MASTERARBEIT

MODELLIERUNG VON MICROSERVICES

Welche Informationen sind für eine Microservice-Architektur von Relevanz und sollten als Modell sichtbar gemacht werden?

ausgeführt an der

am Studiengang
Software Engineering Leadership

Von: Olcay Tümce
Personenkennzeichen: 1420030012

Langenhagen, am 28.07.2017

.....

Unterschrift

EHRENWÖRTLICHE ERKLÄRUNG

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

.....

Unterschrift

DANKSAGUNG

An dieser Stelle möchte ich mich bei allen bedanken, die mich während der Anfertigung dieser Masterarbeit motiviert und unterstützt haben.

Zuerst möchte ich mich herzlichst bei meinem Betreuer Hermann Woock bedanken, der meine Masterarbeit betreut und begutachtet hat. Die Unterstützung bei der Themenfindung, Gliederung bis hin zu inhaltlichen Hilfestellungen hat meine Arbeit sehr bereichert.

Ein besonderer Dank gilt auch allen Teilnehmern und Teilnehmerinnen der Befragung und Interviews, ohne die diese Arbeit nicht hätte entstehen können. Mein Dank gilt ihrer Informationsbereitschaft und ihren interessanten Beiträgen und Antworten auf meine Fragen.

Ebenfalls möchte ich mich bei allen bedanken, die mir mit zahlreichen interessanten Debatten und Ideen zur Seite standen. Ein besonderer Dank gilt hier Guido Steinacker, Marco Ehrentreich und Eberhard Wolff.

Des Weiteren bedanke ich mich bei meiner Familie, allen Kollegen und Freunden, die durch Diskussionen und ihrer Meinung zur Masterarbeit beigetragen haben.

Mein besonderer Dank gilt meiner Freundin Jill, die mich durch die Wertschätzung der Masterarbeit und die notwendigen zeitlichen Freiräume bei der Erstellung der Masterarbeit unterstützt hat.

KURZFASSUNG

Immer mehr Unternehmen setzen ihre Systeme mit Microservices um, da sie ihre Anwendungen schnell, agil und unabhängig in Produktion bringen möchten. So komplex dieser Architekturansatz ist, so vielfältig sind die Informationen, die für diesen Ansatz benötigt werden. Die vorliegende Arbeit beschäftigt sich daher damit, welche Aspekte für eine Microservice-Architektur von Relevanz sind und als Modell sichtbar gemacht werden sollten. Hierbei führt die Arbeit zunächst die allgemeinen Grundlagen der Softwarearchitektur und Modellierung ein.

Anschließend werden auf Basis einer Literaturrecherche die relevanten Informationen einer Microservice-Architektur hervorgehoben. Die Kernelemente dabei sind die Makro- und Mikroarchitektur, die fachliche Aufteilung mit Domain-driven Design sowie die infrastrukturellen Herausforderungen. Aufgrund der sich ständig dynamisch ändernden Servicelandschaft, ist es in einer Microservice-Architektur außerdem erforderlich, sich mithilfe von automatisierter Datensammlung einen Überblick zu verschaffen.

Anhand einer empirischen Studie konnte herausgefunden werden, dass die Unternehmen bereits diese relevanten Daten überwiegend über das Monitoring, Service Discovery oder Logging sammeln. Auch konnte herausgefunden werden, dass es Tendenzen bei den Aufgaben gibt, welche in der Praxis zur Makroarchitektur zugeordnet wurden.

Im weiteren Verlauf der Arbeit wurde untersucht, inwieweit sich eine Microservice-Architektur mit klassischer Architekturbeschreibung wie arc42 beschreiben lässt. Aufgrund der allgemeinen Ausrichtung des Templates wurde prototypisch ein Modell entwickelt, welches die für eine Microservice-Architektur relevanten Entscheidungsfragen bereitstellt und somit bei den konkreten Entscheidungen unterstützt.

Schließlich wurde das Modell mithilfe von qualitativen Interviews evaluiert. Hierbei wurde als Ergänzung ein Experte herangezogen, um mögliche Verbesserungen aus einer weiteren Perspektive miteinbringen zu können. Die Ergebnisse zeigten, dass das Modell zwar eine gute Basis darstellt, um einen Überblick zu den wichtigsten Entscheidungsfragen zu erhalten, aber dennoch weitere zielgruppenspezifische Informationen und detailliertere Ausführungen grundlegender Konzepte benötigt werden.

ABSTRACT

An increasing number of companies are using microservices, as they want to bring their applications into production quickly and in an agile and independent manner. This architectural approach is complex and the information required for this approach is complicated. This paper thus deals with which aspects are relevant for a microservice architecture and should be made visible as a model. In doing so, the paper initially deals with the general fundamentals of software architecture and modelling.

It then goes on to highlight the relevant information for a microservice architecture based on literature research. The core elements in this regard are the macroarchitecture and the microarchitecture, the specialist breakdown with domain-driven design and the infrastructure requirements. As a result of the fact that the service landscape is the subject of constant, dynamic change, it is also necessary for a microservice architecture to create an overview using automated data collection.

It was possible to use an empirical study to find out that companies already collect this relevant data, mostly using monitoring, service discovery or logging. It was also possible to find out that there are tendencies in the tasks, which can be allocated to macroarchitecture in practice.

The paper then investigates the extent to which a microservice architecture can be described using a traditional architecture description such as arc42. As a result of the template's general orientation, a prototype model has been developed which provides the relevant decision-making questions for a microservice architecture, and thus provides support in making correct decisions.

Finally, the model was evaluated using quality-based interviews. In so doing, an expert was also used to supplement the results and to also contribute possible improvements from a different perspective. Results showed that the model provided an excellent basis to obtain an overview of the key decision-making questions, however that additional target group-specific information and more detailed information on fundamental concepts are also needed.

GLEICHHEITSGRUNDSATZ

Aus Gründen der Lesbarkeit wurde in dieser Arbeit darauf verzichtet, geschlechtsspezifische Formulierungen zu verwenden. Jedoch möchte ich ausdrücklich festhalten, dass die bei Personen verwendeten maskulinen Formen für beide Geschlechter zu verstehen sind.

INHALTSVERZEICHNIS

1 EINLEITUNG

Die vorliegende Arbeit beschäftigt sich mit dem Thema „Modellierung von Microservices“. In dieser Einleitung soll als erstes im Abschnitt 1.1 dargelegt werden, aus welcher Motivation heraus das Thema wissenschaftlich untersucht wird. Anschließend werden in Abschnitt 1.2 die zentrale Fragestellung dargelegt und die Ziele der Arbeit erläutert. In Abschnitt 1.3 wird beschrieben, wie die Arbeit aufgebaut ist und mit welchen Methoden die Fragestellung untersucht wird.

1.1 Motivation

Die Softwareentwicklung ist stetig im Wandel. Jedes Jahr entstehen neue Technologien und Vorgehensweisen, die versprechen Software noch schneller auf den Markt zu bringen. Den Rahmen hierfür bilden agile Vorgehensmodelle. Mit ihrer Hilfe kann in kurzen Release-Zyklen nützliche und qualitativ hochwertige Software entstehen. Eine Voraussetzung dafür sind Werkzeuge und Infrastrukturen, die eine automatisierte und kontinuierliche Auslieferung unterstützen. Wie effizient neue Produktinkremente letztendlich ausgeliefert werden können, hängt aber auch von der Architektur eines Systems ab. Ein eng gekoppeltes System kann die Entwicklung und Deployment-Prozesse um ein Vielfaches verlangsamen. Aus diesem Grund sind in den letzten Jahren Microservices immer beliebter geworden. Microservices haben den Vorteil, dass sie unabhängig als kleine Einheiten entwickelt und ausgeliefert werden können. Schaut man sich die aktuelle Literatur zu dem Thema an, findet man Beschreibungen zu Merkmalen und Eigenschaften von Microservices sowie Ansätze zur fachlichen Architektur und Organisationsstruktur in einer Microservice-Umgebung. Auch Infrastrukturthemen wie z.B. Skalierung, Logging und Monitoring werden behandelt und die Vor- und Nachteile verschiedener Ansätze gegeneinander abgewogen. Was gänzlich fehlt, ist eine Darstellung der Informationen, die eine Microservice-Architektur in ihrer Gesamtheit beschreiben. Diese Informationen sind für die Entwicklung allerdings von grundlegender Bedeutung, damit das zu entwickelnde System den Anforderungen der Architektur genügt.

1.2 Ziele der Arbeit

Die vorliegende Arbeit beschäftigt sich mit der Frage, welche Informationen für die Entwicklung einer Microservice-Architektur benötigt werden und wie man diese Informationen sinnvoll darstellen kann. Die Informationen betreffen die fachliche Strukturierung der Services sowie die Infrastruktur für deren Betrieb. Ziel ist es, alle relevanten Informationen zusammenzufassen und diese für die verschiedenen Interessenten der Architektur so aufzubereiten, dass sie davon

profitieren können. Interessenten können zum einen Entwickler sein, die Services auf Mikroebene implementieren und betreiben wollen oder auch Architekten bzw. ebenfalls Entwickler, welche die Architektur auf der Makroebene entwerfen. Auch Projektleiter können von den Informationen profitieren, um bspw. den Aufwand für die Umsetzung besser einschätzen zu können. Weitere Interessenten können außerdem die typischen Interessensvertreter einer Architektur sein, wie Tester, Benutzer, Teamleiter, Designer, Analysten und Auftraggeber.

1.3 Vorgehensweise und Aufbau

Die Arbeit besteht aus einem theoretischen und einem praktischen Teil. Der theoretische Teil beginnt mit den Grundlagen zum Thema Softwarearchitektur. Hier wird unter anderem erläutert wie Softwarearchitekturen definiert sind, woraus sie bestehen und welche Ziele sie verfolgen. Im Anschluss wird die Modellierung von Softwarearchitekturen betrachtet. Nach einer Einführung des Modellbegriffs und den Zielen der Modellierung, wird der Verwendungszweck von Modellen und Modellierungssprachen erläutert und ein Bezug zu den Modellen und Sichten in der Softwarearchitektur hergestellt. Danach werden Microservices eingeführt. Hier werden Ziele und Eigenschaften von Microservices zusammen mit deren fachlicher Architektur und Qualitätsmerkmalen beschrieben. Außerdem werden relevante Architekturstile und Muster wie auch Querschnittsfunktionalitäten, Infrastruktur und Betrieb betrachtet. Die Litertaturrecherche zu den hier relevanten Informationen erfolgt über Fachliteratur wie Wolff (2016b) und Newman (2015) sowie über Artikel aus Fachzeitschriften und wissenschaftlichen Publikationen. Aufgrund der Aktualität des Themas werden auch Internetquellen herangezogen, wobei hierbei aufgrund einer hohen Expertise des Autors „*Microservice Architecture“* (Richardson, 2014a) zu nennen ist. Nach der Einführung wird geprüft welche Ansätze der Modellierung bereits in der Literatur existieren und welche Informationen Unternehmen aus der Praxis bereitstellen. Darauf aufbauend wird im praktischen Teil eine empirische Befragung durchgeführt, um erste Annahmen aus dem theoretischen Teil zu verifizieren. Anschließend wird untersucht, inwieweit sich eine Microservice-Architektur mit dem klassischen arc42-Template beschreiben lässt. Schließlich wird auf Basis der Erkenntnisse eine Evaluation durchgeführt. Anhand einer Befragung, die sich an Unternehmen richtet, welche in Zukunft Microservices einsetzen wollen, soll herausgefunden werden, ob die in einem Modell bereitgestellten Informationen eine Microservice-Architektur ausreichend beschreiben. Am Ende werden die Ergebnisse ausgewertet und interpretiert und die Arbeit schließlich mit einem Ausblick sowie einem Fazit abgeschlossen.

2 GRUNDLAGEN

Dieses Kapitel beschäftigt sich mit den Grundlagen von Softwarearchitekturen und ihrer Modellierung. Hierfür wird zuerst in Abschnitt 2.1 dargestellt, wie eine Softwarearchitektur definiert ist, welche Ziele sie hat und wie diese Ziele erreicht werden können. Im zweiten Teil (Abschnitt 2.2) werden u.a. Ziele und Prinzipien der Modellierung sowie der Einsatz von Modellierungssprachen näher erklärt.

2.1 Softwarearchitektur

Die Softwarearchitektur nimmt in Softwareprojekten eine zentrale Rolle ein und fungiert als Brücke zwischen Analyse und Implementierung (Starke, 2014). Sie entsteht auf Basis der Anforderungen aus der Anforderungsanalyse und gibt wichtigen Input für die zu erstellenden Aufgaben und Arbeitspakete der Projektplanung. Für die Implementierung gibt sie einen Rahmen vor und definiert zentrale Bausteine und Schnittstellen. Architekturen tragen außerdem zum Risikomanagement bei, indem sie Risiken erkennen, bewerten und Lösungsstrategien entwickeln. Weiter hängen die Aufgaben der Architektur eng mit denen des Betriebs, der Hardwarearchitekten und der Qualitätssicherung zusammen. Die Architektur wird außerdem von organisatorischen Faktoren wie dem Entwicklungsprozess und Termindruck beeinflusst und kann auf der anderen Seite die organisatorischen Abläufe mitbestimmen.

2.1.1 Definition

Für den Begriff Softwarearchitektur existieren viele verschiedene Definitionen in der Literatur. Eine häufig anzutreffende Definition lautet:

> *The software architecture of a system is the set of structures needed to reason about the system, which comprise software elements, relations among them, and properties of both. (Bass, Clements, & Kazman, 2013)*

Die Softwarearchitektur wird hier als eine Struktur beschrieben, die benötigt wird, um Aussagen über ein System treffen zu können. Diese Strukturen bestehen aus Softwarebausteinen, den Beziehungen und deren Eigenschaften. Nach Zörner (2012) tauchen in den verschiedenen Definitionen folgende Aspekte immer wieder auf:

- Die Strukturierung des Systems und dessen Zerlegung in Teile, Verantwortlichkeiten sowie das Zusammenspiel dieser Teile und deren Schnittstellen
- Dinge, die im Nachhinein schwer änderbar sind
- Entscheidungen und die Begründung für diese Entscheidungen

Somit kann festgehalten werden, dass es sich bei einer Softwarearchitektur immer um eine im Nachhinein schwer änderbare Struktur handelt, die auf der Summe von einzelnen Entscheidungen basiert. In dieser Struktur wird die Aufteilung von Softwarebausteinen, deren Verantwortlichkeiten und die Beziehung zwischen den einzelnen Bausteinen und deren Schnittstellen beschrieben.

2.1.2 Ziele

Die Softwarearchitektur verfolgt nach Posch, Birken, & Gerdom (2011) vier wesentliche Ziele. Sie will Entwicklungsprojekte effizienter gestalten, Risiken minimieren, indem sie Einflussfaktoren früh berücksichtigt, ein Verständnis zwischen den Beteiligten schaffen und das Kernwissen über das System erhalten. Ein Mittel, um diese Ziele zu erreichen, ist das Treffen von Entscheidungen. Die Entscheidungen können auf verschiedene Weise bspw. über Dokumentation, UML-Modelle oder Quellcode explizit gemacht werden. Ein weiteres Mittel ist die Definition des Systemgerüsts, das die Designentscheidungen festlegt und von den Details abstrahiert. Die frühen Designentscheidungen haben erhebliche Auswirkungen auf das Projekt. Sie definieren Einschränkungen für die Implementierung und beeinflussen die Umsetzung von Qualitätsmerkmalen (Abschnitt 2.1.7) als auch die Organisationsstruktur. Als letztes Mittel dient eine zielgerichtete Dokumentation. Diese orientiert sich an die unterschiedlichen Stakeholder und Aspekte einer Architektur, welche anhand von unterschiedlichen Sichten (Abschnitt 2.1.5) dokumentiert werden.

2.1.3 Systemgedanke

Da sich die Softwarearchitektur mit IT-Systemen beschäftigt, werden nachfolgend die allgemeinen Eigenschaften von Systemen erläutert. Nach Vogel, et al., (2005) ist ein System eine Einheit, welche aus wechselseitig interagierenden Systembausteinen besteht. Ein System kann wiederum ein Systembaustein anderer Systeme sein (*Subsystem*). Es besitzt eine Systemgrenze, die es von seiner Umwelt abgrenzt und existiert, um ein bestimmtes Ziel zu erreichen. Für die Erreichung dieses Ziels kann es entweder mit seiner Umwelt interagieren oder Informationen austauschen. Ein System ist nach der Systemtheorie mehr als die Summe seiner Einzelteile, da es Eigenschaften besitzt, welche aus dem Zusammenspiel seiner Systembausteine entstehen (*Emergenz*). Diese Eigenschaften werden für eine ganzheitliche Betrachtung des Systems herangezogen (*Holismus*). Hierbei werden die Subsysteme als Black Box angesehen und der Fokus auf die Interaktion der Systembausteine gelegt. Im Gegensatz dazu werden beim *Reduktionismus* Systembausteine als White Box getrennt voneinander betrachtet, damit konkrete Aussagen über die Funktionsweise der einzelnen Bausteine getroffen werden können. Nach Vogel, et al., (2005) ist es aufgrund der Emergenz nicht möglich das Verhalten des Gesamtsystems mithilfe des Reduktionismus zu bestimmen. Auf der anderen Seite müssen die einzelnen Systembausteine des Systems bekannt sein, um Aussagen über das Gesamtsystem treffen zu können. Die wechselseitige Betrachtung hilft somit das System und seine Funktionsweise als Ganzes zu verstehen und dadurch seine Komplexität greifbarer zu machen.

2.1.4 Makro- und Mikroarchitektur

Als Makro- und Mikroarchitektur werden zwei Abstraktionsstufen bezeichnet, auf denen ein System entworfen werden kann: dem Grobentwurf und dem Feinentwurf (Vogel, et al., 2005). Diese Abstrahierung wird im Kontext von Architekturebenen verwendet. Architekturebenen dienen dazu, das System in verschiedenen Detailgraden zu betrachten. Es gibt drei grundsätzliche Ebenen:

- **Organisationsebene:** Auf der Organisationsebene werden u.a. Organisationen, Geschäftsprozesse, IT-Landschaften und die Interaktionen mit anderen Organisationen betrachtet. Die hier enthaltenen IT-Systeme werden ohne die innere Struktur dargestellt (Black Box). Auch die Anforderungen einer Organisation sowie organisationsweit verwendete Standards und Richtlinien gehören zu dieser Ebene. Ein Bereich, welcher der Organisationsebene zugeordnet wird, ist die *Enterprise-Architektur.*
- **Systemebene:** Auf der Systemebene werden die IT-Systeme in einem höheren Detailgrad betrachtet. Wesentliche Elemente sind die Anforderungen, Systemkontexte und Subsysteme der Systeme. Der Detailgrad der Systeme beschränkt sich nur auf die Darstellung von möglichen Subsystemen. Die Subsysteme selbst werden als Black Box betrachtet.
- **Bausteinebene:** Die Bausteinebene erhöht den Detailgrad der Subsysteme und legt deren innere Struktur dar (White Box). Somit können auf dieser Ebene schließlich die Verantwortlichkeiten der Systembausteine, deren Schnittstellen und Interaktionen betrachtet werden.

Anforderungen, Entscheidungen und Strukturen der Makroarchitektur sind auf einem hohen Abstraktionsniveau angesiedelt. In den Ebenen findet sie auf der Organisation-, System- und zum Teil auf der Bausteinebene statt. Ein Beispiel für die Makroarchitektur auf Organisationsebene sind organisationsweite Richtlinien. Auf Systemebene zählen Entscheidungen dazu, die sich auf eine Gruppe von anderen Systemen auswirken. Auf Bausteinebene wird zwischen tragenden und nicht-tragenden Bausteinen unterschieden. Tragende Bausteine gehören zur Makroarchitektur, nicht-tragende Bausteine ohne architektonische Relevanz gehören zur Mikroarchitektur.

2.1.5 Architektursichten

Architektursichten helfen dabei Details und Aspekte eines Systems zu abstrahieren. Sie stellen nur die Informationen dar, die für den entsprechenden Sachverhalt notwendig sind. Nach Posch, Birken & Gerdom (2011) gibt es folgende typische Sichten auf eine Softwarearchitektur:

- **Kontextsicht:** Bei einer Kontextsicht wird das System von außen als Black Box betrachtet. Neben der Identifizierung der Systemgrenze werden hier die nach außen sichtbaren Schnittstellen zu Anwendern, Betreibern und Fremdsystemen festgelegt.

- **Bausteinsicht (auch Struktursicht):** Die Bausteinsicht zeigt die Struktur und Zusammenhänge zwischen den Bausteinen. Bei dieser Sicht werden Komponenten, Klassen, Subsysteme, Pakete oder Partitionen sowie deren Abhängigkeiten und Schnittstellen sichtbar gemacht. Die Zerlegung wird je nach Komplexität hierarchisch fortgeführt und kann durch Black und White Boxes beschrieben werden.
- **Laufzeitsicht (auch Ausführungssicht):** Die Laufzeitsicht zeigt die Wechselwirkungen der in der Bausteinsicht beschriebenen Systembausteine zur Laufzeit.
- **Verteilungssicht (auch Infrastruktursicht):** Die Verteilungssicht zeigt die technische Ablaufumgebung des Systems in Form von Hardwarekomponenten und die Verteilung der Softwarebausteine auf dieser Hardware.

Dies sind nur einige Beispiele für die Definition von Sichten. In der Literatur existieren noch viele weitere Vorschläge, wie z.B. das *Zachman-Framework* oder das *4+1-Sichten-Modell*.

