://jaxenter.de/author/stefantoth.

Steinacker, Guido. *Refactoring*. (Abgerufen am 09.11.2016). Wikipedia. URL: https://de.wikipedia.org/wiki/Refactoring.

– *Why Microservices?* (Abgerufen am 27.10.2016). März 2016. URL: https://dev.otto.de/2016/03/20/why-microservices/.

Technology Radar - Microservice envy. (Abgerufen am 02.12.2016). ThoughtWorks. Nov. 2015. URL: https://www.thoughtworks.com/de/radar/techniques/microservice-envy.

Toth, Stefan. „Evolution statt Diktatur“. In: *Entwickler Magazin Spezia* Vol. 9 (2016), S. 12–15.

– *Vorgehensmuster für Softwarearchitektur*. München: Carl Hanser Verlag, 2014. ISBN: 978-3-446-43762-3.

Treacy, Michael und Fred Wiersema. *Customer Intimacy and Other Value Disciplines*. (Abgerufen am 19.10.2016). Harvard Business Review. Feb. 1993. URL: https://hbr.org/1993/01/customer-intimacy-and-other-value-disciplines.

Williams, Craig. *Is REST Best in a Microservices Architecture?* (Abgerufen am 20.09.2016). Capgemini. Dez. 2015. URL: https://capgemini.github.io/architecture/is-rest-best-microservices/.

Wolff, Eberhard. *Microservices 101: Varianten, Herausforderungen, Erfahrungen aus der Praxis*. (Abgerufen am 10.07.2016). JAXenter. Mai 2016. URL: https://jaxenter.de/microservices-101-varianten-herausforderungen-erfahrungen-aus-der-praxis-40571.

– *Microservices: Grundlagen flexibler Softwarearchitekturen*. 1. Auflage. Heidelberg: dpunkt.verlag, 2016. ISBN: 978-3-86490-313-7.

– „Microservices: Weg vom Hype - rein in die Praxis!“ In: *Entwickler Magazin Spezial* Vol. 9 (Sep. 2016), S. 3.

– „Schein und Sein“. In: *Entwickler Magazin Spezia* Vol. 9 (2016), S. 7–10.

– „Self-contained Systems: Microservices-Architektur mit System“. In: *rheinjug - Düsseldorfer Java User Group - Event*. Okt. 2016.

Zörner, Stefan. „Bring your own Architecture“. In: *Entwickler Magazin Spezia* Vol. 9 (2016), S. 16–19.

Anhang A

Leitfaden für Experteninterviews

A.1 Einleitung

Erst einmal möchte ich mich dafür bedanken, dass sie mir die Gelegenheit für dieses Interview geben.

Ich studiere an der TH Köln und schreibe derzeit an meiner Masterarbeit mit dem Titel: „Eine Heuristik zur Eignungsprüfung von Microservice-Architekturen“. In dieser Arbeit wird ein Vorgehen erarbeitet, mit dem die Entscheidung einer Neuentwicklung auf Basis von Microservices (MS) oder der Transition von einem Deployment-Monolithen hin zu einer MS Architektur evaluiert werden kann. Mit dieser Heuristik möchte ich Entscheidern helfen sich die richtigen Fragen mit Bezug zum Projektkontext zu stellen, um dann eine fundierte Entscheidung treffen zu können. Bei dieser Arbeit werde ich von Prof. Dr. Stefan Bente der TH Köln und Dr. Thomas Franz der adesso AG unterstützt.

Das Interview selbst dient der Theorieentwicklung und behandelt Ihre Projekterfahrungen bezüglich MS.

Für das Interview habe ich ca. eine Stunde geplant. Zur Auswertung des Interviews möchte ich unser Gespräch gerne aufnehmen. Sind Sie damit einverstanden?

Sollte etwas veröffentlicht werden, so werden die hier gewonnen Informationen anonymisiert und vertraulich behandelt.

Das Interview ist thematisch wie folgt gegliedert:

1. Projektkontext
2. Organisationsstruktur
3. Technologien
4. Betrieb
5. Entscheidungsfindung
6. Vorteile und Herausforderungen

Haben Sie noch Fragen bevor wir beginnen?

A.2 Leitfaden

1. Bei welchem Projekt haben Sie zuletzt MS eingeführt?	Einführung
1.1. Wer war der Kunde?	Branche
1.2. Was leistet die Software?	Einordnung des Projektes
1.3. Welche Qualitätsanforderungen hat das Projekt?	Qualitätsanforderungen
1.4. Wie groß ist der geplante Funktionsumfang?	Größe des Projektes
1.5. Ist der Funktionsumfang final?	Evolution des P.
1.6. Wie lange soll die Software planmäßig betrieben werden?	Betriebsdauer
2. Wie ist das Team organisiert?	Organisation
2.1. Wie groß ist das Team?	Größe
2.2. Wonach sind die Teams strukturiert?	Feature, Technol., Domain.
2.3. Aus welchen Positionen setzt sich das Team zusammen?	DevOps
2.4. Welche Entscheidungen darf das Team treffen?	Handlungsfähigkeit
2.5. Welches Vorgehensmodell wird eingesetzt?	Agil, Wasserfall
3. Welche Technologien werden eingesetzt?	Technologie Stack
3.1. Wurden die Technologien auf Basis des Problems gewählt oder gab es bestimmte Richtlinien?	Vorschriften
3.2. Hat das Team Vorkenntnisse im Bereich MS?	Know How
3.3. Wie ist das Know How des Teams zu querschnittlichen Themen verteilt?	Know How
4. Wie wird die Software betrieben?	Technologien, Betrieb
4.1. Wie funktioniert die Infrastruktur?	Infrastructure as Code, Docker
4.2. Wie sieht der IT Delivery Prozess aus?	CD
4.3. Wie wird die Software getestet?	Testing

5. Wie sind Sie zu der Entscheidung gekommen, dass eine MS Architektur angewandt werden soll?	Entscheidungsfindung
5.1. Welche Faktoren haben die Entscheidung (wie) beeinflusst?	Faktoren
5.2. Welche Rolle spielten die Qualitätskriterien?	Qualitätskriterien
5.3. Welche Rolle spielten die MS Vorteile?	Vorteile
5.4. Welche Rolle spielten die MS Herausforderungen?	Herausforderungen
5.5. Wie wurden die Kosten bewertet?	Kosten
5.6. Gab es Ziele die erreicht werden sollten, bevor man die Architektur der Software umgesetzt hat?	Vorbedingungen
6. Welche Vorteile die MS mit sich bringen sind für dieses Projekt die größten und warum?	Vorteile
6.1. Gab es Vorteile, von denen man sich viel versprochen hat, welche sich allerdings als nicht sehr wirkungsvoll herausgestellt haben?	Überschätzt Vorteile
6.2. Gab es Vorteile, die nicht in Kraft getreten sind?	Nicht eintretende Vorteilen
7. Welche Herausforderungen die MS mit sich bringen sind für dieses Projekt die größten und warum?	Herausforderungen
7.1. Konnten alle Herausforderungen bewältigt werden?	Nicht bewältigte Herausf.
7.2. Gab es unerwartete Herausforderungen?	Unerwartete Herausf.
7.3. Gab es eine geplante Reihenfolge in der die Herausforderungen angegangen wurden?	Vorbedingungen/Reihenfolge
7.4. Hätten die Herausforderungen durch eine bessere Planung leichter bewältigt werden können?	Vorbedingungen
8. Wenn Sie das Projekt erneut beginnen könnten, was würden Sie jetzt anders bzw. gleich angehen?	Rückblick
9. Möchten Sie abschließend noch Aspekte nennen, die aus Ihrer Sicht im Zusammenhand mit der Entscheidungsfindung für bzw. gegen MS betrachtet werden sollten?	Weiteres

Anhang B

Architecture Tradeoff Analysis Method - Vorgehen im Detail

Die Evaluation mittels ATAM findet in einem mehrtägigen Workshop mit wechselnden Teilnehmern statt. Insgesamt besteht der gesamte Prozess aus vier Phasen.