2.1.6 Architekturstile

Architekturstile sind Architekturarten, die immer wieder verwendet werden, und dienen als Lösungsschablone für wiederkehrende Probleme (Posch, Birken, & Gerdom, 2011). Sie gelten als Strukturierungsprinzip und beschreiben, in welche Bestandteile eine Softwarearchitektur aufgeteilt wird. Architekturstile werden häufig genutzt, um gewissen Qualitätsanforderungen zu genügen und betreffen im Allgemeinen die gesamte Architektur. Nach Starke (2014) gibt es unterschiedliche Stilarten mit verschiedenen Intentionen. Beispiele sind hierarchische, verteilte, ereignisbasierte oder heterogene Systeme. Im Folgenden wird als Beispiel die Schichtenarchitektur vorgestellt, welche zu den hierarchischen Systemen zählt.

Schichtenarchitektur

Eine Schichtenarchitektur strukturiert das System in mehrere Schichten und kann dabei helfen dessen Komponenten zu entkoppeln (Starke, 2014). Eine Schicht nutzt dabei Dienste von darunterliegenden Schichten und bietet den darüberlegenden Schichten ebenfalls bestimmte Dienste an. Eine umgekehrte Nutzung von unteren Schichten zu höheren sollte vermieden werden, da ansonsten eine stärkere Abhängigkeit zwischen den Schichten entsteht. Komponenten, bei denen dies der Fall ist, sollten nach Starke (2014) in eine gemeinsame Schicht verschoben werden. Eine typische Drei-Schichten-Architektur enthält eine *Präsentationsschicht* für die Darstellung der Daten, eine *Logikschicht* für die Verarbeitung der Anwendungslogik und eine *Datenhaltungsschicht* für das Speichern und Laden der Daten aus einer Datenbank. Ein Nachteil einer klassischen Schichtenarchitektur ist, dass die Anfragen durch alle Schichten hindurchgereicht werden, was zu Lasten der Performance geht (Starke, 2014).

2.1.7 Qualitätsszenarien

Um eine Softwarearchitektur abschließend prüfen und bewerten zu können, werden Qualitätsszenarien herangezogen (Starke & Hruschka, 2016). Diese Szenarien helfen dabei zu

messen, inwieweit sich die *nichtfunktionalen Anforderungen* und *Qualitätsmerkmale* der Architektur mit den Zielvorgaben decken. Die wesentlichen zwei Szenarien sind die *Nutzungsszenarien* und die *Änderungsszenarien*. Nutzungsszenarien beschreiben die Reaktion des Systems auf einen bestimmten Auslöser zur Laufzeit, Änderungsszenarien eine Änderung des Systems oder der Systemumgebung. Für sicherheitskritische Systeme können außerdem *Grenz-* oder *Stressszenarien* herangezogen werden, um zu beschreiben, wie das System mit Extremsituationen umgeht. Für einen Überblick können die Szenarien entlang der Qualitätsmerkmale an einen *Qualitätsbaum* entlang angeordnet werden. Qualitätsmerkmale von Softwaresystemen sind bspw. in der (ISO/IEC 9126-1:2001, 2001) definiert und sollen hier kurz erläutert werden:

- **Funktionalität:** Die Fähigkeit des Softwareprodukts Funktionen anzubieten, welche die angegebenen und implizierten Bedürfnisse des Nutzers erfüllen.
- **Zuverlässigkeit:** Die Fähigkeit des Softwareprodukts ein bestimmtes Maß an Performance zu unterstützen.
- **Benutzbarkeit:** Die Fähigkeit des Softwareprodukts verständlich, benutzbar und erlernbar für den Nutzer zu sein.
- **Effizienz:** Die Fähigkeit des Softwareprodukts eine angemessene Leistung, bezogen auf die Menge der genutzten Ressourcen, bereitzustellen.
- **Wartbarkeit:** Die Fähigkeit des Softwareprodukts geändert werden zu können. Änderungen können Korrekturen, Verbesserungen oder Anpassungen der Software an Änderungen der Umgebung, der Anforderungen oder funktionalen Spezifikationen sein.
- **Übertragbarkeit:** Die Fähigkeit des Softwareprodukts von einer Umgebung in eine andere überführt werden zu können.

2.1.8 Verteilte Systeme

Verteilte Systeme sind Softwaresysteme, deren Bestandteile auf unterschiedlichen und ggf. physikalisch getrennten Systemen ablaufen (Starke, 2014). Bei dem Entwurf dieser Systeme kommt es häufig zu Falschannahmen, den *Fallacies of Distributed Computing*, welche von Rotem-Gal-Oz (2006) auf folgende Weise interpretiert werden:

- **Das Netzwerk ist zuverlässig:** Stromausfälle oder Hardwareprobleme können zu einer Trennung mit dem Netzwerk führen. Nutzt man Dienste von Drittanbietern, liegt die Zuverlässigkeit des Netzwerkes nicht mehr in der eigenen Hand.
- **Die Latenz ist gleich Null:** Eine Anwendung kann in Produktion immer Latenzprobleme aufdecken, auch wenn diese in der Testumgebung und im Staging-Umfeld ausreichend getestet wurde.
- **Unendliche Bandbreite:** Es gibt immer eine Grenze für die Bandbreite. Selbst wenn die Bandbreite sehr hoch ist, muss mit Paketverlusten gerechnet werden. Im

Zusammenspiel mit der Annahme, dass die Latenz nicht gleich Null ist, sollte daher die Größe der über das Netzwerk gesendeten Daten limitiert werden.

- **Das Netzwerk ist sicher:** Im Netzwerk finden ständig Angriffe statt, weshalb beim Entwurf verteilter Systeme auch Sicherheitsaspekte berücksichtigt werden müssen.
- **Die Topologie ändert sich nicht:** Die Topologie eines Netzwerkes ändert sich permanent. Man sollte daher nicht von spezifischen Endpunkten oder Routen abhängig sein.
- **Es gibt einen Administrator:** Speziell in Enterprise-Systemen sollte man nicht davon ausgehen, dass es nur einen Administrator gibt. Wird z.B. mit anderen Anwendungen über das Netzwerk kommuniziert, können andere administrative Regeln in diesen Anwendungen gelten.
- **Die Transportkosten sind Null:** Es gibt zwei Interpretationsarten für diese Aussage. Zum einen verursacht der Weg von der Anwendungsschicht in die Transportschicht zusätzliche Kosten (Ressourcen und Zeit), zum anderen ist das Aufbauen und Betreiben eines Netzwerks mit Kosten verbunden.
- **Das Netzwerk ist homogen:** Netzwerke sind heutzutage heterogen. Wenn man dennoch ein homogenes Netzwerk aufbaut, sollte man beim Entwurf *Interoperabilität* berücksichtigen, indem man bspw. keine proprietären Protokolle für die Kommunikation verwendet.

Das CAP-Theorem

Das CAP-Theorem besagt, dass bei einem verteilten Datenbanksystem die Eigenschaften *Konsistenz (consistency)*, *Verfügbarkeit (availability)*, *Partitionstoleranz (partition tolerance)* nicht gemeinsam erreicht werden können (Gilbert & Lynch, 2002):

- **Konsistenz** bedeutet, dass jede Leseoperation, die nach einem Schreibvorgang stattfindet, den Wert dieses Schreibvorgangs zurückliefern muss.
- **Verfügbarkeit** wird gewährleistet, wenn jede Anfrage, die von einem nicht fehlerhaften Knoten entgegengenommen wird, auch beantwortet wird.
- **Partitionstoleranz** (auch Ausfalltoleranz) ist gegeben, wenn das System auch bei dem Verlust von Nachrichten weiterhin fehlerfrei arbeitet.

Bei der Aktualisierung von zwei Knoten kann die Latenzzeit als temporäre Partitionierung betrachtet werden (Messinger, 2013). Dann muss entschieden werden, ob ein Lesezugriff zwischen zwei Knoten auch vor der Aktualisierung erlaubt ist und das System damit verfügbar bleibt oder ob die Knoten zugunsten der Konsistenz gesperrt werden. Oft wird zugunsten der Verfügbarkeit entschieden, damit Endnutzer rechtzeitig eine Antwort erhalten (Trelle, 2011). Hier setzen nach Trelle (2011) viele *NoSQL*-Datenbanken auf das *BASE*-Prinzip (*Basically Available, Soft State, Eventual consistency*). Das System wird dabei nach einer bestimmten Zeitspanne der Inkonsistenz wieder in einen konsistenten Zustand überführt. Ein Beispiel für ein AP-System mit Eventual consistency ist *Cassandra* (Abschnitt 5.1).

2.2 Modellierung

Modelle sind ein zentrales Werkzeug bei der Erstellung eines Architekturentwurfs. Sie helfen dabei die Komplexität einer Softwarearchitektur beherrschbar zu machen. In diesem Kapitel wird der Modellbegriff eingeführt und anschließend erklärt, welche Ziele und Prinzipien die Modellierung verfolgt. Im Anschluss daran werden Modellierungssprachen vorgestellt und anhand eines Beispiels gezeigt, wie eine Architektursicht visuell dargestellt werden kann.

2.2.1 Modellbegriff

Eine allgemeingültige Definition für den Modellbegriff stammt von Stachowiak (1973). Er beschreibt folgende Eigenschaften eines Modells:

- Ein Modell bezeichnet immer eine Abbildung, eine Repräsentation natürlicher Originale, die selbst Modelle sein können.
- Ein Modell erfasst nicht alle Attribute des Originals, sondern nur verkürzt die dem Modellnutzer oder Ersteller relevanten Informationen.
- Ein Modell orientiert sich am Nützlichen (*Pragmatismus*). Es ist einem Original nicht von sich aus zugeordnet, sondern wird vom Modellnutzer oder Ersteller innerhalb einer bestimmten Zeitspanne zu einem bestimmten Zweck für das Original eingesetzt und somit interpretiert.

Zusammengefasst ist ein Modell eine Abstraktion der Realität, welches alle relevanten Informationen des Originals beinhaltet und von den an dem Modell beteiligten Personen interpretiert wird. Die Tätigkeit, die ein System schließlich in ein Modell überführt, wird als Modellierung bezeichnet. Hierbei werden alle relevanten Objekte, Akteure, Zusammenhänge und Abläufe unter Berücksichtigung der Systemgrenzen festgehalten (Bernroider & Stix, 2006).

2.2.2 Ziele und Prinzipien

Das Ziel einer Modellierung ist es, mit Hilfe der verkürzten Informationen komplexe Aufgaben zu lösen. Man analysiert zuerst das System, in dem sich die zu lösende Aufgabe befindet, und dokumentiert die gefundenen Erkenntnisse. Das aus den Erkenntnissen entstandene Modell dient anschließend als Kommunikationsmittel aller beteiligten Personen. Für die Kommunikation ist es wichtig, sich auf einen Modelltypen zu einigen und zu spezifizieren, was einzelne Bestandteile des Modells in der Realität repräsentieren (Bernroider & Stix, 2006). Bei der Überführung eines Originals in ein Modell unterstützen die Prinzipien Abstraktion, Partitionierung und Projektion (Bernroider & Stix, 2006):

- **Abstraktion:** Es werden nur die relevanten Teile betrachtet, die zur Lösung des Problems beitragen. Ein Beispiel hierfür ist das einfache Weglassen von Teilen. Dabei muss darauf geachtet werden, dass man alle für das Modell relevanten Informationen beibehält. Damit diese Betrachtung nicht zu subjektiv wird, schlagen Bernroider & Stix (2006) vor, die Abstraktion mit der Projektion zu kombinieren. Eine weitere Möglichkeit

ist die Zusammenfassung von Objekten in Klassen. Einzelne Systembausteine wie z.B. eine Grafik-, Netzwerkkarte oder Eingabegeräte könnten zu einer Klasse „Rechner“ zusammengefasst werden.

- **Partitionierung:** Bei der Partitionierung werden die Teilsysteme und die Kommunikation der Teilsysteme bzw. die Schnittstellen betrachtet. Hier wird ebenfalls ein Zusammenspiel mit der Abstraktion vorgeschlagen. Nach einer Abstraktion des Systems kann im zweiten Schritt partitioniert und im dritten die Partition konkretisiert werden. So ergeben sich am Ende mehrere Abstraktionsebenen eines Systems mit unterschiedlichen Detailgraden.
- **Projektion:** Für ein vollständiges Modell müssen verschiedene Sichtweisen auf das System eingenommen werden (Abschnitt 2.1.5). Diese Sichtweisen entsprechen den Projektionen. Die Projektion beugt somit Missverständnissen oder unvollständiger Modellierung relevanter Informationen vor.

2.2.3 Einsatz in der Softwarearchitektur

Modelle können nach Starke (2014) in der Softwarearchitektur in verschiedenen Situationen Verwendung finden. Beispielsweise lassen sich damit statische oder dynamische Codestrukturen modellieren, um die Komplexität großer Mengen Code zu reduzieren. Damit Entwickler ausschließlich die relevanten Teile für die Umsetzung erhalten, werden außerdem fachliche Situationen modelliert. Dadurch kann die inhaltliche Komplexität für die Umsetzung entschärft werden. Personen mit weniger technischem Vorwissen können ebenfalls von Modellen profitieren, da diese ihnen ein einfaches Verständnis über das System vermitteln. Ein weiteres großes Einsatzgebiet von Modellen ist die Generierung von Source Code und anderen Artefakten. In diesem Zusammenhang wird von *modellgetriebener Architektur (MDA)* gesprochen. Diese basiert darauf, ein Softwaresystem mittels verschiedener Abstraktionsstufen anhand von formalen Modellen zu beschreiben und automatisch mittels formal definierter Transformationen ineinander zu überführen (Reussner & Hasselbring, 2009).

2.2.4 Modellierungssprachen

Um schließlich ein System modellieren zu können, wird eine Modellierungssprache benötigt. Modellierungssprachen können textuell oder visuell dargestellt werden. Ein Modell, welches eine Modellierungssprache definiert, wird auch als Metamodell bezeichnet (Reussner & Hasselbring, 2009). Es besteht aus einer Semantik und einer Syntax, welche sich in eine konkrete und eine abstrakte Syntax unterteilt. Die konkrete Syntax beschreibt die Notation und daher wie etwas dargestellt wird; die abstrakte Syntax stellt die Menge der definierten Vokabeln und deren strukturellen Zusammenhang dar. Die Semantik beschreibt letztendlich die Bedeutung und Verwendung der Modellelemente (Weilkiens, Weiss, Grass, & Duggen, 2015). Zu den bekanntesten Modellierungssprachen gehört die *Unified Modeling Language (UML)*. Sie beinhaltet Struktur- und Verhaltenselemente, die anhand verschiedener Diagrammtypen visualisiert werden können. Strukturelemente beschreiben die statische Struktur eines Systems

und Verhaltenselemente dessen dynamische Funktionen. So können z.B. in der UML Strukturelemente als *Klassen*-, *Komponenten*-, oder *Verteilungsdiagramme* und Verhaltenselemente als *Aktivitäts*-, *Zustands*- oder *Sequenzdiagramme* dargestellt werden (Weilkiens, 2008). Für die UML gibt es ebenfalls ein entsprechendes Metamodell, welches die Modellierungssprache definiert - das *UML-Metamodell* (Reussner & Hasselbring, 2009). Das UML-Metamodell wird wiederum durch ein Meta-Metamodell definiert - der *Meta Object Facility (MOF)*. Becker, Probandt & Vering (2012) unterteilen Modellierungssprachen in Sprachen für Daten und Sprachen für Prozesse. Typische Sprachen für die Modellierung von Daten sind *Entity Relationship-Diagramme* oder die UML. Prozesse können mit *ereignisgesteuerten Prozessketten*, *Petrinetzen* oder der *Business Process Modeling Notation* modelliert werden.

2.2.5 Architekturmodelle und Sichten

Architekturmodelle sind textuelle oder grafische Metabeschreibungen, die es erst möglich machen, die Komplexität einer Softwarearchitektur zu erfassen (Posch, Birken, & Gerdom, 2011). Damit ein Architekturmodell dabei keine relevanten Teile auslässt, kommt das in Abschnitt 2.2.2 beschriebene Prinzip der Projektion zum Tragen. Für alle Aspekte, die in einem System berücksichtigt werden müssen, werden Sichten erstellt (Abschnitt 2.1.5). Die Modellierungssprachen helfen bei der Darstellung dieser Sichten. Dabei gilt, dass die Sichten bzw. die dargestellten Informationen für sich stehend immer unvollständig sind, da sich die vollständige Beschreibung des Systems ausschließlich im Architekturmodell befindet (Weilkiens, 2008). Abbildung 2-1 zeigt ein Beispiel für die Modellierung einer Bausteinsicht. Das System wird dort ausgehend vom Systemkontext hierarchisch zergliedert (Zörner, 2012). Hierzu wird auf der obersten Ebene das System erst wie in der Kontextsicht als Black Box beschrieben und immer detaillierter auf die weiteren Ebenen heruntergebrochen.

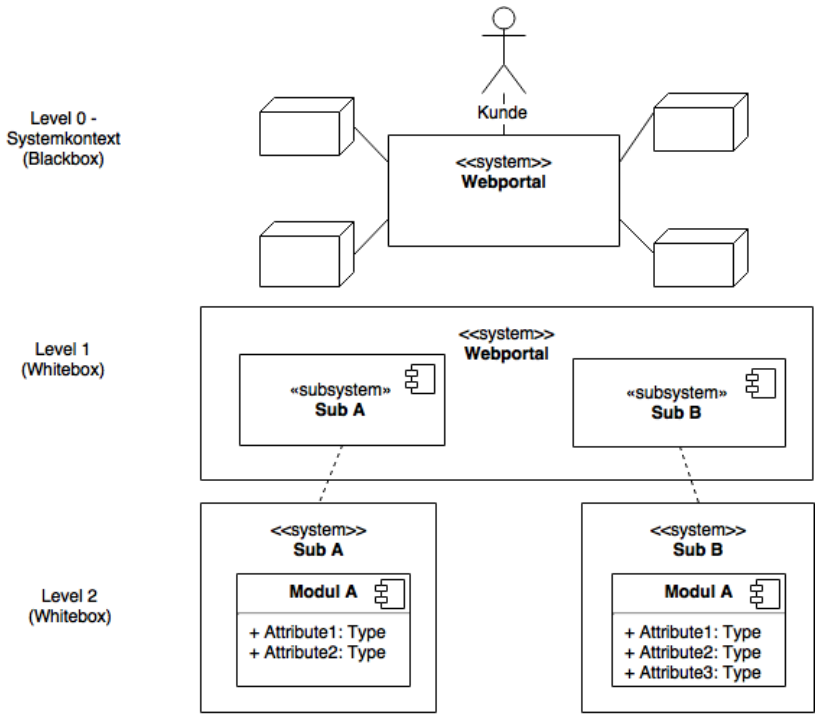

Abbildung 2-1: Bausteinsicht, angelehnt an Starke (2014)

3 MICROSERVICES

Dieses Kapitel beschreibt die Ziele und Eigenschaften von Microservices und wie diese fachlich strukturiert werden können. Außerdem werden Architekturstile und Muster vorgestellt, die in einer Microservice-Architektur Verwendung finden. Abschließend werden die Querschnittsfunktionalitäten betrachtet, die beim Entwurf dieser Architektur berücksichtigt werden müssen, und dargelegt, welche Werkzeuge und Muster für die Infrastruktur und den Betrieb von Microservices benötigt werden.

3.1 Ziele und Eigenschaften

Microservices dienen zur Aufteilung der Software in Module (Wolff, 2016b). Im Gegensatz zu anderen Architekturmustern, welche ebenfalls eine Modularisierung anstreben, sind Microservices in sich abgeschlossene Programme, die als eigenständige Prozesse laufen. Dieses Vorgehen ist an der Unix-Philosophie angelehnt. Programme erledigen nur eine Aufgabe, interagieren mit anderen Programmen und nutzen hierfür eine gemeinsame Schnittstelle. Mit Microservice-Architekturen will man Problemen entgegenwirken, die üblicherweise bei der Entwicklung und dem Betrieb von monolithischen Architekturen auftreten. Deployment-Monolithen haben z.B. nach Wolff (2016b) und Fowler & Lewis (2014) den Nachteil, dass sie nur zusammenhängend auf die Systeme verteilt werden können, was sehr zeitintensiv und kostspielig sein kann. Es spielt dabei keine Rolle, ob der Monolith selbst modular aufgebaut ist, da er bei jeder Änderung eines Moduls als Gesamtpaket den Deployment-Prozess durchlaufen muss. Das Deployment eines Monolithen ist zudem risikoreicher, da auch Module Fehler verursachen können, auf denen keine Änderung stattgefunden hat (Richardson, 2014c). Fowler & Lewis (2014) halten hierzu fest, dass je weiter die Entwicklung in einem Monolithen voranschreitet, desto schwieriger wird es, die modulare Struktur am Leben zu halten. Diese steigende Erosion geht außerdem zu Lasten der Produktivität der Entwickler. Microservice-Architekturen versuchen die Produktivität langfristig beizubehalten, indem sie sich auf Ersetzbarkeit fokussieren (Nadareishvili, Mitra , McLarty , & Amundsen , 2016). Sie können durch folgende Eigenschaften beschrieben werden:

Klein

Microservices sind klein und auf eine bestimmte Aufgabe spezialisiert. Sie folgen damit dem Prinzip der hohen Kohäsion (Newman, 2015). Eine Möglichkeit die Größe eines Microservice festzulegen, ist die Anzahl an Codezeilen zu begrenzen. Newman (2015) und Wolff (2016b) weisen allerdings darauf hin, dass die Komplexität eines Service von der verwendeten Programmiersprache abhängt und deshalb die *Lines of Code (LoC)* kein eindeutiger Indikator sind. Wolff (2016b) ergänzt weiter, dass sich die Architektur nach den „*Gegebenheiten in der fachlichen Domäne“* richten soll anstatt nach der Anzahl der Codezeilen. Beide empfehlen den

Service auf die Struktur des Entwicklerteams auszurichten. Ein Service sollte nur so groß sein, dass er von einem Team entwickelt und betreut werden kann. Eine weitere Möglichkeit die Größe festzulegen, ist die Kommunikation zwischen den Services zu betrachten (Wolff, 2016b). Findet zu viel Kommunikation zwischen den Services statt, wirkt sich das negativ auf die Performance aus und die Services müssen ggf. größer geschnitten werden. Hassan & Bahsoon (2016) stellen hierzu fest, dass eine aggressive Isolation der Services entlang der Fachlichkeit nicht immer ideal für die Erfüllung nicht-funktionaler Anforderungen ist. Dennoch empfehlen Newman (2015) und Wolff (2016b), die Services so klein wie möglich zu halten. Eine Voraussetzung hierfür ist nach Newman (2015), dass man die ansteigende strukturelle Komplexität bewältigen kann, die mit vielen kleinen Services einhergeht.