Phase	**Aktivität**	**Teilnehmer**	**Dauer**
0	Vorbereitung	Leiter Evaluationsteam und Hauptentscheider des Projektes	Einige Wochen
1	Evaluation	Evaluationsteam und Entscheider	1 Tag
2	Evaluation	Evaluationsteam, Entscheider und Stakeholder	2 Tage
3	Resultate	Evaluationsteam und Teilnehmer	1 Woche

TABELLE B.1: Übersicht aller Phasen des ATAM Prozesses (Quelle: Bass u. a., *Software Architecture in Practice*)

Phase 0: Vorbereitung

In Phase 0 "Vorbereitung" wird das Evaluationsteam von den wichtigsten Entscheidern in das Projekt eingewiesen, sodass das Team um die benötigten Kompetenzen erweitert werden kann.

Die Teams einigen sich auf Deadlines und Verantwortlichkeiten, Stakeholder werden Namentlich und nicht nur anhand der Rolle benannt und es werden Architekturdokumentationen ausgetauscht, welche für die Evaluation genutzt werden können. Abschließend erklärt das Evaluationsteam welche Informationen von den Managern und Architekten in der kommenden Phase 1 erwartet werden. Dies kann in Form einer Präsentationsvorlage erfolgen.

Phase 1 und 2: Evaluation

Diese beiden Phasen bestehen aus insgesamt acht Schritten in denen die eigentliche Evaluation der Architektur stattfindet. Die beiden Phasen bauen aufeinander auf und finden an aufeinanderfolgenden Tagen mit unterschiedlichen Teilnehmern statt (siehe Tabelle B.1 "Übersicht aller Phasen des ATAM Prozesses (Quelle: Bass u. a., *Software Architecture in Practice*)").

In **Phase 1**, am ersten Tag, führten das Evaluationsteam und technisch-orientierte Stakeholder die Schritte 1 bis 6 durch. Die Gruppe verfolgt bei der Erstellung des Utility Trees den Top-Down Ansatz. Auf diese Weise werden von den Qualitätszielen konkrete Qualitätsattribute und Szenarien abgeleitet.

1. **Präsentation von ATAM.**
   In diesem Schritt stellen die Leiter des Evaluationsteams den anwesenden Stakeholdern die Methode ATAM vor. Die Prozessschritte werden erklärt und Fragen beantwortet. Es ist wichtig, dass die Teilnehmer wissen welche Informationen zusammengetragen, wie sie untersucht und an wen die Ergebnisse geliefert werden.[1]

2. **Vorstellung der unternehmerischen Einflussfaktoren.**
   Um die Evaluation erfolgreich durchführen zu können, müssen die Teilnehmer das System verstehen. In diesem Schritt wird die Architektur in einem hohen Abstraktionsgrad aus Unternehmenssicht präsentiert. Dabei sollten die wichtigsten Funktionen, technische, politische oder historische Rahmenbedingungen, Unternehmensziele und Stakeholder zur Sprache gebracht werden.[2]

3. **Präsentation der Architektur.**
   Es folgt eine Präsentation der entwickelten oder entworfenen Softwarearchitektur durch

[1] vgl. Kazman, Klein u. a., *ATAM: Method for Architecture Evaluation*, S. 25.
[2] vgl. ebd., S. 26.

den Architekten oder das verantwortliche Team. In dieser Präsentation sollten vor allem technische Rahmenbedingungen, Architekturentscheidungen (Patterns) und wie diese zur Erreichung der Architekturziele beitragen zur Sprache kommen. Als Kommunikationsmittel bieten sich Diagramme technischer Sichten mit einem angemessenen Abstraktionsgrad an.[3]

4. **Identifizieren der architektonischen Lösungsansätze.**
   ATAM stützt sich bei der Analyse einer Architektur auf das Verständnis der Architekturentscheidungen bzw. der genutzten Pattern, denn diese wirken sich auf bestimmte Qualitätsziele aus. An dieser Stelle hat das Evaluationsteam durch das Studieren der Dokumentation und der Präsentation des Architekten ein gutes Verständnis der Architektur. In diesem kurzen Schnitt werden die identifizierten Lösungsansätze in einer Liste erfasst und dienen der weiteren Analyse als Basis.[4]

5. **Erstellung des Quality Attribute Utility Trees.**
   An dieser Stelle werden die Qualitätsziele explizit genannt und definiert. Dazu wird eine Technik namens Utility Tree genutzt, mit der die wichtigsten Qualitätsziele identifiziert, priorisiert und präzisiert werden.