Unabhängig

Microservices können unabhängig von anderen Services deployed werden. Sie laufen als eigenständige Prozesse oder virtuelle Maschinen, besitzen eine eigene Datenbank (*Dezentral Data Management*) oder ein abgetrenntes Datenbankschema einer gemeinsamen Datenbank (Wolff, 2016b). Schmale Schnittstellen sollen dabei helfen eine lose Koppelung zwischen den Services zu erreichen. Für die fachliche Unabhängigkeit werden Services nach *Business Capabilities* (Abschnitt 4.1) strukturiert (Fowler & Lewis, 2014). In einer Microservice-Architektur können Entwurfsentscheidungen auf drei Ebenen getroffen werden - der Makro-, der Mikro- und der *Domänenarchitektur* (Preissler & Tigges, 2015). Auf der Ebene der Mikroarchitektur können Teams ihre eigenen Technologieentscheidungen treffen. Fowler & Lewis (2014) sprechen hier von *dezentraler Governance*. Das hat den Vorteil, dass zu jedem Service die am besten geeignete Technologie eingesetzt werden kann. Die Auswirkung einer falschen Technologieentscheidung ist auf Mikroarchitekturebene gering, da sich diese nicht auf die gesamte Architektur auswirkt (Abschnitt 2.1.4). Um eine möglichst hohe Unabhängigkeit zu erreichen, sollten daher nach Wolff (2016b) die meisten Entscheidungen auf der Microebene getroffen werden.

Verteilt

Microservices kommunizieren verteilt über das Netzwerk. Die Kommunikation sollte aus Performancegründen so einfach wie möglich gehalten werden. Fowler & Lewis (2014) nennen diesen Ansatz *Smart Endpoints and dumb Pipes.* Die Intelligenz liegt dabei bei den Endpunkten (bzw. Services) und die Kommunikation erfolgt über einen einfachen Mechanismus zum Nachrichtenaustausch wie *REST (Representational State Transfer)*. Durch die einheitliche Schnittstelle von REST lässt sich die Kommunikation erheblich minimieren, was eine lose Koppelung des Systems begünstigt (Tilkov, 2011). Über das Netzwerk sollten außerdem nur wenige grobgranulare Aufrufe erfolgen, da eine *Interprozesskommunikation (IPC)* im Gegensatz zu Aufrufen auf demselben Prozess vergleichbar langsam ist (Fowler M. , 2014a). Auch auf die Stabilität der Schnittstellen sollte geachtet werden, damit eine Änderung der Schnittstelle nicht auch Änderungen weiterer Services nach sich zieht (Röwekamp & Limburg, 2016). Für das potentielle Scheitern der Aufrufe über das Netzwerk wird außerdem eine Fehlerbehandlung benötigt, die entsprechend reagiert, wenn die Services nicht erreichbar sind (Wolff, 2016b).

3.2 Fachliche Architektur

Das Aufteilen der Architektur wird oft entlang der Kommunikationsstrukturen einer Organisation vorgenommen (Wolff, 2016b). Dieses Vorgehen ist als *Conways Law* bekannt und wurde von Melvin Edward Conway beschrieben:

> *Any organization that designs a system (defined broadly) will produce a design whose structure is a copy of the organization's communication structure.*
> *(Conway, 1968)*

Wenn eine große Applikation in Teile zerlegt wird, fokussiert sich das Management oft auf die technologischen Schichten eines Systems (Fowler & Lewis, 2014). Dies führt zu getrennten Teams, die nur für die Schichten zuständig sind, auf die sie spezialisiert sind. So gibt es bspw. User Interface-Teams, Geschäftslogikteams und Datenbankteams. Microservices dagegen werden nach Fachlichkeit getrennt. Dadurch bekommen Teams die Möglichkeit funktionsübergreifend an einem oder mehreren Services zu arbeiten. Änderungen an der Fachlichkeit werden im Idealfall auch nur von einem Team vorgenommen. Dieses Vorgehen minimiert den Kommunikations- und Koordinationsaufwand und steigert die Produktivität der Teams. Fowler & Lewis (2014) sehen hier einen weiteren Vorteil. Durch die Fokussierung auf *Produkte statt Projekte* sind Teams für den gesamten Lebenszyklus eines Service verantwortlich, wodurch die Beziehung zwischen Nutzern und Entwicklern verstärkt wird. Stimmt die fachliche Aufteilung der Services nicht, können sich daraus einige Nachteile ergeben (Wolff, 2016b). Muss bspw. Funktionalität nachträglich in einen anderen Service verschoben werden, kann es passieren, dass diese aufgrund von unterschiedlichen Programmiersprachen neu implementiert werden muss.

Domain-driven Design

Domain-driven Design (DDD) ist ein Ansatz, um die Grenzen eines Subsystems im Kontext eines großen Systems zu bestimmen (Nadareishvili, et al., 2016). Der Ansatz bietet eine modellzentrierte Sicht auf das System. Hierfür werden komplexe Problemdomänen betrachtet. Eine Problemdomäne oder auch Anwendungsdomäne bezieht sich auf den Fachbereich, für den eine Software entwickelt wird (Millet & Tune, 2015). Durch die genaue Betrachtung der Domäne soll sichergestellt werden, dass Software entwickelt wird, die einen bestimmten Nutzen hat. Die Konzepte des Domain-driven Designs helfen bei der fachlichen Aufteilung von Microservices (Wolff, 2016b) und werden nachfolgend zusammengefasst:

- **Ubiquitous Language:** Eine einheitliche Sprache für ein gemeinsames Verständnis zwischen Entwicklern und Domänenexperten (Evans, 2003). Ohne diese einheitliche Sprache können bei der Übersetzung von Modellierungskonzepten Missverständnisse entstehen. Refactoring-Maßnahmen oder unzuverlässige Software sind die Folgen.
- **Building Blocks:** Grundlegende Muster, um ein Domänenmodell zu entwerfen, sind Entitäten, Wertobjekte, Aggregate, Services, Repositories und Fabriken. Aggregate stellen zusammengesetzte Domänenobjekte dar und werden als eine Einheit geladen und gespeichert. Parallele Änderungen innerhalb von Aggregaten könnten die

Konsistenz gefährden und müssen daher serialisiert und koordiniert werden. Aus diesem Grund sollten Aggregate in einer Microservice-Architektur nicht auf mehrere Services aufgeteilt werden (Wolff, 2016b).

- **Bounded Context:** Da es aufwendig ist, ein einheitliches Modell über die gesamte Domäne hinweg konsistent zu halten, werden mehrere Modelle für verschiedene Teile des Systems benötigt (Millet & Tune, 2015). Hierbei muss berücksichtigt werden, welche dieser Modelle mehrdeutig sind und in welcher Beziehung sie zueinanderstehen. Das *Bounded Context Pattern* hilft dabei die Anwendbarkeit von bestimmten Modellen zu beschränken und bietet ein klares und einheitliches Verständnis für die innere Konsistenz und die Beziehungen zu anderen Kontexten.
- **Context Map:** Die *Context Map* stellt eine Übersicht der verschiedenen Bounded Contexts dar und wie diese zusammenhängen. Die Abhängigkeiten werden durch Interaktionsarten beschrieben (*Context Relationships*). Es gibt Abhängigkeitsformen mit einer engen Koppelung wie *Shared Kernel*, *Customer/Supplier* und *Conformist* und lose gekoppelte Abhängigkeitsformen wie *Open/Host Service*, *Separate Ways*, *Published Language* und der *Anticorruption Layer* (Evans, 2003).

Eine Möglichkeit Domain-driven Design anzuwenden, ist das *Event Storming* (Lowe, 2015). Beim Event Storming wird in einer Gruppe das gesamte Domänenmodell betrachtet. Anhand von Domänenevents wird durch das Modell navigiert. Die Gruppe fügt Kommandos oder Auslöser der Events hinzu, bspw. User-Interaktionen, externe Systeme oder zeitliche Auslöser. Anschließend werden Aggregate identifiziert, die Kommandos entgegennehmen und Ereignisse ausführen, und in Bounded Contexts gruppiert. Daneben werden wichtige Testszenarien, Nutzer sowie Ziele identifiziert und schließlich die Beziehungen zwischen den Bounded Contexts in einer Context Map dargestellt.

3.3 Qualitätsmerkmale

Nachfolgend werden einige der in Abschnitt 2.1.7 vorgestellten Qualitätsmerkmale in Bezug auf Microservices vorgestellt. Zusätzlich wird die Evolvierbarkeit einer Microservice-Architektur betrachtet.

Übertragbarkeit

Wie in Abschnitt 3.1 beschrieben, sind Microservices unabhängig und kommunizieren über schmale Schnittstellen und einfache Mechanismen miteinander. Sie können leicht ausgetauscht werden und bieten daher eine hohe Übertragbarkeit.

Zuverlässigkeit

Da Microservices über das Netzwerk kommunizieren, haben diese ein hohes Ausfallrisiko. Es muss festgelegt werden, wie man in einer Microservice-Architektur auf Ausfälle reagiert. Wolff (2016b) schlägt bspw. vor, eine Weiterleitung auf einen reduzierten Service einzurichten oder beim Ausfall eines angeforderten Service Standardwerte aufzunehmen. Wird auf Ausfälle

entsprechend reagiert, sind laut Newman (2015) und Wolff (2016b) Microservices robuster als Monolithen. Der Grund ist, dass Deployment-Monolithen als Ganzes deployed werden, wodurch Änderungen einzelner Module schneller zu einem Ausfall des Gesamtsystems führen. Newman (2015) fügt hinzu, dass es bei einem Totalausfall von Services möglich ist, die Funktionalität nur schrittweise abzubauen. Sollte ein Microservice zu viel Speicher allokieren oder die CPU zu sehr auslasten, beeinflusst dies keine weiteren Microservices, was ebenfalls zu einem robusten System führt (Wolff, Schwartz, & Heusingfeld, 2016).

Wartbarkeit

Dank der losen Koppelung und hochgradigen Geschlossenheit bieten Microservices eine hohe Wartbarkeit. Sie sind einfach erweiterbar und können als eigene Einheiten getestet werden. Im Gegenzug dazu sind Integrationstests bei vielen Services mit vielen Verbindungen sehr aufwendig durchzuführen (Dragoni, Giallorenzo, Lluch Lafuente, Mazzara, & Montesi, 2016). Bei der Änderung der Organisationsstruktur hingehend zu Microservices können sich weitere Vorteile ergeben. Da Services den gesamten Lebenszyklus lang von funktionsübergreifenden Teams betreut werden sollen, kann eine bessere Softwarequalität entstehen, welche die Wartbarkeit ebenfalls erhöht.

Effizienz

Klassische Softwarearchitekturen verwenden mehr Aufrufe in Prozessen anstatt über das Netzwerk zu kommunizieren, was ihnen bei der Performance Vorteile gegenüber verteilten Architekturen verschafft. Dennoch können auch Microservices effizient sein (Dragoni, et. al.). Services mit einem gut abgegrenzten Kontext, die nur wenige Nachrichten über einfache Protokolle versenden, bilden die Grundlage für ein lose gekoppeltes und effizient arbeitendes System. In Bezug auf Skalierbarkeit haben Microservices außerdem den Vorteil, dass sie einzeln skaliert werden können (Fowler & Lewis, 2014). So werden nur die Systemteile auf mehrere Instanzen verteilt, für die eine Skalierung auch wirklich notwendig ist, was ebenfalls der Effizienz des Systems zugutekommt.

Evolvierbarkeit

Eine hohe Evolvierbarkeit einer Software beschreibt den Zustand langfristig schnelle und einfache Änderungen zu erlauben (Riebisch & Bode, 2009). Weist ein System eine hohe Wartbarkeit auf, führt dies nicht gleichzeitig zu einer hohen Evolvierbarkeit, da die Wartbarkeit nach Riebisch & Bode (2009) den Fokus auf aktuellen Aufwand für Änderungen in den Vordergrund stellt. In Microservice-Architekturen ist die Evolvierbarkeit durch andere technische und architektonische Ausprägungen als bei Monolithen gefährdet (Franz, 2017). Anstelle eines Anstiegs von zyklischen Abhängigkeiten, redundanten Code und steigenden modulübergreifenden Aufrufen, entstehen in einer Microservice-Architektur sich funktional überschneidende Microservices und Services, die viel Verständnis anderer Microservices erfordern oder aufgrund hoher Abhängigkeit viel miteinander kommunizieren müssen. Nach Franz (2017) erzwingt die strikte Trennung der Services, das Erarbeiten von konzeptuellen Modellen, wodurch die Verletzung der Modelle transparenter und aufwendiger sein kann als in konventionellen Architekturen. Als Unterstützung, um eine Microservice-Architektur langfristig

stabil zu halten, helfen laut Fildebrandt (2016) auch altbewährte Entwurfsprinzipien für Objekte wie *SOLID*.

3.4 Architekturstile und Muster

In einer Microservice-Architektur haben sich einige Architekturstile und Muster etabliert, die im Folgenden vorgestellt werden.

Hexagonale Architektur

Ein häufig erwähnter Architekturstil für Microservices ist die hexagonale Architektur (Newman, 2015; Wolff, 2016b). Diese besteht aus einer zentralen Logik und Adaptern, die über Schnittstellen eine Kommunikation nach außen ermöglichen. Die Adapter können als Schichten angesehen werden, die allerdings nicht auf eine Präsentations- und Datenhaltungsschicht beschränkt sind. Die Logik kann über eine REST-Schnittstelle anderen Microservices oder auch Benutzern einer Web-Benutzerschnittstelle angeboten werden. Für alle Schnittstellen können eigene Testimplementierungen umgesetzt werden. Dies vereinfacht das Testen eines Service und ermöglicht eine unabhängige Implementierung sowie ein unabhängiges Deployment.

Schichtenarchitektur

Die in Abschnitt 2.1.6 vorgestellte Schichtenarchitektur findet ebenfalls bei Microservices Verwendung. Verfolgt man bspw. den Ansatz *Monoliths First*, startet man mit einer konventionellen Schichtenarchitektur und überführt das System anschließend schrittweise in eine Microservice-Architektur (Nadareishvili, et al., 2016). Diesem Ansatz liegt die Idee zugrunde, dass ein funktionierendes komplexes System nur aus einem funktionierenden einfachen System entspringen kann. Eine Möglichkeit wäre die Präsentationsschicht und die Datenhaltungsschicht aus einer klassischen Schichtenarchitektur herauszulösen und zu zentralisieren (Kansy, Lubkowitz, & Schäfer, 2016). Anschließend kann die Logikschicht fachlich in einzelne Services aufgeteilt werden. Aufgrund der gemeinsamen Verwendung von Benutzerschnittstelle und Datenbank, ist bei dieser Variante allerdings eine hohe Koordination zwischen den Services notwendig, was zu Lasten der Unabhängigkeit geht. Nach Fowler (2015) können sich auch einige Vorteile durch den Monoliths First-Ansatz ergeben. Zum einen kann aufgrund der geringeren Komplexität schneller evaluiert werden, ob die Software einen sinnvollen Nutzen bringt und zum anderen müssen die Kontextgrenzen nicht bereits am Anfang festgelegt werden.

Self-Contained Systems

Self-Contained Systems (SCS) oder auch *Vertikale* ist ein Architekturansatz, der eine Unterteilung von großen Systemen in mehrere kleine Systeme anstrebt (innoQ, 2015). SCS besitzen eine eigene Datenhaltung, Logik und Benutzerschnittstelle und können optional eine Service API anbieten. Damit die SCS unabhängig, lose gekoppelt und fehlertolerant bleiben, erfolgt die Kommunikation zu anderen SCS asynchron. Verglichen mit den Schlüsselmerkmalen von Microservices weisen SCS einige Unterschiede auf. Microservices sind in der Regel kleiner als SCS und aufgrund ihrer beschränkten Größe die besser anpassbaren Bausteine, während

SCS mehr Verantwortung kapseln und das System somit beherrschbarer machen (Ghadir, 2016). Auch kommunizieren SCS im Gegensatz zu Microservices idealerweise nicht miteinander. Microservices müssen zudem keine eigene Benutzerschnittstelle bereitstellen und können auf verschiedene Weise integriert werden, während SCS vorzugsweise auf der UI-Ebene integriert werden. Es ist möglich, SCS so zu teilen, dass sie aus mehreren Microservices bestehen. In diesem Fall können SCS als spezieller Ansatz einer Mikroarchitektur angesehen werden.

Shared Nothing

Das *Shared Nothing-Prinzip*, auf Microservices angewandt, bedeutet, dass sich die Services keinen gemeinsamen Zustand teilen (Wolff, 2016b). Es gibt z.B. keinen gemeinsamen Code, keine HTTP-Sessions oder geteilte Datenbanken (Steinacker, 2015). Die Redundanz, die dabei entstehen kann, wird zugunsten einer losen Koppelung, unabhängigem Deployment und Skalierung in Kauf genommen.

Event-driven architecture

Erstreckt sich ein fachlicher Prozess über mehrere Microservices, kann mithilfe von *Event-driven architecture* auch über Servicegrenzen hinweg Konsistenz erreicht werden (Richardson, 2014g). Eventbasierte Aufrufe fördern eine lose Koppelung, da der auslösende Service nicht alle weiteren Services in dem fachlichen Prozess kennen muss (Wolff, 2016b). Die Services registrieren sich hierzu beim Auslöser (*Event Emitter)*, welcher wiederum ein Ereignis an die interessierten Services (*Event Consumer)* schickt. Für die Verwaltung der Zustände kann zusätzlich das *Event Sourcing* verwendet werden. Dieses sichert alle Events als eine Ereigniskette in einem Ereignisspeicher (*Event Store*), damit die Zustände, bspw. beim Ausfall eines Service, wieder rekonstruiert werden können. Da das Abfragen des Event Stores komplex und ineffizient sein kann, wird hierfür oft eine *Command Query Responsibility Segregation* (CQRS) verwendet, welche die Befehle für die Zustandsänderungen (Commands) von den Abfragen (Queries) trennt (Richardson, 2014b).

3.5 Querschnittsfunktionalitäten

In diesem Abschnitt werden die Querschnittsfunktionalitäten (auch *Cross Cutting Concerns*) einer Microservice-Architektur vorgestellt. Querschnittsfunktionalitäten sind serviceübergreifende Belange und werden in der Regel der Makroarchitektur zugeordnet. Eine eindeutige Zuordnung ist allerdings nicht festgelegt, so dass auch Lösungen auf der Mikroebene möglich sind (Wolff, 2016b).

Monitoring und Logging

Die Herausforderung beim Monitoring und Logging von Microservices ist es, die Metriken oder Logfiles aller Services an einer zentralen Stelle zusammenzuführen. Metriken für das Monitoring können Metainformationen, Verfügbarkeiten sowie detaillierte Informationen zum Ausfall von Services sein (Wolff, 2016b). Die Informationen können dabei je nach fachlichem Prozess physisch auf einem oder mehreren Microservices liegen (Fichtner, 2016). Vor der

technischen Umsetzung müssen zunächst die relevanten Kennzahlen und deren Zielgruppe festgelegt werden. Fichtner (2016) unterteilt diese Kennzahlen in *System*-, *Anwendungs*- und *Businessmetriken*. Für die Überwachung eines Microservices gibt es nach Newman (2015) je nach Verteilung der Serviceinstanzen mehrere Möglichkeiten:

- **Ein Service pro Host:** Bei einem Service pro Host wird zunächst die Auslastung der Hosts überwacht. Anschließend wird der Zugriff auf die Protokolldateien des Servers eingerichtet und zuletzt die Anwendung selbst überwacht.
- **Ein Service mehrere Hosts:** Bei einem Service, der auf mehreren Hosts repliziert wird, muss festgestellt werden, auf welchem Host das Problem aufgetreten ist und ob das Problem bei den Hosts oder dem Service adressiert werden kann. Mithilfe eines *SSH-Multiplexer* kann man sich auf mehreren Hosts anmelden und die Protokolle der verschiedenen Hosts auswerten.
- **Mehrere Services mehrere Hosts:** Arbeiten mehrere Services zusammen und laufen auf unterschiedlichen Hosts, müssen für eine Fehlererkennung alle Informationen darüber an einer zentralen Stelle zusammengefasst werden. Ein Tool, welches die Kennzahlen zusammenführt, ist *Graphite*. In Kombination mit *Grafana* können die Daten komfortabel innerhalb eines Dashboards angezeigt werden.

Wählt man den Ansatz eines serverlosen Deployments (Abschnitt 3.6) können Dienste wie *AWS Lambda* die Funktionen bereits mit einem Monitoring versehen (Wolff, 2016c). Für das zentrale Logging von Microservices können die Log-Daten direkt über das Netzwerk mitgesendet und mit diversen Tools geparst, gespeichert und auf einer Weboberfläche ausgewertet werden (Wolff, 2016b). Ein Beispiel für eine solche Toolkette ist der *ELK-Stack* bestehend aus *Logstash*, *Elasticsearch* und *Kibana*. Wird ein fachlicher Prozess auf mehrere Services abgebildet, kann eine Korrelations-ID verwendet werden, welche zwischen den Services durchgereicht wird (Fichtner, 2016). So können Daten auch über mehrere Services hinweg nachverfolgt werden. Für eine systemübergreifende Analyse empfehlen Fichtner (2016) und Wolff (2016b) das Tool *Zipkin*, welches die Kommunikationsdaten sammelt und anschließend auf einer Weboberfläche auswertet. Abbildung 3-1 zeigt eine beispielhafte Monitoring-Landschaft einer Microservice-Architektur mit Grafana, Graphite und Zipkin.

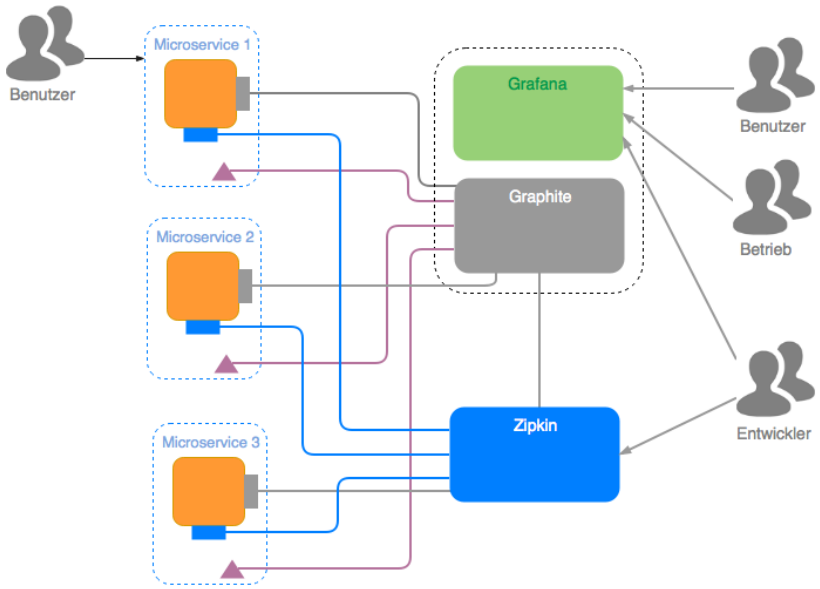

Abbildung 3-1: Monitoring-Landschaft einer Microservice-Architektur (Fichtner, 2016)

Resilienz

Resilienz oder *Resilient Software Design* beschreibt einen Ansatz angemessen auf Fehler zu reagieren. Die Zeit der Fehlerbehebung soll bestenfalls so weit verkürzt werden, dass der Anwender nichts von dem Fehler mitbekommt (Friedrichsen, 2016). Fowler & Lewis (2014) sprechen in diesem Zusammenhang von *Design for Failure*. Auch wenn ein Service in einem kooperierenden fachlichen Prozess ausfällt, sollte der Nutzerfluss dadurch nicht beeinflusst werden. Für die Isolierung der Services kann ein *Bulkhead-Muster* verwendet werden, mit dem die Services so entkoppelt werden, dass keine kaskadierenden Fehler entstehen (Friedrichsen, 2016). Für Services, die nicht verfügbar sind, kann ein *Circuit Breaker* geöffnet werden (Wolff, 2016b). Dieser fängt nach einem Ausfall eines Service weitere Anfragen an diesen ab. Wird ein Circuit Breaker geöffnet, kann anschließend bspw. ein Fehler angezeigt oder die Funktionalität eingeschränkt werden. Ein Circuit Breaker kann außerdem mit einem Timeout kombiniert werden, der den Circuit Breaker öffnet, sobald ein Service nicht mehr erreichbar ist. Ein Beispiel für die Implementierung eines Circuit Breaker zusammen mit einem Timeout ist die Bibliothek *Hystrix* von Netflix (Abschnitt 5.1).

Authentifizierung und Autorisierung

Bei der Authentifizierung und Autorisierung in einer Microservice-Architektur gibt es mehrere Möglichkeiten. Beides kann je Service oder als eigener Microservice implementiert werden (Franz, 2016). Mischformen mit einer zentralen Authentifizierung und einer dezentralen Autorisierung und umgekehrt sind ebenfalls möglich. Damit der Aufwand bei einer Implementierung je Service reduziert wird, schlägt Newman (2015) eine Bibliothek vor. Er weist darauf hin, dass bei dieser Lösung die Gefahr einer engen Kopplung besteht und dass sie voraussetzt, dass alle Services dieselbe Technologie nutzen. Franz (2016) sieht bei dieser Variante den Vorteil, dass jedes Team seine Lösung entlang der fachlichen Anforderungen entsprechend umsetzen kann. Franz (2016) und Wolff (2016b) bevorzugen den Weg, die Authentifizierung zu zentralisieren und die Autorisierung den Services selbst zu überlassen. Der

Grund ist, dass die einzelnen Services in der Regel keine unabhängige Authentifizierung in einem System vornehmen müssen, sondern, dass Benutzername und Passwort über das gesamte System hinweg gelten. Die Autorisierung hingegen bezieht sich auf den Zugriff einer speziellen Ressource und ist deshalb mit den fachlichen Anforderungen eines Service verknüpft. Für die technische Umsetzung von Authentifizierung und Autorisierung kann nach Wolff (2016b) das *OAuth2*-Protokoll mit diversen Ansätzen oder als Alternative *Kerberos*, *SAML* und *SAML 2.0* verwendet werden.

3.6 Infrastruktur und Betrieb

Für den Betrieb von Microservices wird eine Infrastruktur benötigt. Dieser Abschnitt beschreibt das Deployment, die Kommunikation und die Integration von Services in solch einer Infrastruktur.

Deployment

Auch für das Deployment von Microservices gibt es mehrere Möglichkeiten. Die Services können alleine oder mit mehreren Services auf dem gleichen Host sowie als Instanz auf einem Container oder serverlos ausgeliefert werden.

- **Eine Serviceinstanz pro Host:** Jede Serviceinstanz wird auf ihrem eigenen Host ausgeliefert. Das hat nach Richardson (2014d) den Vorteil, dass die Serviceinstanzen isoliert voneinander sind. Es gibt keine Konflikte zwischen Ressourcenanforderungen oder den Abhängigkeiten von Versionen. Es ist außerdem einfach jede Instanz erneut auszuliefern, zu verwalten oder zu überwachen. Newman (2015) empfiehlt diesen Ansatz, weil das Modell einfach zu verstehen ist und die Komplexität einer Microservice-Architektur verringert. Er weist aber darauf hin, dass mit diesem Ansatz mehrere Server verwaltet werden müssen und der Betrieb weiterer Hosts auch Kosten mit sich bringen kann. Zur Verfeinerung dieses Musters kann jeder Service zu einer Instanz einer virtuellen Maschine hinzugefügt werden und als separate virtuelle Maschine deployed werden (Richardson, 2014d).
- **Mehrere Serviceinstanzen pro Host:** Mehrere Instanzen pro Host laufen physisch oder auf einer virtuellen Maschine. Nach Richardson (2014i) gibt es zwei Möglichkeiten die Instanzen auf einem geteilten Host zu deployen - je Serviceinstanz als eigener Prozess in einer *Java Virtuell Machine (JVM)* oder mehrere Serviceinstanzen auf derselben JVM. Der Vorteil gegenüber einzelnen Instanzen pro Host ist nach Richardson (2014i) eine effizientere Ressourcennutzung. Auf der anderen Seite entstehen laut ihm Nachteile wie Konflikte bei den Ressourcenanforderungen oder den Abhängigkeiten von Versionen. Falls mehrere Serviceinstanzen auf demselben Prozess deployed sind, wird es seiner Ansicht nach außerdem schwierig die Ressourcenauslastung einzelner Services zu betrachten. Die Auslastung kann sich laut Newman (2015) zudem auf andere Services auswirken, so dass diese weniger Ressourcen zur Verfügung haben. Weiter gibt er an, dass es bei diesem Modell insgesamt schwierig wird, eine Unabhängigkeit zu gewährleisten, da sich die Services häufig beeinflussen. Dies geht seiner Meinung nach

auch zu Lasten der Entwickler, da die Zuständigkeit des Betriebs nicht mehr eindeutig ist, wenn Services verschiedener Teams auf demselben Host laufen.

- **Serviceinstanz pro Container:** Services können in ein *Container Image* hinzugefügt werden, so dass jede Instanz als ein Container deployed werden kann (Richardson, 2014f). Für die Verteilung der Container sind sogenannte *Cluster-Orchestrierer* zuständig. Eine sehr populäre Containerlösung ist *Docker*. Im Gegensatz zu virtuellen Maschinen verbraucht der Docker Container nach Richardson (2014f) weniger RAM und ist somit leichtgewichtiger. Der Grund dafür ist, dass virtuelle Maschinen einen eigenen Kernel ausführen, während Docker direkt auf dem Kernel des Host-Systems läuft und sich mit verschiedenen Mechanismen des Kernels vom Rest des Systems abschottet (Preissler & Tigges, 2015). Dies ist laut Preissler & Tigges (2015) auch der Grund, weshalb die Container schneller starten als die virtuellen Maschinen, da nur der Applikationsprozess gestartet werden muss und nicht das gesamte Betriebssystem. Auch das Bauen eines Paketes in einem Docker Container ist nach Richardson (2014f) bis zu 100 Mal schneller als das Bauen einer *Amazon Machine Instance* (AMI). Werden Microservices als Self-Contained Systems implementiert (Abschnitt 3.4), bietet Docker nach Preissler & Tigges (2015) zudem eine ideale Umgebung für diesen Ansatz.
- **Serverloses Deployment:** Hiermit wird eine Deployment-Infrastruktur bezeichnet, welche die Konzepte hinter den Servern verbirgt (Richardson, 2014j). Die Infrastruktur nimmt den Servicecode entgegen und führt ihn aus (*Function as a Service*). Um diesen Ansatz nutzen zu können, muss die Software zuerst als Paket in die Deployment-Infrastruktur hochgeladen werden. Anschließend werden die gewünschten Performanceeigenschaften beschrieben. Die Deployment-Infrastruktur ist hierbei ein Dienst, der von einem öffentlichen Cloud Provider betrieben wird, bspw. in der *AWS Cloud* mit AWS Lambda. Dabei wird entweder eine Container- oder eine VM-basierte Lösung genutzt. Das Betriebssystem, die virtuellen Maschinen und andere Low Level-Infrastrukturbereiche sind hierbei in dem Dienst versteckt und können nicht verwaltet werden. Um weiterhin Querschnittsfunktionalitäten wie Logging und Monitoring in dieser Infrastruktur zu berücksichtigen, kann auf Amazon Services wie *Cloud Watch* oder *DynamoDB* zurückgegriffen werden.

Continuous Delivery

Eine Continuous Delivery Pipeline dient dazu, Software öfter und zuverlässiger in Produktion zu bringen (Wolff, 2014). Ein Bestandteil von Continuous Delivery ist Continuous Integration. In dieser ersten Commit-Phase wird die Applikation kompiliert, anhand von Unit Tests geprüft und einer statischen Analyse unterzogen. Weitere typische Phasen sind *Akzeptanz-*, *Kapazitäts-* und *explorative Tests,* durch die die Software läuft, bevor sie in der letzten Phase in die Produktion übergeben wird. Wolff (2016a) erklärt, dass Microservices ohne eine Continuous Delivery Pipeline nicht umsetzbar sind. Der Aufbau einer Continuous Delivery Pipeline kann durch Microservices außerdem vereinfacht werden (Wolff, 2016b). Einzelne unabhängige Services laufen schneller durch die Pipeline als ein Monolith und geben daher schnelles Feedback über deren Integration. Das Risiko von Deployments wird verringert, da bei einem

Ausfall eines Service das System weiterhin genutzt werden kann. Continuous Delivery hat außerdem Auswirkungen auf die Softwarearchitektur. Benutzt man bspw. *Feature Branches,* um die Features voneinander zu entkoppeln, arbeitet man gegen die eigentliche Idee von Continuous Delivery, die Software jederzeit integrieren und ausliefern zu können (Wolff, 2016a). Eine Alternative hierzu stellen *Feature Toggles* dar. Hierbei befinden sich alle Features im selben Entwicklungsstrang und können über einen Schalter aktiviert werden, sobald diese in Produktion gehen sollen. Das hat den Vorteil, dass Features bereits vorab getestet werden können, indem man sie z.B. für bestimmte Nutzer freischaltet. In einer Microservice-Architektur stellen außerdem *Canary Releases* eine weitere Alternative dar (Zuther, 2016). Bei Canary Releases wird eine bestimmte Anzahl von Nutzern auf einen Service mit geänderter Funktionalität weitergeleitet, um zu testen, wie diese auf die Änderung reagieren.

Servicekonfiguration

Für die Konfiguration von Services wird eine Lösung benötigt, welche die Konfigurationsparameter für alle Services bereithält. Mögliche Parameter können z.B. *Credentials* sein (Vitz, 2016b). Sind die Parameter für die gesamte Laufzeit einer Serviceinstanz statisch, wird keine Infrastrukturkomponente benötigt und die Parameter können beim Deployment mit übergeben werden. Andernfalls kann die Konfiguration mithilfe eines Konfigurationswerkzeuges wie *Chef* oder *Puppet* während des Continuous Delivery-Prozesses zur Laufzeit geändert werden (Schwartz, 2016). Ändern sich Konfigurationsdaten mehrmals am Tag, empfiehlt Schwartz (2016) zentrale Konfigurationsdatenbanken zu verwenden. Vitz (2016b) weist darauf hin, dass es dabei nicht reicht, die Konfigurationen aus der Datenbank zu laden, sondern, dass auch dafür Sorge getragen werden muss, dass die Anwendung weiterhin fehlerfrei reagiert.

Skalierung

Im Gegensatz zu monolithischen Architekturen, bei denen mittels einer horizontalen Skalierung eine Replikation der gesamten Anwendung auf weitere Instanzen erfolgt, können bei einer Microservice-Architektur nur die Teile repliziert werden, für die eine Skalierung notwendig ist (Fowler & Lewis, 2014). Abbott & Fischer (2015) beschreiben hierzu einen Skalierungswürfel mit verschieden Dimensionen (Abbildung 3-2). Die Replikation der gesamten Anwendung auf mehrere Instanzen wird auf der x-Achse dargestellt und die funktionale Aufteilung der Anwendung auf der y-Achse. Die z-Achse zeigt eine Partitionierung der Daten (*sharding*), bei der jeder Instanz ein bestimmter Teil der Daten zugeordnet wird (Richardson, 2014h). Die Lastverteilung erfolgt mit einem *Load Balancer*. Ein proxy-basierter Load Balancer bspw. fragt Informationen von den Serviceinstanzen ab und verteilt anschließend die Anfragen der Clients je nach ermittelter Last auf die Instanzen (Wolff, 2016b). Fällt der Load Balancer aus, kann dies zum Ausfall eines Microservice führen. Daher empfiehlt Wolff (2016b) keinen zentralen Load Balancer für alle Microservices zu verwenden. Außerdem sollte laut Wolff (2016b) bei einer hohen Anzahl von Microservices eine dynamische Skalierung eingesetzt werden, die dafür sorgt, dass abhängig von der Last neue Instanzen gestartet werden. Die dynamische Skalierung basiert auf Metriken, welche ebenfalls für das Monitoring verwendet werden können. Über Cloud-Dienste wie *Amazon AWS* können hierfür fertige Lösungen verwendet werden. So

kann das System, z.B. je nach Nutzeraufkommen, zu bestimmten Zeiten unterschiedlich skalieren (Newman, 2015).

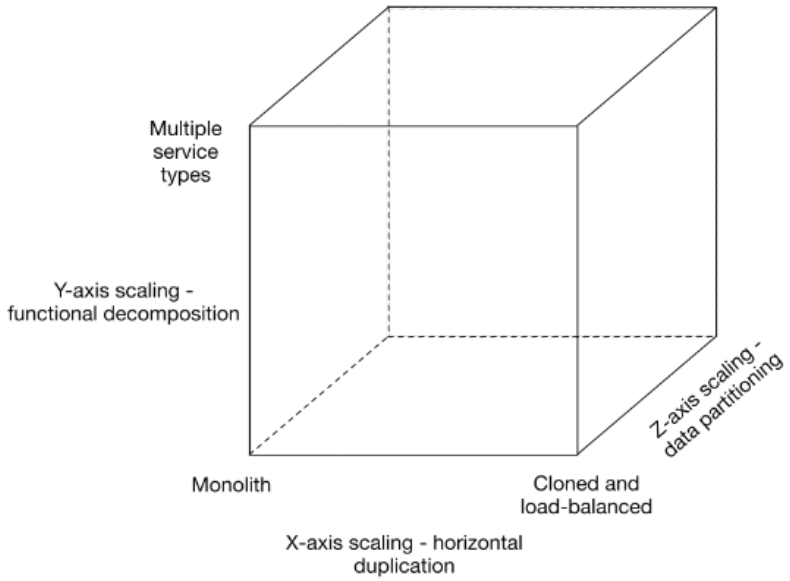

Abbildung 3-2: Der Skalierungswürfel (Richardson & Smith, 2016)

Service Registry und Discovery

In traditionellen verteilten Systemen haben Services feststehende bekannte Hosts und Ports und können einfach über HTTP/REST oder *RPC* (*Remote Procedure Call*) miteinander kommunizieren. In Microservice-basierten Applikationen, ändert sich die Anzahl der Instanzen und deren *Location* dynamisch. Es wird daher eine *Service Registry* benötigt, welche die Instanzen und die Location der Services bereithält und die Anfragen der Clients beantwortet. Die Serviceinstanzen registrieren sich beim Start bei der Service Registry und melden sich beim Herunterfahren wieder ab. Für das Auffinden der Services wird zwischen einer *Client-seitigen* und einer *serverseitigen Discovery* unterschieden (Richardson, 2015).

- **Client-seitige Discovery:** Um die Location eines Service zu ermitteln, durchsucht der Client die Service Registry und benutzt anschließend einen Load Balancing-Algorithmus, um einen der verfügbaren Services auszuwählen und eine Anfrage zu senden (Richardson, 2015). *Netflix Eureka* bspw. ist eine Service Registry des *Netflix OSS* (Abschnitt 5.1). Sie bietet eine REST API, um die Registrierung der Serviceinstanzen zu verwalten und die verfügbaren Instanzen zu durchsuchen.
- **Serverseitige Discovery:** Bei einer Anfrage an einen Service sendet der Client zunächst eine Anfrage an einen Load Balancer, dessen Location bekannt ist (Richardson, 2015). Der Load Balancer fragt schließlich die Service Registry an und leitet den Request an eine verfügbare Serviceinstanz weiter (Abbildung 3-3). Ein Beispiel für eine serverseitige Discovery ist der *Elastic Load Balancer* (*ELB*) von AWS.

Ein einheitlicher Ansatz einer *Service Discovery* hilft die Architektur übersichtlicher zu gestalten (Wolff, 2016b). Daher ist es nach Wolff (2016b) sinnvoll, die Entscheidung für eine Registry auf Makroebene zu treffen. Bei Systemen, die Messaging verwenden, kann laut ihm auf eine

Service Discovery verzichtet werden, da Messaging-Systeme Sender und Empfänger bereits entkoppeln.

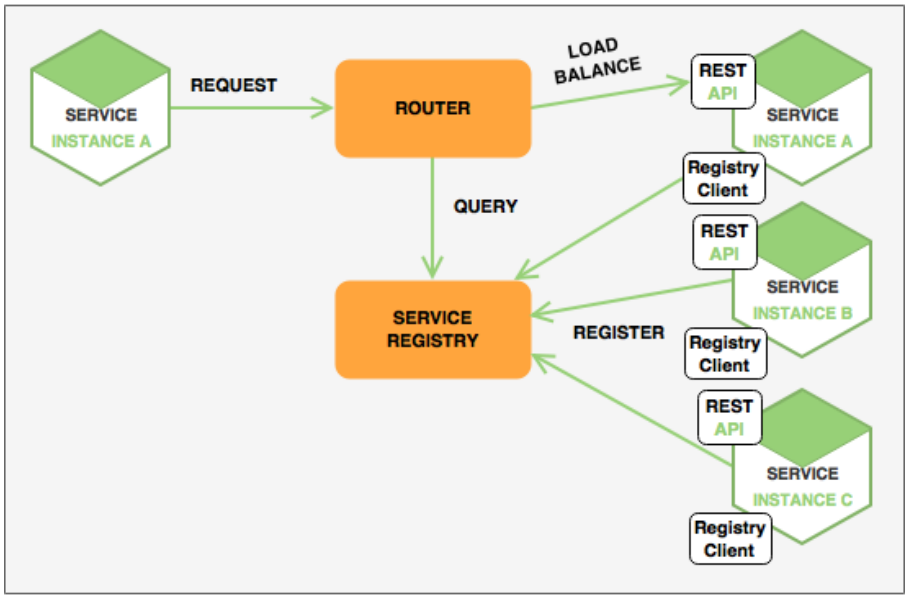

Abbildung 3-3: Serverseitige Discovery (Richardson & Smith, 2016)

API Gateway

Clients können unterschiedliche Anforderungen an die Schnittstellen der Services stellen (Richardson, 2014e). So werden z.B. unterschiedliche Daten benötigt oder je nach Client mit einer unterschiedlichen Anzahl an Microservices kommuniziert. Auch die Netzwerkperformance kann zwischen den verschiedenen Client-Typen schwanken. Eine serverseitige Webapplikation kann z.B. mehr Anfragen an Backend Services senden, während sich mobile Clients auf wenige Anfragen beschränken müssen, um den Nutzerfluss nicht zu beeinträchtigen. API Gateways können diese unterschiedlichen Anforderungen bearbeiten und isolieren somit die Services von den Clients. Hierbei agiert das API Gateway als einziger Eintrittspunkt für jeden Client (*Backend for frontend*) und leitet die Anfragen weiter oder verteilt diese ggf. auf mehrere Services. Das ist auch für die sich dynamisch ändernden Instanzen von Vorteil, da die Lokalisierung nicht über die Clients erfolgen muss. Um die Anfragen hierbei an die verfügbaren Instanzen weiterzuleiten, muss das API Gateway entweder das clientseitige oder das serverseitige Discovery verwenden.