   Die Wurzel des Baumes bildet die *utility*, welche die Güte des Systems beschreibt. Auf der ersten Ebene werden die Qualitätsziele aus Schritt zwei aufgeführt, denn die Güte des Systems besteht aus ihnen. Typischerweise werden Attribute wie Performance, Modifiability, Security, Usability oder Availability genannt. Die Teilnehmer können jedoch auch eigene Ziele nennen, denn es kommt vor, dass unterschiedliche Benutzergruppen verschiedene Begriffe für die gleiche Eigenschaft nennen oder Qualitätsattribute erst in dem jeweiligen Kontext, aber nicht in allen anderen, sinnvoll sind.

[3] vgl. Kazman, Klein u. a., *ATAM: Method for Architecture Evaluation*, S. 27 f.
[4] vgl. ebd., S. 28.

Auf der dritten Ebene werden die Qualitätsziele explizit gemacht. Qualitätsziele wie Performance werden in *Data Latency* und *Transaction Throughput* zerlegt. Auf der darauffolgenden Ebene werden die Zielattribute mit Szenarien konkretisiert um priorisiert und analysiert werden zu können. Data Latency könnte mit den Szenarien "Senkung der Dauer des Datenspeicherns auf unter 20 Millisekunden" und "Ausliefern eines 20 Frame/Sekunden Videos in Echtzeit". Diese Szenarien sind explizit genug, um getestet zu werden. In den nachfolgenden Schritten wird für jedes Szenario geprüft, wie die Architektur darauf reagiert oder ob sie das Ziel erreicht.

Da auf diese Weise eine ganze Reihe an Szenarien entstehen kann, müssen diese priorisiert werden. Dies geschieht auf zwei Ebenen. Zunächst einigen sich die Entscheider darauf, wie wichtig ein Attribut ist. Hier bietet sich eine Bewertung nach hoch H (high), mittel M (medium) und gering L (low). Anschließend bewerten die Architekten wie schwer ein Ziel erreichbar ist. Szenarien welche mit H,H gekennzeichnet sind, sollten während der Analyse im Fokus stehen.[5]

Abbildung B.1 zeigt beispielhaft den Auszug eines Utility Trees.

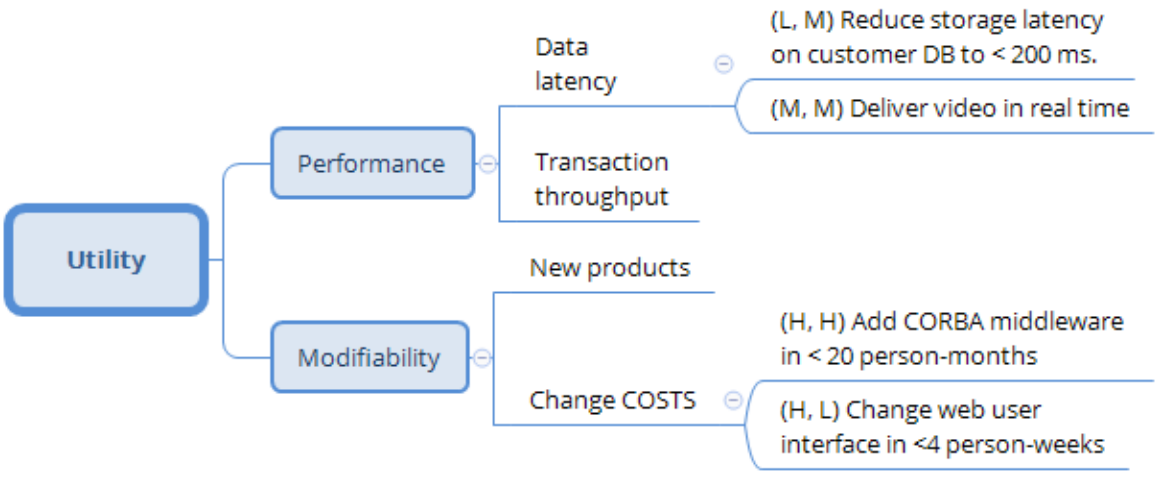

ABBILDUNG B.1: Auszug eines möglichen Utility Trees (Quelle: Kazman, Klein u. a., *ATAM: Method for Architecture Evaluation*, S. 17)

[5]vgl. Kazman, Klein u. a., *ATAM: Method for Architecture Evaluation*, S. 29.