Edge Server

Ein *Edge Server* bietet den Nutzern einen Einstiegspunkt in eine Microservice-Architektur (Vitz, 2016b). Dabei nimmt der Edge Server Anfragen von außen entgegen und verteilt sie mit Hilfe der Service Discovery an die Services (Adersberger, Siedersleben, & Weigend, 2016). Die *Edge Services* stellen nach Vitz (2016b) zusätzlich viele Funktionen wie bspw. Load Balancing, Caching, Reverse Routing, *SSL-Terminierung* und ggf. Serviceintegration bereit. Vitz (2016b) stellt außerdem einen typischen Ablauf für einen Edge Service vor. Nach dem Nutzer-Request erfolgt als erstes eine SSL-Terminierung. Anschließend kommen Load Balancer zum Einsatz, welche verhindern, dass der Eintrittspunkt des Nutzers zum *Single Point of Failure* wird. Backend-Anfragen oder statische Assets wie Bilder, CSS- oder Javascript-Dateien können

ebenfalls innerhalb der Edge Services gecacht werden. Als letztes werden die Anfragen zu dem entsprechenden Service geroutet. Techniken wie *Edge Side Includes (ESI)* oder *Server Side Includes (SSI)* erlauben es mithilfe von *Include-Direktiven* Referenzen aufzulösen, so dass der Nutzer zusammengesetzte Antworten aus mehreren Services erhalten kann.

Service Integration

Microservices können auf den folgenden Ebenen integriert werden (Wolff, 2016b):

- **Auf Benutzerschnittstellenebene:** Grundsätzlich gibt es drei Möglichkeiten wie Services auf der UI-Ebene strukturiert werden können (Zörner, 2016). Die Services besitzen eigene Benutzerschnittstellen und werden über Links integriert; die Services besitzen eigene Benutzerschnittstellen und beinhalten einen übergeordneten UI-Rahmen; die Services teilen sich eine gemeinsame Benutzerschnittstelle und beinhalten selbst nur die Geschäftslogik. Als Beispiel für eine eigene Benutzerschnittstelle kann jeder Service eine *Single Page App* (SPA) implementieren und diese mit einer JSON/REST-Schnittstelle an die interne Logik anbinden (Wolff, 2016b). Die Nachteile dieser Variante sind potentiell unterschiedlich aussehende Oberflächen und lange Ladezeiten beim Wechsel von einer SPA zur anderen. Um die Ladezeiten zu verkürzen, kann eine gemeinsame SPA zwischen den Services verwendet werden. Bei diesem Ansatz sind die Services allerdings eng gekoppelt und können möglicherweise nicht mehr unabhängig deployed werden.
- **Auf der Logikebene:** Microservices können REST, SOAP, RPC oder Messaging-Systeme nutzen, um auf der Logikebene miteinander zu kommunizieren (Wolff, 2016b). Mit dem Konzept *HATEOAS (Hypermedia as the Engine of Application State)* werden in einer REST-Architektur die Beziehungen zwischen den Systemen durch Links definiert. Über einen Einstiegspunkt können Services bspw. per *HAL* über weitere Links navigieren und daher transparent für andere Clients verschoben werden.
- **Auf der Datenebene:** Die Services können sich für die Integration eine gemeinsame Datenbank teilen. Diese Integrationsform hat nach Newman (2015) und Wolff (2016b) mehrere Nachteile. Da alle Services auf eine Datenbank zugreifen, müssen Änderungen zwischen den Services koordiniert werden. Außerdem kann das Schema der Datenbank nicht einfach von einem Team geändert werden, da eine Änderung den Zugriff eines anderen Service einschränken könnte. Auch das Shared Nothing-Prinzip ist verletzt und eine Technologieunabhängigkeit auf Persistierungsebene nicht mehr gegeben. Müssen dennoch gemeinsame Daten geteilt werden, kann dies über Events erfolgen (Abschnitt 3.4).

Consumer-driven Contracts

Consumer-driven Contracts (CDC) sorgen dafür, die Schnittstellen zwischen den Services stabil zu halten (Vitz, 2016a). Bei CDC spezifiziert der Nutzer einer Schnittstelle wie er diese nutzen will. Die somit vereinbarten Verträge können anschließend automatisiert gegen eine Schnittstelle geprüft werden. Der Serviceanbieter kann seine Schnittstelle weiterhin ändern, sofern der Vertrag weiter erfüllt wird, was zur Rückwärtskompatibilität des Service führt. Eine

Implementierung von CDC stellt *Pact* dar. Hier definiert der Aufrufer im ersten Schritt die durchzuführenden Aufrufe und die zu erwarteten Antworten innerhalb eines Unit Tests in der Sprache des Aufrufers. Nach der Ausführung des Tests wird mithilfe von Pact ein Server gestartet, der in dem zuvor definierten Rahmen antwortet. Währenddessen wird eine Pact-Datei generiert, die alle spezifizierten Aufrufe zusammen mit den Antworten in einem *JSON*-Format enthält. Anschließend wird die Schnittstelle mit den spezifizierten Anfragen aus der Datei aufgerufen und die zurückgegebenen Antworten werden mit den erwarteten abgeglichen.

Testen von Standards

Nach Wolff (2016b) lassen sich bestimmte Eigenschaften von Microservices automatisiert testen. Das hat den Vorteil, dass sich die gemeinsamen Regeln für alle Services jederzeit überprüfen lassen. Somit kann laut Wolff (2016b) das korrekte Verhalten neuer Komponenten gewährleistet werden, auch wenn diese z.B. einen abweichenden Technologie-Stack nutzen. Zu den möglichen Szenarien zählt er skriptbasierte Tests für Service Discovery, Konfiguration, Monitoring, Deployment, Steuerung, Sicherheit und Resilienz.

3.7 Abgrenzung zu SOA

Eine Serviceorientierte Architektur (SOA) ist eine unternehmensweite IT-Architektur, die mit Services als zentrales Konzept hilft, Geschäftsfunktionen zu realisieren oder zu unterstützen (Starke, 2014). Wie bei Microservices standen bei SOA anfänglich lose Koppelung, Isolation, Komposition und Integration von autonomen Services im Vordergrund (Bonér, 2016). Der Fokus verschob sich allerdings, aufgrund von Fehlinterpretationen, in Richtung technologische Debatten (Newman, 2015). Schließlich gab es nach Newman (2015) trotz mehreren Versuchen keinen Konsens über die richtige Umsetzung einer SOA-Technologie. Microservices dagegen kommen aus praxisnahen Ansätzen, etabliert von Firmen wie Netflix, Amazon und Spotify und können nach Newman (2015) und The Open Group (2016) als bestimmter SOA-Ansatz angesehen werden. Im Gegensatz zu SOA können Microservices für einzelne Projekte eingesetzt werden, ohne sich auf die Unternehmensarchitektur auszuwirken (Wolff, 2016b). Nach Richards (2016) passt eine Serviceorientierte Architektur zu großen und komplexen Enterprise-Systemen, in die mehrere heterogene Applikationen und Services integriert werden. Microservices dagegen passen besser zu kleinen, gut aufgeteilten webbasierten Systemen mit wenigen gemeinsam genutzten Komponenten. Auch weil Microservices API Layer benutzten, anstelle von Messaging Middleware, sind nach Richards (2016) diese eher unpassend für komplexe Geschäftsanwendungsumgebungen. Werden jedoch Enterprise-Systeme in eine Microservice-Architektur überführt, ist es nach Soika (2016) notwendig, auch übergeordnete Geschäftsprozesse zu berücksichtigen. Er stellt hierzu eine *Business Process Service-Architektur* vor, bei der ein Geschäftsprozess als Microservice implementiert wird und anschließend als zentraler Koordinator (*Service-Orchestrierer*) fungiert. Diese Architektur hat den Vorteil, dass Services, welche einen gemeinsamen fachlichen Prozess abbilden und sequentiell Events versenden müssen, nicht mehr über die Prozesslogik gekoppelt sind. Laut Flohre (2015) hat diese Lösung jedoch den Nachteil, dass das Team, welches so einen

Business Process Service betreut, Fachlichkeit aus mehreren Services implementieren muss. Dies kann eine erhöhte Koordination beim Deployment zur Folge haben und beeinträchtigt somit die Änderbarkeit. Die entscheidende Frage an dieser Stelle ist nach Flohre (2015), ob eine zentrale Kontrolle oder Änderbarkeit in einer Microservice-Architektur bevorzugt wird.

4 ANSÄTZE DER MODELLIERUNG

In diesem Kapitel werden Ansätze der Modellierung aus der Literatur dargelegt. Abschnitt 4.1 zeigt Modelle, die Microservice-Architekturen aus einer ganzheitlichen Sicht heraus betrachten. In Abschnitt 4.2 und 4.3 folgen Beispiele aus der Makroarchitektur und der fachlichen Architektur. In Abschnitt 4.4 wird erläutert, wie Legacy-Anwendungen in eine Microservice-Architektur integriert werden können.

4.1 Ganzheitliche Modelle

Im Folgenden werden drei Modelle eingeführt, die eine ganzheitliche Sicht auf eine Microservice-Architektur ermöglichen.

Das System Design Model

Nadareishvili, et al. (2016) verwenden ein *System Design Model* (Abbildung 4-1), um die für eine Microservice-Architektur relevanten Systemteile darzustellen. Das Modell zeigt folgende fünf Elemente:

- **Service:** Entwurf von Schnittstellen und Umfang einzelner Services (Fachliche Architektur).
- **Solution:** Entwurf der Makroarchitektur, um die Komplexität einzelner Services zu reduzieren.
- **Process and Tools:** Tools und Prozesse in Bezug auf die Entwicklung, das Deployment, die Wartung und das Produktmanagement.
- **Organization:** Struktur, Hierarchie, Granularität und Zusammensetzung der Teams.
- **Culture:** Werte, Überzeugungen und Ideale, die Entwurfsentscheidungen direkt beeinflussen können.

Die Elemente bilden zusammen ein emergentes System und können somit das Verhalten des Systems ändern (Abschnitt 2.1.3). Wird bspw. die Größe eines Teams geändert, kann das einen direkten Einfluss auf den Service haben, an dem das Team arbeitet. Je nach Organisation kann außerdem der Rahmen für die Makroarchitektur ein anderer sein (Wolff, 2016b). Auch umgekehrt haben Microservices Auswirkungen auf Organisation und Kultur. Beispielsweise können sich aus einer isolierten, unabhängigen Architektur mit kleinen Services, kleine, isolierte und unabhängige Teams entwickeln (*Inverse Conways Law*) (Fowler S. , 2017).

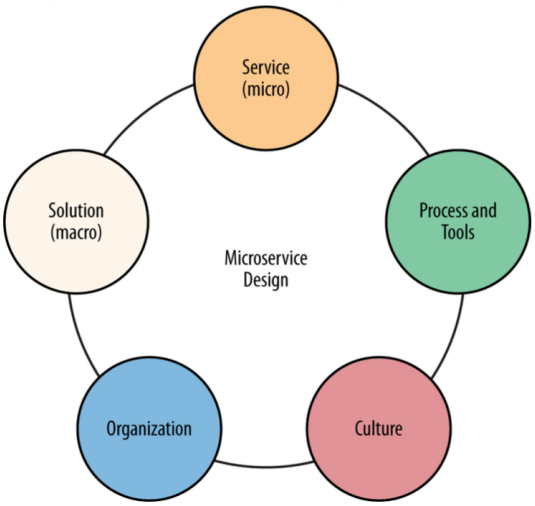

Abbildung 4-1: The System Design Model for Microservices (Nadareishvili, et al., 2016)

Das Capability Model

Als Business Capability wird die Fähigkeit einer Organisation bezeichnet, eine Aktivität durchzuführen, die einen Wert erzeugt (Reinhard, 2011). Das Capability Model (Abbildung 4-2) zeigt diese Fähigkeiten gruppiert im Kontext einer Microservice-Architektur (RV, 2016). Das Modell ist in vier Bereiche untergliedert:

- **Core capabilities:** Alle Fähigkeiten, die direkter Bestandteil der Services selbst sind. Hierzu zählen Service Listener, Speichermechanismen, Event Sourcing, Serviceendpunkte und Kommunikationsprotokolle sowie ein API Gateway und Benutzerschnittstellen.
- **Supporting capabilities:** Softwarelösungen, welche die Kern-Microservices unterstützen, wie Load Balancer, ein zentrales Log-Management, Service Registry, Servicekonfiguration, Testing Tools, Monitoring und Tools für das Architekturmanagement.
- **Infrastructure capabilities:** Die Erwartungen an die Infrastruktur für eine erfolgreiche Microservice-Integration, z.B. Cloud-Lösungen, Container oder virtuelle Maschinen und eine Clusterverwaltung. Dazu gehören auch Application Lifecycle Management Tools, welche z.B. in Kombination mit der Clusterverwaltung automatisiert Fehlerszenarien aufdecken und entsprechend reagieren.
- **Governance capabilities:** Prozess-, Kultur- und Referenzinformationen wie *DevOps*, DevOps Tools, ein Microservice-Repository, Dokumentation der Services und Referenzarchitekturen und Bibliotheken.

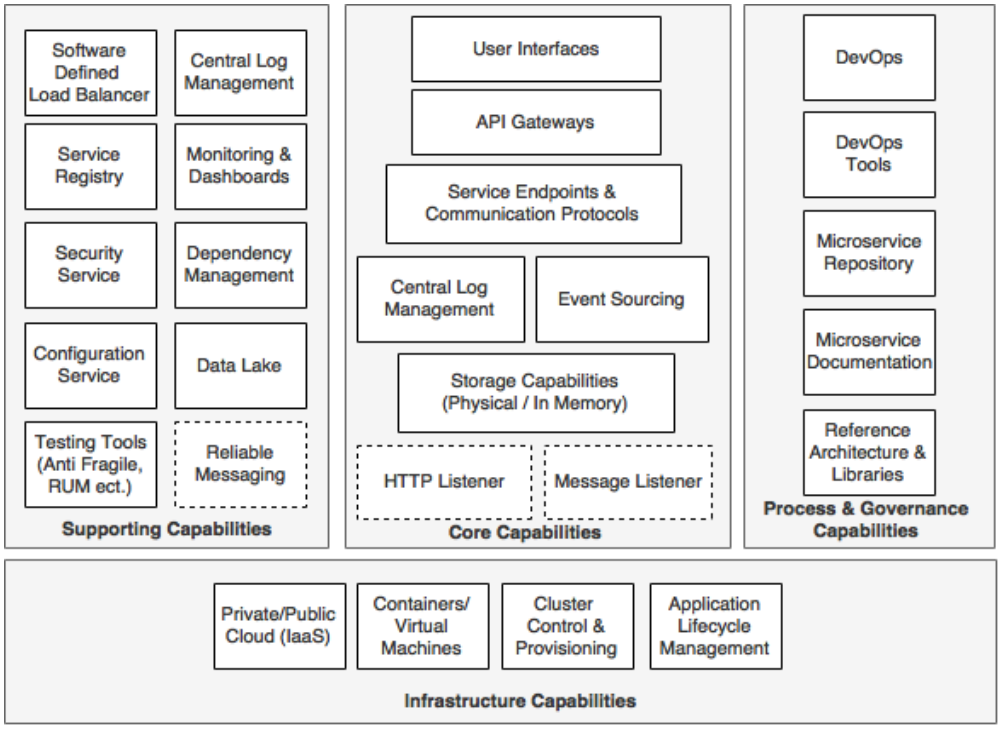

Abbildung 4-2: The capability model for Microservices (RV, 2016)

Das Four-Layer Model

Das Four-Layer Model (Abbildung 4-3) basiert auf der Idee eines Microservice-Ökosystems, in dem die Microservices von allen infrastrukturellen Aufgaben abstrahiert sind (Fowler S. , 2017). Die Grenzen der Schichten sind nicht klar definiert und einige Elemente der Infrastruktur ziehen sich durch alle der folgenden Schichten hindurch:

- **Hardwareschicht:** Auf der Hardwareschicht befinden sich alle physischen Maschinen, auf denen die Microservices laufen. Hier wird außerdem festgelegt, welches Betriebssystem auf den Servern installiert wird, wie Services konfiguriert werden und auf welche Weise das Monitoring und Logging durchgeführt wird.
- **Kommunikationsschicht:** Die Kommunikationsschicht zieht sich durch alle Schichten hindurch. Zu den Elementen der Kommunikationsschicht zählen das Netzwerk, DNS, RPCs, API Endpoints, Service Registry und Discovery und Load Balancing.
- **Applikationsplattform:** Die Applikationsplattform beinhaltet alle von den Microservices unabhängigen Werkzeuge und Dienste. Eine gute Applikationsplattform zeichnet sich nach Fowler S. (2017) dadurch aus, dass sie möglichst viel vereinheitlicht und zentrale Lösungen bspw. für Deployment, Monitoring und Logging bereitstellt.
- **Microservice:** Auf der obersten Schicht werden die Microservices von den anderen Schichten abstrahiert. Ausnahme sind die Konfigurationen, die für die Verwendung der Werkzeuge für jeden Microservice spezifisch sind. Die Konfigurationen können über Repositories innerhalb der Microservices, den Systemen und Werkzeugen der darunterliegenden Schichten zur Verfügung gestellt werden.

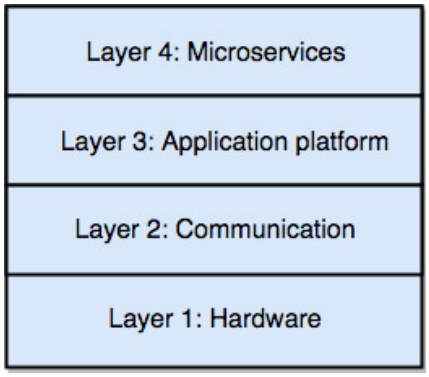

Abbildung 4-3: Das Four-Layer Model (Fowler S. , 2017)

4.2 Makroarchitektur

Die Makroarchitektur sollte nur die Informationen beinhalten, die ihr auch eindeutig zugeordnet werden können. Die Trennlinie zwischen Mikro- und Makroarchitektur ist jedoch nicht eindeutig definiert (Zörner, 2016). Wie im System Design Model beschrieben, können Organisation und Kultur des Unternehmens die Makroarchitektur beeinflussen. Tabelle 4-1 zeigt, welche Aufgaben nach Wolff (2016b) der Makroarchitektur eindeutig und welche ihr potentiell zugeordnet werden können.

Mögliche Zuordnung	**Eindeutige Zuordnung**
Programmiersprache / Plattform	Sicherheit
Konfiguration und Koordination	Service Discovery
Resilienz	Kommunikationsprotokolle
Load Balancing	Integrationstests
Monitoring	
Logging	
Übergreifende fachliche Architektur	
Technische Tests, Stubs, CDC-Tests	

Tabelle 4-1: Umfang der Makroarchitektur

Preissler & Tigges (2015) teilen die Makroarchitektur in zwei Bereiche der Systemarchitektur: Die Kommunikation zwischen den Services, inklusive der zu verwendeten Protokolle, und die Integration der Services in die Plattform (Infrastruktur). Die übergreifende fachliche Architektur wird nach Preissler & Tigges (2015) als weitere Architekturebene (Domänenarchitektur) betrachtet. Die vorliegende Arbeit folgt dieser Unterteilung und trennt im weiteren Verlauf die fachliche Architektur von der technischen Makroarchitektur.

Infrastruktur

Abbildung 4-4 zeigt ein Beispiel für eine Infrastruktur für Microservices. Die Zielarchitektur wird dort in drei horizontale Schichten geteilt (Larsson, 2015). Die *Core Services* kümmern sich um

die Persistierung der Geschäftsdaten und die Anwendung von Geschäftsregeln und weiterer Logik. Die *Composite Services* können entweder eine Anzahl von Kerndiensten orchestrieren, um eine gemeinsame Aufgabe zu erfüllen oder Informationen aus einer Anzahl von Kerndiensten aggregieren. Die *API Services* bieten eine Schnittstelle nach außen, um z.B. Drittanbietern einen Zugriff auf die Funktionalität der Systemlandschaft bereitzustellen. Der Zugriff von außen erfolgt über einen Edge Server, der gleichzeitig mit Hilfe des OAuth-Protokolls die API Services vor unautorisiertem Zugriff schützt. Über die Service Discovery können sich die Microservices registrieren. Sollten mehrere Instanzen eines Service existieren, entscheiden die Load Balancer (LB), an welchen Service die Anfrage geroutet wird. Das *Monitor Dashboard* zeichnet den Status der Circuit Breaker (CB) auf und kann somit über deren Laufzeitstatistiken einen Überblick über den Health-Status der Systemlandschaft geben. Der *Configuration Server* stellt den Services eine zentrale Konfigurationsverwaltung bereit. Das *Logging Analyses Dashboard* sammelt schließlich alle Log-Daten, welche die einzelnen Services produzieren.

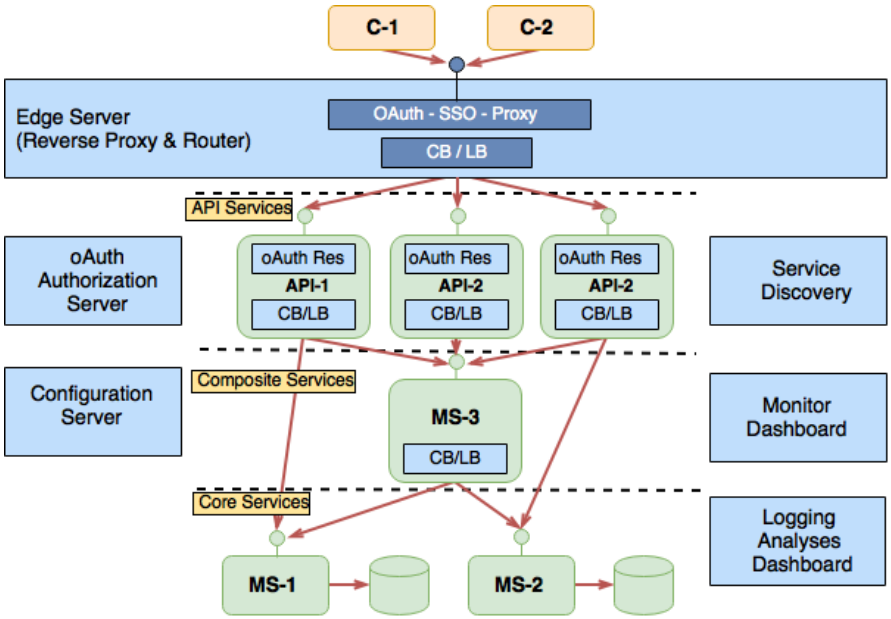

Abbildung 4-4: An operations model for Microservices (Larsson, 2015)

Kommunikation

Die Interprozesskommunikation kann auf verschiedene Arten umgesetzt werden. Richardson & Smith (2016) empfehlen, sich vor der Auswahl des Kommunikationsmechanismus die Interaktionsmöglichkeiten vor Augen zu führen. Die möglichen Varianten werden in Tabelle 4-2 in *synchron* und *asynchron* sowie *one-to-one* und *one-to-many* unterteilt.

	ONE-TO-ONE	ONE-TO-MANY
SYNCHRONOUS	Request/Response	-
ASYNCHRONOUS	Notification	Publish/Subscribe
	Request/async response	Publish/async responses

Tabelle 4-2: Interprozess Kommunikationsarten (Richardson & Smith, 2016)

- **Request/Response:** Ein Client sendet eine Anfrage und wartet auf die Antwort in einer angemessenen Zeitspanne. In einer Thread-basierten Applikation kann es dabei zu einem Block des wartenden Threads kommen.
- **Notification:** Ein Client sendet eine Anfrage an einen Service ohne eine Antwort zu erwarten.
- **Request/async response:** Ein Client sendet eine Anfrage an einen Service, welcher asynchron antwortet. Es wird davon ausgegangen, dass die Antwort mit einem großen zeitlichen Verzug erfolgen kann. Der Client blockt nicht während er wartet.
- **Publish/Subscribe:** Ein Client veröffentlicht eine Benachrichtigung, welche von keinem oder mehreren Services abonniert wurde.
- **Publish/async responses:** Ein Client sendet eine Anfrage und wartet eine bestimmte Zeit auf die Antworten der abonnierten Services.

Die asynchrone Kommunikation basiert auf Messaging-Systemen und kann entweder Punkt-zu-Punkt oder mit Publish/Subscribe über einen Channel erfolgen. Protokolle, die hierbei oft genutzt werden, sind *AMQP*, *STOMP* oder *MQTT* (Indrasiri, 2016). Synchrone Request/Response-Kommunikation kann mithilfe von HTTP-basierten Mechanismen wie REST oder *Thrift* umgesetzt werden. Nach Richardson & Smith (2016) ist es nicht notwendig, sich auf einen IPC-Mechanismus festzulegen. Oft verwenden die Services eine Kombination von verschiedenen Mechanismen (Abbildung 4-5).

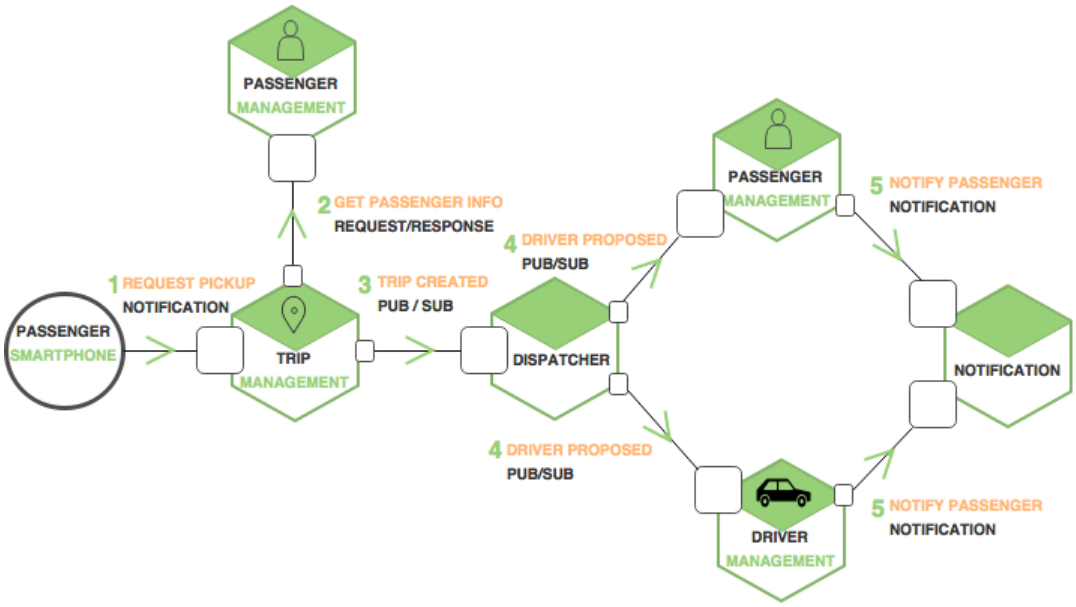

Abbildung 4-5: Verschiedene IPC-Mechanismen für Serviceinteraktionen (Richardson & Smith, 2016)

In dem Beispiel kommuniziert der Client asynchron mittels einer Notification mit dem Trip-Management. Der Trip Management Service sendet eine synchrone Anfrage, um zu prüfen, ob der Account des Clients aktiv ist. Anschließend erstellt das Trip-Management eine Reise und nutzt den Publish/Subscribe-Mechanismus, um weitere Services zu benachrichtigen. Ein solches Kommunikationsmodell hilft dabei ineffiziente Kommunikationsarten zu identifizieren.

Wird bspw. eine Kette von synchronen Aufrufen zwischen mehreren Services verwendet, so summiert sich die Antwortzeit, was zu einer erhöhten Latenz führt.

4.3 Fachliche Architektur

Die fachliche Architektur kann in einer Microservice-Architektur ebenfalls übergreifend und intern modelliert werden. Um die übergreifende fachliche Architektur abzubilden, kann eine Context Map verwendet werden, welche die Bounded Contexts einer Microservice-Architektur visualisiert (Abbildung 4-6). In dem Beispiel aus Wolff (2016b) erfasst die Registrierung die Basisdaten eines Kunden und der Bestellprozess kommuniziert auf Basis desselben Datenformats mit dem Kunden. Die Basisdaten des Kunden werden im Bestellprozess um weitere Daten wie Rechnungs- oder Lieferadressen ergänzt. Das Domänenmodell wird hier mit einem Shared Kernel geteilt. Die Lieferung und Rechnung benutzen ebenfalls die Kundenbestelldaten für die Kommunikation. Die Lieferung nutzt außerdem die Daten für die interne Repräsentation der Kunden. Das Mainframedatenmodell wiederum verwendet ein veraltetes Datenmodell für die interne Repräsentation. Aus diesem Grund werden die Daten über den Anticorruption Layer entkoppelt, so dass die Kommunikation sich nach außen nicht auf andere Microservices auswirken kann.

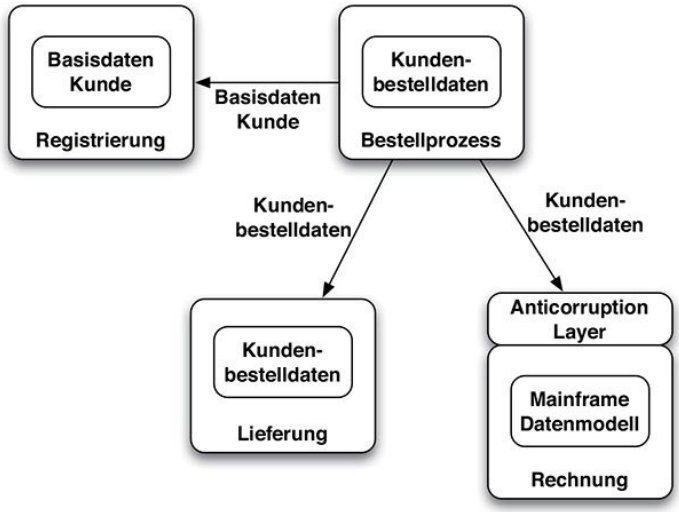

Abbildung 4-6: Beispielhafte Context Map (Wolff, 2016b)

Die Modellierung der Abhängigkeitsformen hilft dabei Abhängigkeiten noch vor der Implementierung zu lösen oder im Nachhinein zu entscheiden, wie Funktionalität ausgelagert werden soll. Teilen sich bspw. mehrere Services wie in Abbildung 4-6 Teile des Modells kann eine Funktion nicht einfach in einen neuen Service ausgelagert werden, ohne dass Abhängigkeiten berücksichtigt werden müssen. Die Abhängigkeitsformen wirken sich außerdem direkt auf Conways Law aus, da je nach Grad der Abhängigkeit der Koordinations- und Kommunikationsaufwand zwischen den Teams steigt (Plöd, 2016).

Für die interne Strukturierung der Bounded Contexts unterstützen die im DDD definierten Building Blocks. Abbildung 4-7 stellt Aggregate aus Plöd (2016) dar, welche Entitäten desselben fachlichen Bereichs zusammenfassen. Die fachlichen Blöcke *Customer* und *Loan*

Application Form fungieren als Root-Entitäten, die als einzige von außen referenziert werden dürfen und den gesamten Lebenszyklus der Aggregate steuern. Für eine lose Koppelung sollten nach (Richardson, 2016) die Aggregate Identitäten anstelle von Objektreferenzen verwenden, um sich gegenseitig zu referenzieren. So kann bspw. der Bestellprozess aus Abbildung 4-6 eine Kunden-ID anstelle einer Referenz auf das Kundenobjekt nutzen. Das hat nach (Vernon, 2017) u.a. den Vorteil, dass aufgrund der fehlenden Objektreferenz andere Aggregate nicht einfach innerhalb der gleichen Transaktion geändert werden können.

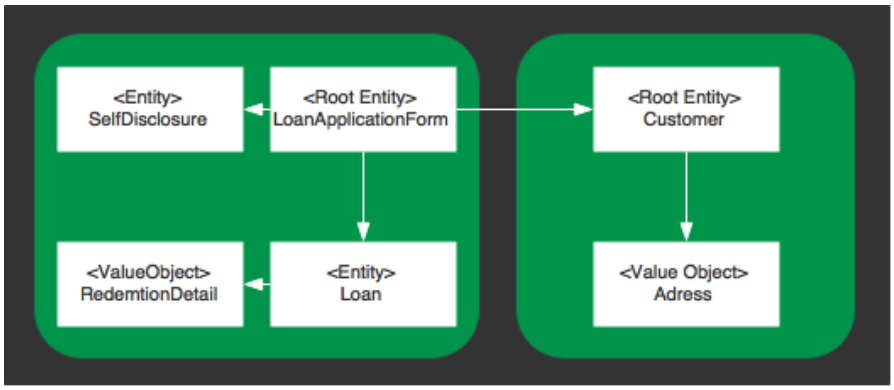

Abbildung 4-7: Fachlich zusammenhängende Entitäten in Aggregaten (Plöd, 2016)

Aggregate sind auch dafür geeignet, die in Abschnitt 3.1 diskutierte Größe von Microservices zu bestimmen (Plöd, 2016). Ein Microservice sollte nach Plöd (2016) mindestens so groß sein wie ein Aggregat, aber nur maximal so groß wie ein Bounded Context.

Architekturmanagement

Die manuelle Pflege einer Context Map ist nur schwer möglich, da sich die Servicelandschaft dynamisch ändert. Wolff (2016b) und Newman (2015) schlagen deshalb vor, die nötigen Informationen direkt aus den Services zu beziehen. Diese Informationen können anschließend an einer zentralen Stelle abgelegt werden. Fowler (2008) nennt dieses Verfahren eine *Humane Registry*. Jeder Service könnte nach Wolff (2016b) bspw. seine Abhängigkeit zu anderen Services dokumentieren und diese Information in einem einheitlichen Format zurückgeben. Über die Service Discovery kann ermittelt werden, welche Services es gibt und mit wem diese in Verbindung stehen. Wolff (2016b) nennt außerdem die Regeln von Firewalls und die Kommunikation im Netzverkehr als weitere Möglichkeiten, um die Abhängigkeiten zu identifizieren. Als Werkzeuge, die bei der Visualisierung unterstützen, schlägt er *Structure 101*, *Gephi* und *jQAssistant* vor. Die Service-APIs können außerdem mit HAL und *Swagger* explizit gemacht werden (Newman, 2015). Eine komplette Lösung liefert das *Ordina Dashboard* für Spring-Boot, welches mit einem Microservice-Dashboard in Verbindung steht (Schlosser, 2016b). Newman (2015) hält hierzu fest, dass die Beschreibung von komplexen Systemen sich von manuellen Dokumentationen in das Monitoring verschiebt. Neben dem Betriebszustand können dort weitere Informationen zu Abhängigkeiten, Funktionalitäten und Location der Services festgehalten werden. Wolff (2016b) stellt dagegen die Notwendigkeit eines Gesamtüberblicks in einer komplexen Microservice-Architektur in Frage. Er argumentiert, dass die altbekannten Prinzipien wie Kapselung, Modularisierung und Information Hiding bei

Microservices noch mehr zum Tragen kommen und sich das notwendige Wissen über das System daher geringhält.

4.4 Legacy-Systeme

Oft werden Anwendungen nicht für eine Microservice-Architektur entworfen, sondern basieren auf einem bestehenden Legacy-System. Das Legacy-System kann dann entweder langsam in ein Microservice-System überführt oder mit dem neuen System verbunden werden. Letzteres wird auch als *Strangler Application* bezeichnet (Fowler M. , 2004). Abbildung 4-8 zeigt eine E-Commerce-Anwendung, die Daten einer Produktdatenbank anzeigt und Bestellungen in einer Bestelldatenbank speichert (Krause, 2015). Auf der linken Seite befindet sich das Legacy-System. Dort sind die Geschäftslogik und die Datenzugriffslogik eng miteinander verbunden. Die Produktkomponenten können nun schrittweise von den Bestellkomponenten getrennt werden. Für alle identifizierten Bounded Contexts werden einzelne Services erstellt. Ein API Gateway routet die Anfragen und entkoppelt somit die Clients von den Services. Das Bestellsystem kann nun beliebig skaliert werden. Die Koppelung zwischen der Produktsuche und dem Produktservice wird aufgelöst, indem man der Produktsuche ausschließlich lesenden Zugriff auf die Daten gewährt. Um eine Kommunikation zwischen den Services zu ermöglichen, kann das Beispiel nach Krause (2015) um einen *Message Broker* ergänzt werden, der die Nachrichten in einer für die Transaktionen relevanten Reihenfolge verteilt.

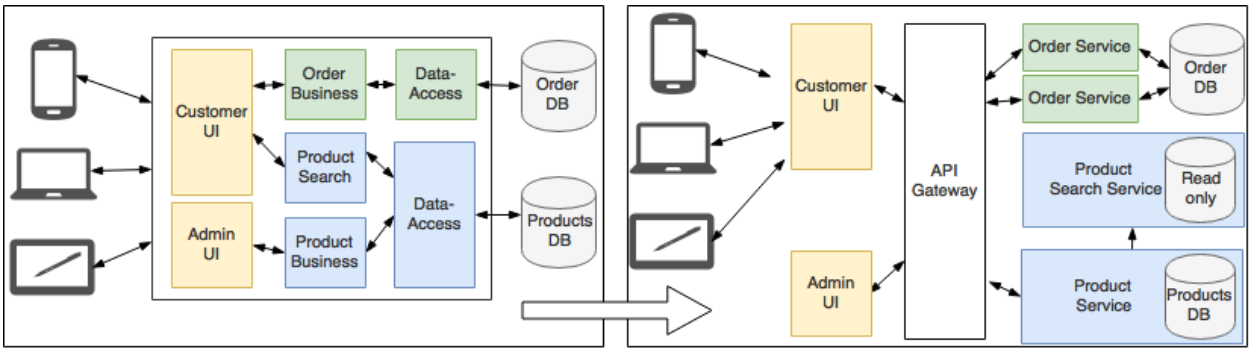

Abbildung 4-8: Überführung eines Legacy-Systems, angelehnt an Krause (2015)

Der Message Broker gehört zu den *Enterprise Integration Patterns* und kann ebenfalls für eine reine Verbindung zwischen dem Legacy-System und dem Microservice-System genutzt werden (Wolff, 2016b). Dabei entscheidet der *Context-Based Router* abhängig vom Inhalt der Nachricht, wohin die Nachricht gesendet wird. Ein *Message Filter* sorgt dafür, dass ein Microservice keine Nachrichten erhält, die nicht an ihn adressiert sind. Mithilfe eines *Message Translators* kann die Nachricht übersetzt werden, falls die Microservice-Architektur ein anderes Format als die Legacy-Anwendung benutzt. Möchte man die Informationen aus der Legacy-Anwendung ergänzen, kann man einen *Content Enricher* verwenden. Der *Content Filter* wiederum sorgt dafür, dass überflüssige Informationen entfernt werden. Nach Vernon (2017) sollte außerdem das *At Least Once Delivery-Muster* bei der Integration verwendet werden. Bei diesem Verfahren werden die Nachrichten bspw. bei Nachrichtenverlust, hohen Latenzen oder ausgefallenen Empfängern regelmäßig erneut ausgeliefert. Der Empfänger sollte in diesem Fall

nach dem *Idempotent Receiver*-Muster immer mit derselben Operation antworten oder bereits empfangende Nachrichten ignorieren. Eine weitere Möglichkeit, um zwei Systeme miteinander zu verbinden, stellen Richardson & Smith (2016) vor. Eingehende HTTP-Requests werden über einen Request Router an das Legacy-System oder an einen Microservice weitergeleitet. Mithilfe eines Anticorruption Layers wird das neue System in das Legacy-System integriert. Das neue System kann bspw. über den Layer eine Remote API aufrufen, direkt auf die Datenbank des Legacy-Systems zugreifen oder dessen Daten synchronisieren.

5 MODELLIERUNG IN DER PRAXIS

In diesem Kapitel wird untersucht, wie Unternehmen Microservices in der Praxis modellieren. Hierfür wird in Abschnitt 5.1 zunächst die Netflix-Architektur näher betrachtet. Im Anschluss daran wird in Abschnitt 5.2 dargestellt, wie Unternehmen ihre Makroarchitektur definieren und welche Informationen diese über ihre Infrastruktur bereitstellen. Anschließend folgen in Abschnitt 5.3 und Abschnitt 5.4 praktische Beispiele des Architekturmanagements und der Integration von Microservices in Legacy-Systeme, bevor in Abschnitt 5.5 abschließend auf die Verantwortlichkeiten in einer Microservice-Architektur eingegangen wird.

5.1 Die Netflix-Architektur

Netflix war einer der „Early Adopters“ von Microservices (Mauro, 2015). Das Unternehmen wechselte von einem traditionellen Entwicklungsmodell mit einer monolithischen Architektur zu einer Microservice-Architektur mit vielen kleinen Teams, die für die Entwicklung von hunderten von Microservices verantwortlich waren. Das Entwicklerteam etablierte unter dem Cloud-Architekten Adrian Cockroft mehrere Best-Practices für die Entwicklung einer Microservice-Architektur:

- **Ein eigener Datenspeicher für jeden Microservice:** Jedes Team wählt einen eigenen Datenspeicher, der am besten zum Service passt. Um Inkonsistenzen zwischen den Daten der Services zu beheben, wird ein *Master Data Management System* (MDD) benötigt.
- **Ein gleicher Reifegrad des Codes:** Wird Code zu einem bestehenden Microservice hinzugefügt oder neugeschrieben, ist für Netflix der beste Ansatz, diesen in einen neuen Service zu verlagern und den alten Service solange bestehen zu lassen, bis der neue Code stabil ist.
- **Ein separater Build für jeden Service:** Die Services können sich bei dieser Variante ohne Gefahr die Dateien mit der Revision aus dem Repository holen, die sie benötigen. Der Nachteil ist, dass verschiedene Microservices evtl. gleiche Dateien mit unterschiedlichen Revisionen vorhalten, was die Bereinigung der Codebasis von veralteten Revisionen erschwert.
- **Deployment in Containern:** Die Auslieferung in Containern hat den Vorteil, dass nur ein Tool für das Deployment benötigt wird.
- **Zustandslose Server:** Server sollen als austauschbare Einheiten einer Gruppe behandelt werden, die alle die gleichen Funktionen ausführen. Fällt ein Server aus, kann er somit problemlos von einem anderen ersetzt werden.

Abbildung 5-1 zeigt eine vereinfachte Struktursicht der Netflix-Architektur (Toth, 2015; Zörner, 2015). Die Clients verbinden sich über HTTP-Requests mit einem Elastic Load Balancer von Amazon, der die Anfragen auf die Streaming API oder das API Gateway verteilt. Die beiden APIs bilden zusammen den Edge Service, welcher mit *Zuul* aus der hauseigenen Netflix OSS (Open Source Software) umgesetzt ist. Um die Clients an den zu ihrem Device passenden Service weiterzuleiten, verwendet das API Gateway gerätespezifische Endpoint-Adapter. Der *Fault Tolerance Layer* wird durch Hystrix umgesetzt, einer Latenz- und Fehlertoleranzbibliothek, die ebenfalls aus der Netflix OSS stammt. Hystrix fungiert als Circuit Breaker, isoliert die Zugangspunkte zwischen den Services, stoppt kaskadierende Fehler und bietet Fallback-Möglichkeiten an. Die Resilienz wird mit der *Simian Army* getestet, welche Ausfälle simuliert oder Anomalien im System aufdeckt (Tseitlin, 2013). Informationen über die Zuverlässigkeit sendet Hystrix an das Monitoring System *Atlas*. Die Service Registry (Netflix Eureka) steuert die Lastverteilung der Mid-Tier Services. Eureka bietet außerdem ein Dashboard über angemeldete Dienste und weitere Metadaten. Zur Persistierung werden Cassandra, in-memory caches, *Amazon S3* oder *MySQL* verwendet (Toth, 2015). Die Querschnittsfunktionalitäten, wie Logging, Configuration und Security, werden von weiteren *Platform Services* bereitgestellt, wobei viele dieser Aufgaben ebenfalls über den Edge Service realisiert werden (Pelka & Plöd, 2016). Für die Interprozesskommunikation wird Netflix *Ribbon* verwendet.