6. **Analysieren der architektonischen Lösungsansätze.**
   In diesem Schritt begründet der Architekt oder das Team für jedes Szenario, wie die Architektur dieses unterstützt. Auf dem Weg werden alle relevanten Architekturentscheidungen dokumentiert. Das Abarbeiten der Szenarien führt zu Diskussionen in denen Risiken (Risks), Nicht-Risiken (Nonrisks), Punkte, welche besonders beachtet werden müssen (Sensitivity Points) und Kompromisse (Tradeoffs) erfasst und katalogisiert werden können.[6]

Am zweiten Tag, in **Phase 2**, besteht das Team aus einer größeren Gruppe aus Stakeholdern und ist Stakeholder-orientiert. Anfangs fasst das Evaluationsteam die Schritte 1 - 6 des Vortages zusammen und präsentiert die Ergebnisse dem Team. Anschließend wird der Utility Tree nach dem Bottom-Up-Prinzip mit Szenarien angereichert, welchen dann wiederum Qualitätsattributen zugewiesen werden.

7. **Brainstorming und Priorisierung von Szenarien.**
   Der Zweck von Szenarien-Brainstorming ist den Impuls einer größeren Gruppe aus Stakeholdern zu nutzen. Das Evaluationsteam bitte die Stakeholder sich Szenarien zu überlegen, welche aus ihrer jeweiligen Sicht von Interesse sind. Ein Mitarbeiter des Betriebs wird andere Szenarien entwickeln als ein Endbenutzer. Anschließend werden die Szenarien den Qualitätsattributen zugeordnet. Passen diese gut zusammen, kann davon ausgegangen werden, dass die Ziele des Architekten mit den Bedürfnissen der Stakeholder übereinstimmen.[7]

8. **Analysieren der architektonischen Lösungsansätze.**
   Dieser Schnitt ist äquivalent zu Schnitt 6. Die neu erfassten Szenarien werden auf die gleiche Weise analysiert.[8]

Phase 3: Resultate

In der letzten Phase werden die Ergebnisse der Evaluation präsentiert und an den Auftraggeber

[6] vgl. Kazman, Klein u. a., *ATAM: Method for Architecture Evaluation*, S. 29 - 33.
[7] vgl. ebd., S. 33 - 36.
[8] vgl. ebd., S. 36 f.

übergeben. Bestandteile eines solchen Reports sind die Szenarios, die Attribut-spezifischen Fragen, der Utility Tree, die Risks, die Sensitivity Points und die Trade-Offs.

Neben der Übergabe von Ergebnissen führt das Evaluationsteam eine Selbstreflexion durch, sodass zukünftige Evaluationen effizienter durchgeführt werden können. Dies kann mit Hilfe eines Fragebogens unterstützt werden, welcher durch die beteiligten Personen ausgefüllt wird.[9]

[9] vgl. Kazman, Klein u. a., *ATAM: Method for Architecture Evaluation*, S. 37 f.

Erklärung über die Selbständige Abfassung der Arbeit

Ich versichere an Eides Statt, die von mir vorgelegte Arbeit selbständig verfasst zu haben. Alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten oder nicht veröffentlichten Arbeiten anderer entnommen sind, habe ich als entnommen kenntlich gemacht. Sämtliche Quellen und Hilfsmittel, die ich für die Arbeit benutzt habe, sind angegeben. Die Arbeit hat mit gleichem Inhalt bzw. in wesentlichen Teilen noch keiner anderen Prüfungsbehörde vorgelegen.

_____

Datum, Ort

_____

Unterschrift