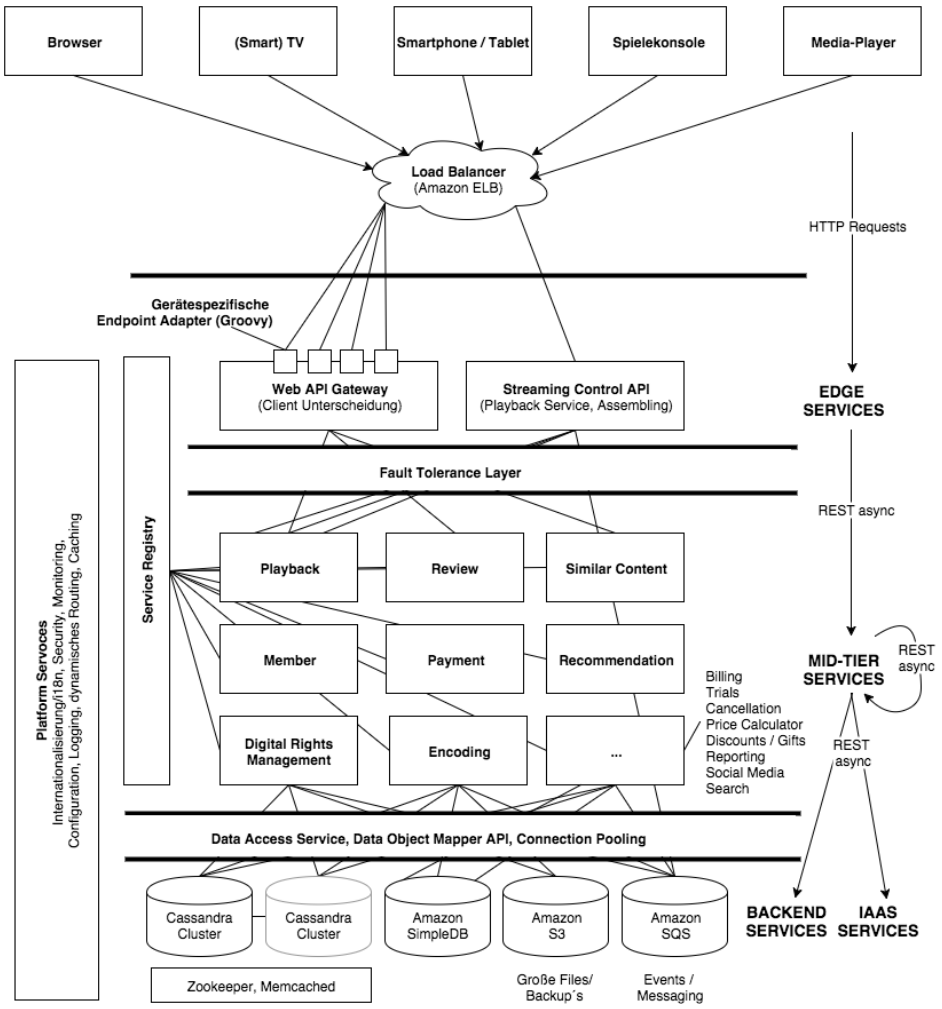

Abbildung 5-1: Die Netflix-Architektur, angelehnt an Toth (2015) und Zörner (2015)

5.2 Makroarchitektur

In der Praxis führen einige Unternehmen Richtlinien ein, um ihre Makroarchitektur explizit zu machen. Tabelle 5-1 zeigt die Richtlinien von der Hypoport AG, Otto.de und Zalando im Vergleich.

Hypoport AG	Otto.de	Zalando
Continuous Deployment	Vertikale Dekomposition	Microservices
DNS-Namensschema	REST	API First
Logging	Shared Nothing Architecture	REST
Monitoring	Data Governance	Cloud
Metrikenanzeige		Software as a Service
Request-Verfolgung		
REST		
Security		

Tabelle 5-1: Makroarchitektur im Vergleich

Die Hypoport AG hat das Continuous Deployment als erste Richtlinie festgelegt, um die Anwendung direkt nach einem Build in Produktion zu bringen (Freudl-Gierke, 2014). Um das explorative Testen der Services zu unterstützen, wird ein einheitliches DNS-Namensschema definiert. Die Logausgaben der Services werden zentral mit dem Tool *Splunk* aggregiert. Das Monitoring erfolgt mit dem System *Icinga* und die Metriken werden mit Graphite dargestellt. Für die Request-Verfolgung werden Trace-IDs generiert, die nachfolgend an andere Requests weitergereicht werden. Die IDs sollen vorzugsweise bereits als *HTTP-Header X-Trace-ID* im Ajax Call enthalten sein. Die Kommunikation erfolgt auf Basis von REST und HAL. Damit die Services unabhängig von den Konsumenten ausgerollt werden können, müssen die Schnittstellen abwärtskompatibel sein. Außerdem gibt es eine Security-Richtlinie, z.B. für *Auth Token*.

Auch Otto.de hat vier Richtlinien für die Makroarchitektur definiert (Steinacker, 2015). Das System wird in Vertikale geschnitten, deren Verantwortung bei den Teams liegt. Die Kommunikation zwischen den Vertikalen erfolgt über REST. Sie soll ausschließlich im Hintergrund stattfinden und nicht während der Ausführung eines User Requests (Abschnitt 5.4). Es darf keinen gemeinsamen Zustand geben, über den sich die Services austauschen. Für Daten muss es außerdem immer ein führendes System geben. Werden in einem anderen System Daten benötigt, können diese über eine REST-Schnittstelle gelesen und in der eigenen Datenbank redundant gehalten werden.

Zalando nennt ihre Richtlinien für die Makroarchitektur *Rules of Play* (Apple, 2016). Die erste Richtlinie bezieht sich auf die allgemeine Umsetzung von Applikationen als Microservices. Zalando verfolgt außerdem den *API First*-Ansatz. Die Entwicklung beginnt dabei mit der API-Definition, welche anschließend einer Peer Review unterzogen wird (Schäfer, 2016). Die Services werden auf Amazon AWS bereitgestellt und sollen von den Teams so entwickelt

werden, als würden sie die Software als *SaaS*-Lösungen (*Software as a Service*) Drittanbietern bereitstellen. Das bedeutet bspw., dass es wohldefinierte Schnittstellen, eine gute Dokumentation und *Service Level Agreements (SLA)* gibt. Wie die Hypoport AG und Otto.de verwendet Zalando REST zur Kommunikation.

Infrastruktur

Zalando verwendet für das Deployment Docker Container (Schäfer, 2016). Das Monitoring erfolgt über das selbstentwickelte Tool *Zmon*. Als Plattform nutzt Zalando das hauseigene *stubs.io,* welches auf Amazon AWS aufsetzt. Stubs.io regelt den autonomen Zugriff der Teams auf AWS und sorgt für ein Setup nach rechtlichen Vorgaben. Customer werden von dem eigenen HTTP-Router *Skipper* zur Legacy-Anwendung oder zum Layout Service weitergeleitet. Der Layout Service *Inceeper* dient der frontendseitigen Integration der Microservices. Dieser setzt die Webseite aus Fragmenten zusammen (z.B. Header, Produkt und Footer). Die Fragmente werden von Webapplikationen bedient, welche in der Verantwortung einzelner Teams liegen und besitzen ggf. eine eigene Datenhaltung (Abbildung 5-2).

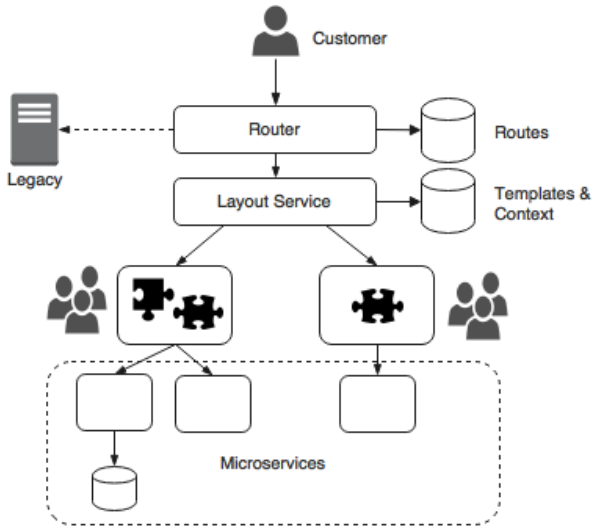

Abbildung 5-2: Zalando-Architektur, angelehnt an Schäfer (2016)

Spotify beschreibt ihr System mit einem *Core Data Model* (Abbildung 5-3). Jedes System kann aus mehreren Komponenten bestehen. Komponenten können z.B. Microservices, Data Stores oder Libraries sein und gehören zu genau einem System. Eine *Squad* repräsentiert ein Team bei Spotify und kann für mehrere Komponenten verantwortlich sein. Eine Komponente kann mehrere Services aufrufen und somit von mehreren *Discovery Names* abhängen. Sie kann selbst mehrere Discovery Names registrieren, über die sie erreichbar ist. Spotify nutzt außerdem *View Aggregation Services*, um Aufrufe der Customer zu kapseln (Goldsmith, 2015). Bei einer Client-Anfrage werden mehrere Services angestoßen, welche Resultate zurück liefern, die für den Nutzer nicht relevant sind. Die View Aggregation Services liefern dem Aufrufer ausschließlich die Antwort zurück, die er benötigt. Somit kann die Latenzzeit erheblich reduziert werden.

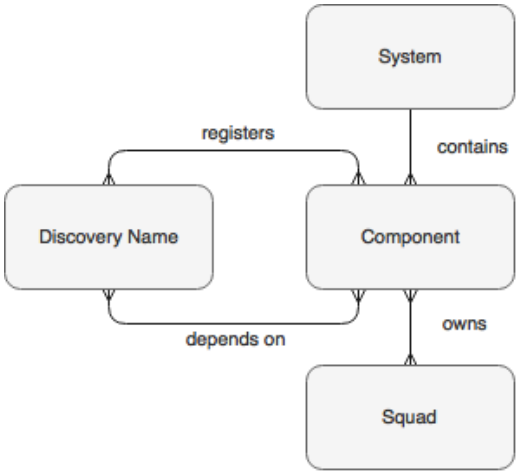

Abbildung 5-3: Spotify Core Data Model (Linders, 2015)

Auch die Taxi-Plattform Hailo benötigt speziell für die Kundendaten eine Möglichkeit, um die Latenzzeiten zu verringern. Alle Daten müssen auf drei Rechenzentren gleichzeitig verfügbar sein, weshalb sich Hailo bewusst für eine Cassandra-Datenbank mit einem eventually consistent Data Store entschieden hat. Die Kernfunktionalität der Kunden-API besteht aus mehreren zustandslosen HTTP-basierten Services, welche mit Unterstützung von Cassandra in allen drei Regionen laufen. Um die Kosten gering zu halten, einfach skalieren und schnell auf Marktänderungen reagieren zu können, entschied sich Hailo für Amazon Services als Cloud-basierte Infrastruktur. Zur Kommunikation werden *Protocol Buffer* von Google verwendet, mit deren Hilfe streng typisierte Nachrichten über einen *RabbitQM message bus* zwischen den in Java oder Go geschriebenen Services ausgetauscht werden können (Abbildung 5-4). Aufgrund der Erweiterbarkeit der Protocol Buffer, können diese während der Laufzeit der Services geändert werden und weiterhin ältere Versionen der Clients unterstützen. Die Services selbst basieren auf einer *Platform Layer-Bibliothek*, welche den Message Bus Transport Layer abstrahiert und Nachrichten mit registrierten Message Handlern innerhalb der Services austauscht. Hierüber wird ebenfalls das Framework für die Interprozesskommunikation, Service Discovery, Monitoring, Authentifizierung und Autorisierung sowie A/B-Testing zur Verfügung gestellt. Zusätzlich gibt es *Service Layer-Bibliotheken,* welche die Abstraktionsschichten der meisten Drittanbieterservices, wie z.B. Cassandra, bereitstellen.

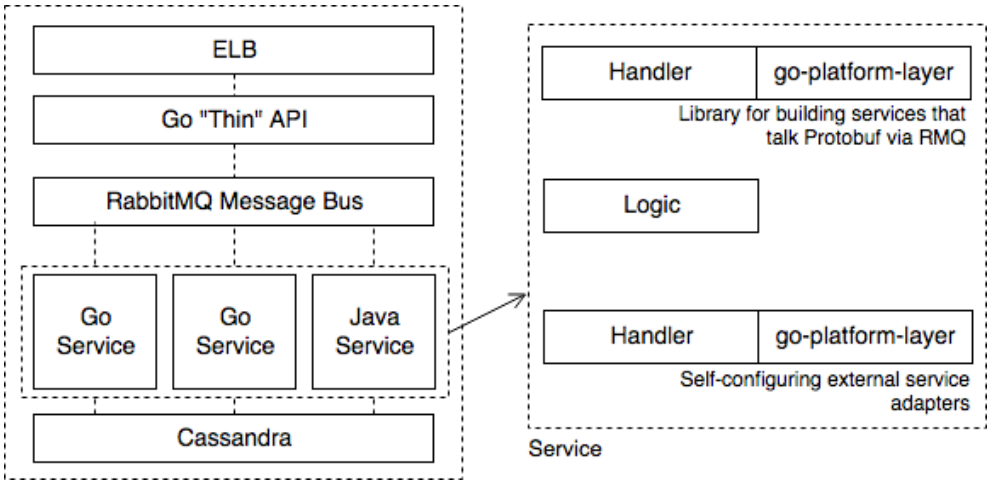

Abbildung 5-4: Die Hailo-Instrastruktur, angelehnt an Heath (2015)

5.3 Fachliche Architektur

Um einen Gesamtüberblick über die Architektur zu erhalten, hat Netflix ihr Monitoring Tool Atlas mit weiteren Tools kombiniert. Mit dem internen Dienstprogramm *Slalom* können z.B. alle Services betrachtet werden, die im Zusammenspiel eine bestimmte Anforderung erfüllen (Watson, Emmons, & Gregg, 2015). Es wird außerdem *Mogul* verwendet, um Performanceverschlechterungen feststellen zu können. Mogul sammelt Metriken von Atlas, reduziert die Daten über einen Filter und korreliert die Ergebnisse der wichtigsten Metriken. Ein weiteres Tool ist *Vizceral*, auf dessen Basis Entscheidungen intuitiv getroffen werden sollen (Abbildung 5-5). Netflix nennt diesen Ansatz *Intuition Engineering* (Reynolds & Rosenthal, 2016). Das Tool verwendet eine Laufzeitvisualisierung, die den Traffic unterschiedlicher Regionen darstellt. Hierfür sammelt Netflix mit einem serverseitigen Dienst Daten von Atlas und dem internen Request Tracing Service *Salp*. Anschließend wandelt der Dienst die Daten in das für die Vizceral-Komponente benötigte Format um und aktualisiert die Benutzeroberfläche über WebSockets. Gesammelt werden Echtzeitdaten, Informationen über Volumen, Latenzzeit und Health-Status der Requests, IPC-Informationen sowie Abhängigkeiten unter den Services (Kosewski, Tatelman, Reynolds, & Rosenthal, 2015). Neben der globalen View, die den Traffic aller Netflix-Regionen darstellt, gibt es eine Regional View mit einer Übersicht der sich in Betrieb befindenden Microservices und deren Abhängigkeiten. Fährt man auf einen Knoten, können die eingehenden und ausgehenden Verbindungen hervorgehoben werden.

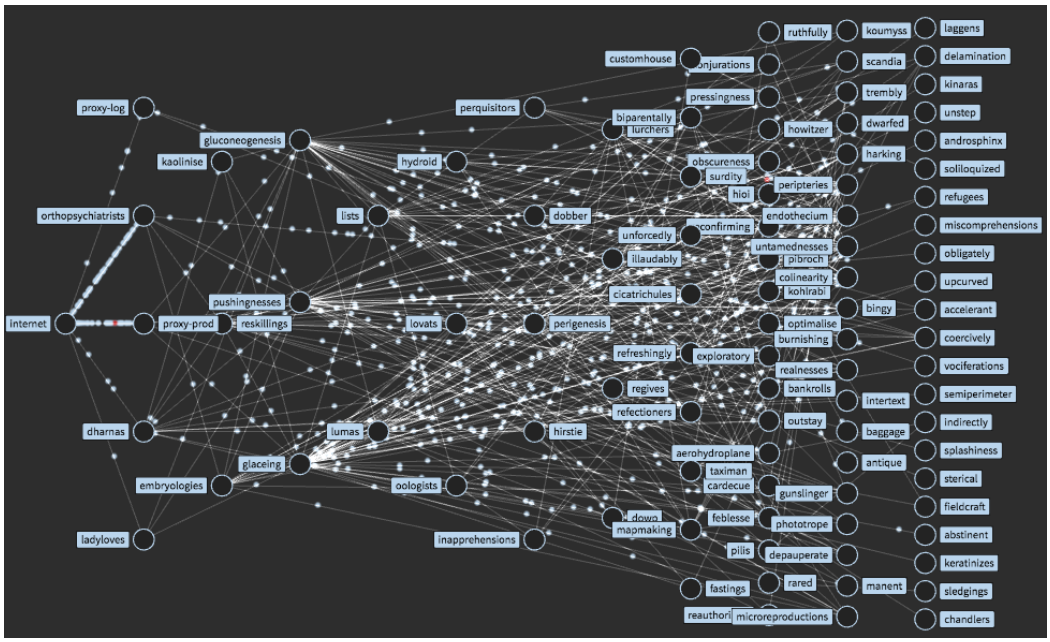

Abbildung 5-5: Netflix Vizceral (Reynolds & Rosenthal, 2016)

Zalando sammelt ebenfalls Metadaten, um einen Überblick über ihre Architektur zu erhalten (Wehrens, 2016). Dabei wird *TWINTIP* als API Discovery verwendet. In Verbindung mit *Kio,* einer Application Registry, durchsucht TWINTIP die Applikationen und generiert eine Liste mit allen Applikationen und ihren Serviceendpunkten und holt sich daraus ihre Open API-Spezifikationen (Müller F. , 2015). Im Hintergrund agiert *TWINTIP Storage* als Storage Engine und kann für weitere Integrationszwecke verwendet werden.

Als Toolunterstützung der Enterprise-Architektur kommt bei Zalando *LeanIX* zum Einsatz. Das Tool bietet Transparenz über alle Applikationen und deren Abhängigkeiten. So kann bspw. eine Volltextsuche über alle technischen Daten durchgeführt werden, um Services mit gemeinsamen Datenobjekten herauszufiltern.

Die E-Post hat ebenfalls aufgrund fehlender Open Source-Alternativen ein eigenes Tool für die Datensammlung namens *pivio.io* entwickelt. Das Tool stellt generelle Informationen zu den Services, Laufzeitinformationen sowie Informationen zu Schnittstellen, Ports, Protokollen und Abhängigkeiten bereit. Hierzu sammelt es alle Daten, welche die Services über *YAML* bereitstellen, legt diese an einer zentralen Stelle ab und macht sie durchsuchbar.

Auch Spotify hat mit *System-Z* ein eigenes Tool entwickelt, welches Daten zur Laufzeit sammelt (Karcher, 2016). Die Metadaten werden ebenfalls mit YAML-Dateien von den Services bereitgestellt. Über die *searchview* von System-Z können anschließend u.a. generelle Informationen, Abhängigkeiten und Deployment-Informationen abgerufen werden.

Hailo wiederum verwendet eingebaute Health-Checks, um Metriken auf Serviceebene darzustellen (Heath, 2015). Jeder Service sendet hierfür mittels *Public/Subscribe* eine Anzahl an Health-Checks über den *RabbitMQ* Message Bus. Anschließend werden Informationen darüber gesammelt, ob die Servicekonfigurationen erfolgreich geladen wurden oder ein Service genügend Kapazitäten hat, um sein aktuelles Anfragevolumen zu beantworten.

5.4 Legacy-Systeme

SoundCloud begann aufgrund von Skalierungsproblemen Microservices in ihre monolithische Architektur, basierend auf *Ruby on Rails*, zu integrieren (Calçado, 2014). Der erste Ansatz dabei war, die Microservices über dieselbe Public API mit dem Monolithen kommunizieren zu lassen, welche von den Usern und Drittanbietern genutzt wird. Die Microservices sollten nur neue Funktionalitäten anbieten. Sobald ein neuer Service Berührungspunkte mit dem Monolithen hatte, sollte dieser aus dem Monolithen gelöst und somit langsam aufgeteilt werden. Dieser Ansatz war jedoch unzureichend, da Microservices auf User-Aktivitäten reagieren mussten. Aus diesem Grund wurde ein *Semantic Events Model* eingeführt. Änderungen in den Domänenobjekten resultieren in einer Nachricht, welche an einen Message Broker gesendet wird. Anschließend kann die Nachricht von den Microservices konsumiert werden, die diese verarbeiten möchten. Die Architektur ermöglicht Event Sourcing, welche viele Services für eine geteilte Datenhaltung verwenden. Dennoch musste weiterhin die Public API angefragt werden, bspw., wenn E-Mail-Adressen bestimmter Nutzer für einen Nachrichtenversand benötigt wurden. Um dieses Problem zu lösen, verwendet das Team die Features von Rails *Engines*, um eine interne API zu erstellen, welche nur im internen privaten Netzwerk erreichbar ist. Zum Schutz der internen Zugriffe wird das OAuth-Protokoll verwendet, mit dem je nach Microservice bestimmte Ressourcen freigegeben werden. Abbildung 5-6 zeigt die Integration in die Legacy-Applikation von SoundCloud, welche als *The Mothership* bezeichnet wird.

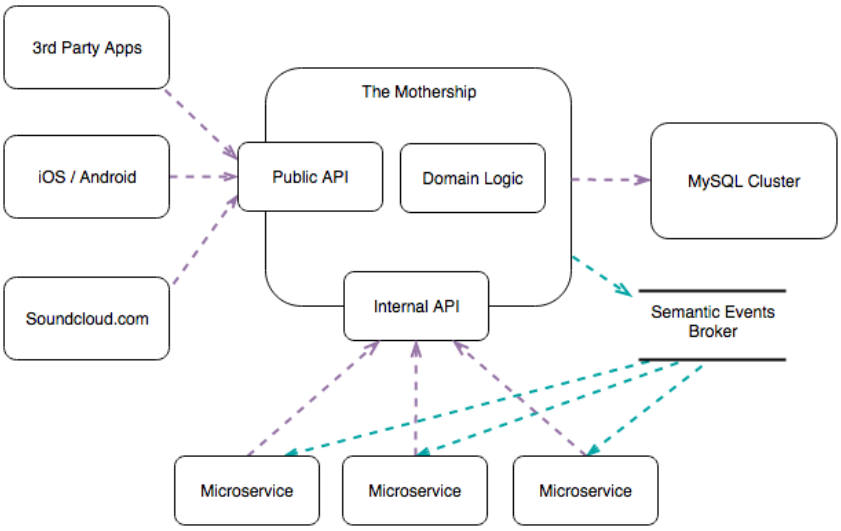

Abbildung 5-6: SoundCloud Legacy-Integration (Calçado, 2014)

Kühne + Nagel wählten den Ansatz der Self-Contained Systems, um ihre schwergewichtige Applikation *KN Login* langsam abzulösen (Annenko, 2016). KN Login basiert auf einem Java-Webframework mit über 1.5 Millionen Codezeilen. Wie bei SoundCloud wurden keine neuen Features mehr in dem Monolithen integriert. Als Prototyp für ein SCS wurde *KN FreightNet* gewählt, eine Logistik-E-Commerce-Plattform, bei der Kunden gezielt Angebote anfordern und Buchungen durchführen konnten. Der Angebotsteil war bereits in KN Login integriert und konnte somit nach einer erfolgreichen Implementierung des SCS abgelöst werden. Für weitere Funktionalitäten kamen neue SCS-Einheiten, zusammen mit SCS für Querschnittsfunktionalitäten, hinzu.

Autoscout 24 und Otto.de nutzen ebenfalls Self-Contained Systems zur Aufteilung ihrer Architektur (Schlosser, 2016a; Steinacker, 2015). Bei Otto.de können nach Steinacker (2015) die Vertikale zur weiteren Zerlegung horizontal geschnitten werden. Die SCS können somit selbst aus Microservices bestehen, welche z.B. über REST miteinander kommunizieren. Requests werden von einem Service entgegengenommen und auf weitere Services aufgeteilt. Die Teilergebnisse werden schließlich aggregiert und an den Aufrufer zurückgesendet. Für die Datenreplikation verwendet Otto.de *Atom Feeds* anstelle von Message Queues. Hier fragen Services, welche bestimmte Daten benötigen, regelmäßig den Feed der führenden Vertikale ab. Laut Steinacker (2015) lässt sich diese Replikation in einigen Fällen vermeiden, wenn Services während eines User Requests synchron auf andere Vertikale zugreifen. Aufgrund der negativen Auswirkungen auf Testbarkeit, Skalierbarkeit, Verfügbarkeit und einem unabhängigen Deployment wird allerdings auf diese Variante verzichtet.

5.5 Verantwortlichkeiten

Müller J. (2016) konnte das Inverse Conways Law bei der praktischen Anwendung mit Microservices bei der Hypoport AG beobachten. Nach dem Konzept der Holokratie änderte sich die Organisation von der klassischen Unterteilung zwischen Entwicklern und Architekten hin zu verteilter Verantwortung von mehreren Personen in bestimmten Bereichen. Bei der E-Post

erfolgte ebenfalls eine Angleichung der Teams an die Microservice-Architektur (Wehrens & Gentsch, 2016). Dies führte neben Vorteilen in der Umsetzungsgeschwindigkeit und technischen Qualität dazu, dass die Teams eigeninitiativ ihre Technologie-Stacks vereinheitlichten. Auch bei Autoscout24 begannen die Microservice-Teams selbstständig die Vorgaben der Makroarchitektur an ihre Bedürfnisse anzupassen und eigenständig Lösungen zu entwickeln (Bryant, 2016). Die Verschiebung der Verantwortung Richtung Teams kann allerdings auch eine Gefahr sein. Martraire (2016) warnt davor, dass viele Teams nicht genügend Erfahrung besitzen, um Architekturentscheidungen alleine zu treffen. Er schlägt vor, dass Architekten die Teams vorerst schulen, bevor diese ihre Entscheidungen alleine treffen. Eine weitere Auswirkung auf die Organisation ist nach Müller J. (2016), die Vermischung von Entwicklung und Betrieb (DevOps). Werner Vogels, CTO von Amazon, konnte in diesem Zusammenhang beobachten, dass sich die Qualität der Services verbesserte, nachdem die Teams neben der Entwicklung auch für den Betrieb verantwortlich waren (Gray, 2006). Nach dem Motto „*You built it you run it*“ kamen die Entwickler täglich mit den Aufgaben des Betriebs und Benutzern in Kontakt, wobei letzteres maßgeblich zur Qualitätsverbesserung der Software beitrug. Für Stropek (2016) ist diese Vermischung eine Grundvorrausetzung, um Microservices erfolgreich einzusetzen. Dabei empfiehlt er speziell die Verwendung von Cloud-Technologien, um schnell eine robuste Infrastruktur verwenden zu können. Hier könnte ein Vorteil sein, dass beim Einsatz von Cloud-Technologien weniger Expertise erforderlich ist, als beim Aufbau einer eigenen Infrastruktur. Gerade das Finden von Entwicklern mit einem starken DevOps-Profil ist nach Wootton (2014) eine große Herausforderung bei Microservices.

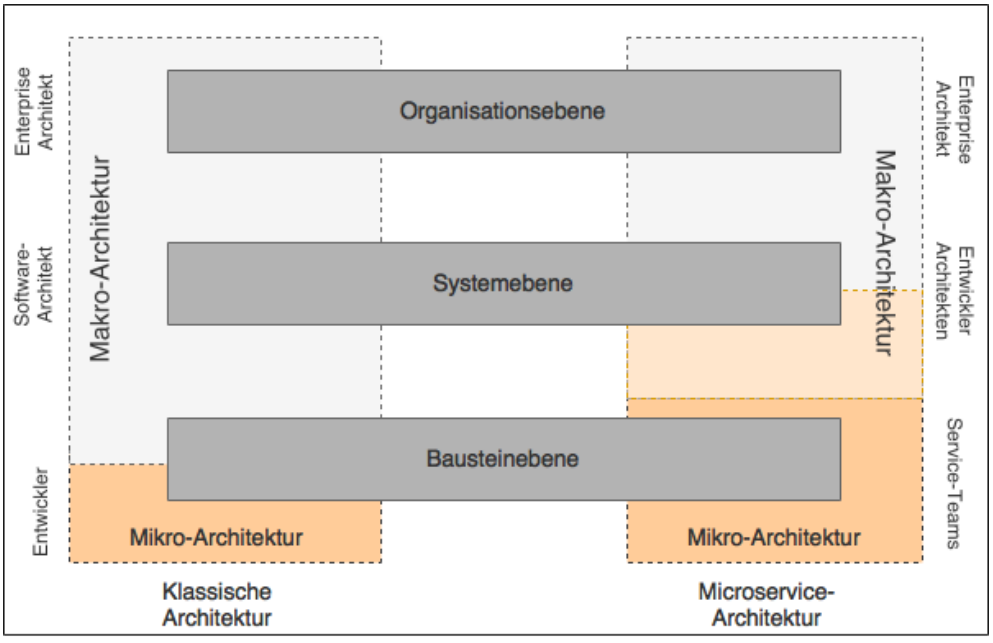

Abbildung 5-7: Verschiebung der Verantwortlichkeiten, angelehnt an Vogel, et al. (2005)

Abbildung 5-7 zeigt eine mögliche Verschiebung der Verantwortlichkeiten bei der Entwicklung von Microservices anhand der Ebenen aus Abschnitt 2.1.4. Während bei monolithischen Architekturen oft noch der Softwarearchitekt für alle tragenden Bausteine innerhalb der gesamten Applikation zuständig ist, sind in einer Microservice-Architektur die Serviceteams sowohl für die tragenden als auch für die nicht-tragenden Bausteine der Services

verantwortlich. Die Erfahrungen der Hypoport AG, E-Post und Autoscout24 deuten außerdem darauf hin, dass die Serviceteams auch auf der Systemebene mitentscheiden dürfen. Müller J. (2016) ergänzt hierzu, dass seine beobachteten Veränderungen mit der Einführung von Microservices einhergingen, aber nicht zwingend darauf zurückgeführt werden müssen. Vielmehr können diese Teil eines Trends sein, der sich mit der steigenden Release-Geschwindigkeit der Industrie abgezeichnet hat und durch Microservices verstärkt wurde.

6 EMPIRISCHE ANALYSE

Anhand einer empirischen Analyse sollen einige Annahmen basierend auf dem theoretischen Teil überprüft und einige der gesammelten Informationen um Wissen aus der Praxis ergänzt werden. Nachfolgend werden Aufbau und Vorgehen sowie Zielgruppe und Ergebnisse dieser Analyse vorgestellt.

6.1 Befragung

Als Erhebungsmethode wurde eine standardisierte Befragung im Form einer Online-Umfrage gewählt. Die komplette Umfrage befindet sich in Anhang A.

6.1.1 Aufbau und Zielgruppe

Die Fragen und Reihenfolge der Befragung waren fest vorgegeben. Es gab insgesamt 13 Fragen, davon acht mit vordefinierten Antwortmöglichkeiten und fünf Freitextfragen. Die Fragen wurden in drei Teile gegliedert: Allgemeine Informationen zum Unternehmen, Makro- und Mikroarchitektur und fachliche Architektur. Die Datenerhebung erfolgte anonym. Es wurden keine persönlichen Daten zum Unternehmen oder Personen erhoben. Aus diesem Grund beschränkten sich die allgemeinen Informationen auf eine Frage zur Unternehmensgröße (Abbildung 6-1).

Für die zwei folgenden Fragen wurden vordefinierte Auswahlmöglichkeiten anhand von Checkboxen erstellt:

- Welche Aufgaben werden der Makroarchitektur zugeordnet?
- Welche Daten werden gesammelt?

Die vordefinierten Antworten der ersten Frage stammen aus Tabelle 4-1. Diese wurden mit den Infrastrukturthemen Service Integration und Deployment aus dem theoretischen Teil ergänzt. Für die zweite Frage wurden die Informationen aus den Literaturquellen aus Abschnitt 5.3 herangezogen. Der Fragebogen soll folgende Annahmen verifizieren:

- Die Trennlinie zwischen Makro- und Mikroarchitektur ist nicht klar definiert.
- Teams wird mehr Verantwortung in der Mikroarchitektur zuteil. Inhalte der Makroarchitektur werden ggf. von Teams mitbestimmt.
- Es wird vorwiegend Domain-driven Design verwendet, um die Services fachlich aufzuteilen.

- Es werden Metriken gesammelt, um einen Überblick über die Gesamtarchitektur zu erhalten.

Außerdem sollen folgende Informationen ermittelt werden:

- Welche weiteren Regeln gelten für die Makroarchitektur?
- Welche weiteren Daten werden gesammelt?

Zur Zielgruppe gehörten Personen, die in einem Projekt mit einer Microservice-Architektur tätig waren oder aktuell tätig sind. Hierzu zählen die genannten Interessensvertreter aus Abschnitt 1.2 wie z.B. Softwareentwickler, Teamleiter, Tester und Architekten. Die Fragen wurden so gestellt, dass sie nicht aus Sicht einer bestimmten Rolle beantwortet werden mussten.

6.1.2 Vorgehen

Der Fragebogen wurde vor der Verteilung einem Pretest unterzogen und an eine Expertengruppe gesendet, um zu prüfen, ob er fachlich korrekt aufgebaut ist und ob es Spielraum für Verständnis- oder Interpretationsprobleme gibt. Anhand der Rückmeldungen konnte der Fragebogen optimiert werden. Neben Ergänzungen zu Beschreibungen wurden bspw. Pflichtfelder hinzugefügt oder Fragen gestrichen und abgeändert. Weiter wurde Kontakt mit dem Verfasser einer ähnlichen Studie aufgenommen (Aichinger, 2017), um das Vorgehen bei der Verbreitung zu optimieren. Als Ergebnis wurde überwiegend über die sozialen Netzwerke *Xing* und *LinkedIn* direkt Kontakt mit Personen aufgenommen, welche angaben Erfahrungen mit Microservice-Projekten zu haben. Die Umfrage wurde mit Google Forms verfasst, ein kostenfreies Online Tool, das es ermöglicht, verschiedene Fragetypen einzustellen und die Ergebnisse direkt in eine Tabelle zu exportieren. Um eine regionale Einschränkung zu vermeiden, wurde die Umfrage in Deutsch und Englisch verfasst. An der Befragung nahmen insgesamt 44 Personen teil.

6.1.3 Ergebnisse

Die Auswertung der Ergebnisse erfolgte in Excel und wurde in zwei Teilen durchgeführt. Als erstes wurden die Antworten der vordefinierten Fragen summiert und in Diagrammen dargestellt. Anschließend wurden für drei Freitextfragen homogene Gruppen (Cluster) definiert und passende Antworten oder Teile der Antworten diesen Gruppen zugeordnet. Dies förderte die Übersichtlichkeit und es konnten Tendenzen aus den freiformulierten Antworten gewonnen werden. Die weiteren Freitextantworten waren so unterschiedlich, dass sie nicht grafisch dargestellt, sondern nachfolgend vollständig erläutert werden. Die Daten wurden außerdem bereinigt, nicht auswertbare Daten wurden entfernt.

Größe des Unternehmens

Von den 44 Teilnehmern gehört die Mehrheit (34%) großen Unternehmen mit mehr als 500 Mitarbeitern an (Abbildung 6-1). An zweiter Stelle kamen mit 29% Unternehmen mit 51-500 Mitarbeitern. An dritter Stelle Unternehmen mit weniger als 20 Mitarbeitern (23%). Die

wenigsten Teilnehmer kamen aus Unternehmen mit einer Größe von 21-50 Mitarbeitern mit 14%.

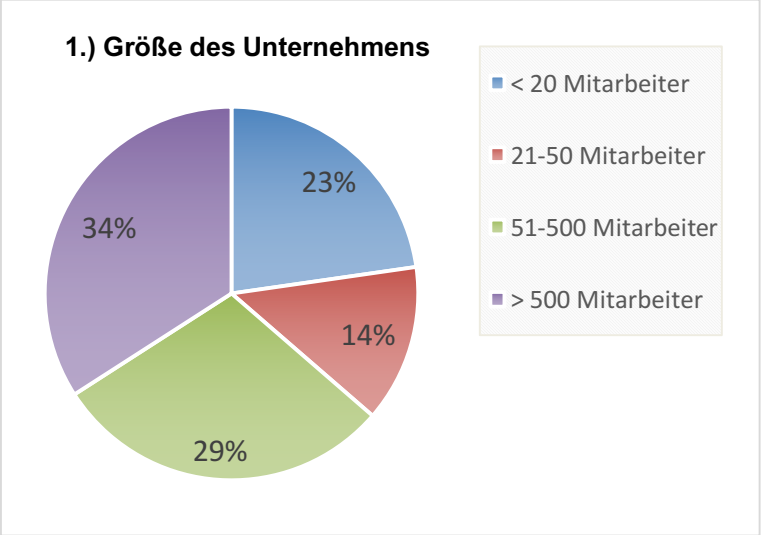

Abbildung 6-1: Unternehmensgröße der Teilnehmer

Definierte Makroarchitektur

39 Teilnehmer gaben an, eine Makroarchitektur zu definieren (Abbildung 6-2). Lediglich fünf hatten in ihrem Unternehmen keine definierte Makroarchitektur. Bei diesen Unternehmen handelte es sich um drei Unternehmen mit einer Größe von 51-500 Mitarbeiter, eines mit einer Größe von 21-50 Mitarbeiten und eines mit mehr als 500 Mitarbeitern.

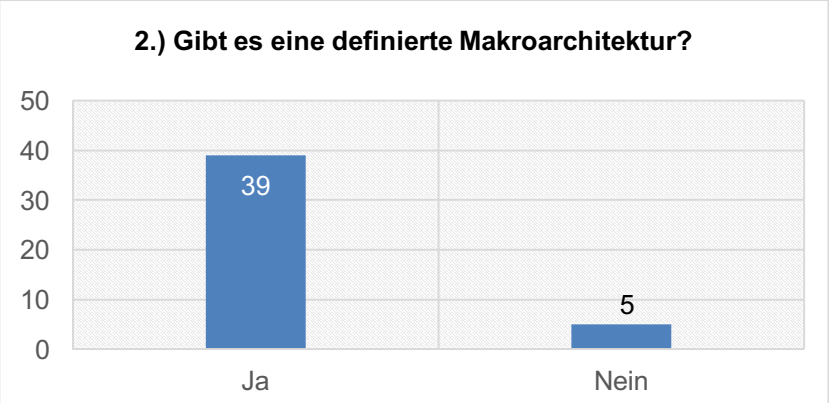

Abbildung 6-2: Definierte Makroarchitektur

Durchsetzung der Makroarchitektur

Bei der Frage nach der Durchsetzung der Makroarchitektur wurden Gruppen definiert (Abbildung 6-3). Dabei konnte eine Antwort zu mehreren Gruppen gehören, wenn sich bspw. die Durchsetzung der Makroarchitektur auf verschiedene Bereiche und Rollen erstreckte. Zwölf Teilnehmer gaben an, eine Form von Dokumentation zu nutzen, um die Makroarchitektur transparent zu machen. Hierzu zählten allgemeine Richtlinien, Architekturrichtlinien oder eine Definition of Done.

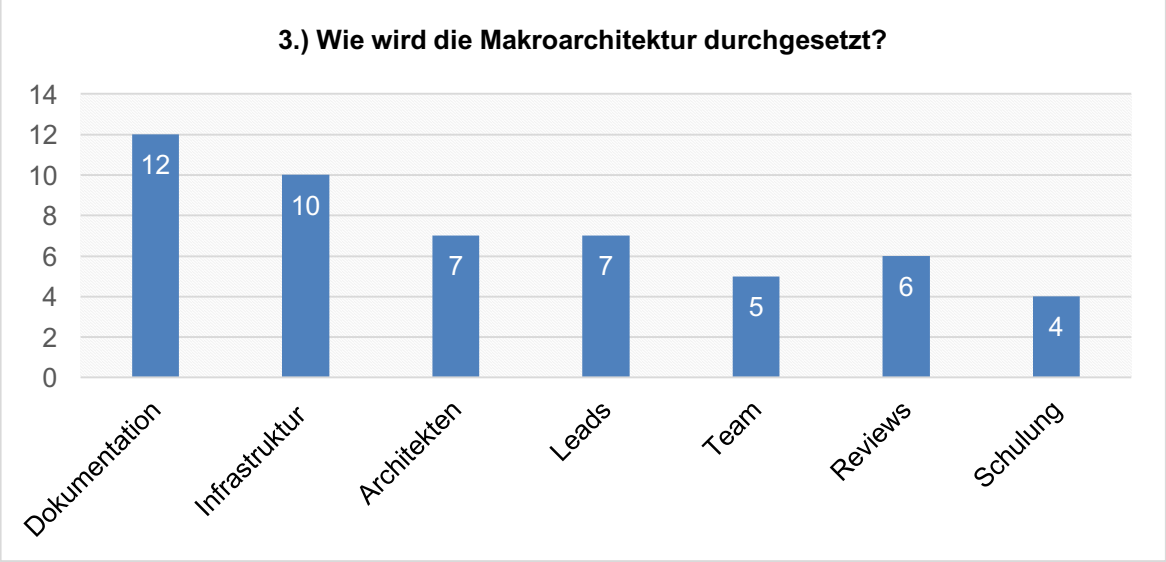

Abbildung 6-3: Durchsetzung der Makroarchitektur

Zehn Teilnehmer gaben an, dass die Vorgaben der Makroarchitektur über die Infrastruktur bereitgestellt oder validiert werden. Eine Person erklärte hierzu, dass die Vereinbarungen, ähnlich dem *conformity monkey* von Netflix, automatisch über die Infrastruktur überprüft werden. Außerdem wurden den Entwicklern Templates, Blueprints und Bibliotheken zur Verfügung gestellt, um sich an die Vorgaben zu halten. Konkrete Infrastrukturkomponenten wie Deployment, Monitoring, Logging oder eine Build Pipeline wurden ebenfalls genannt. Bei sieben Teilnehmern waren Architekten bei den Entscheidungen bezüglich der Makroarchitektur involviert. Hier gab es zum einen Architekturteams- oder Gruppen und zum anderen einzelne Architekten oder Enterprise-Architekten, welche die Verantwortung übernahmen. Hinzu kamen bei ebenfalls sieben Teilnehmern andere Rollen, die bei der Entscheidung mitwirkten. So gab es bspw. einen Head of Engineering, eine Geschäftsleitung, einen Software Platform Director, zwei Team Leader, einen CTO und ein getrenntes Dev-Ops-Team, welche über die Makroarchitektur entschieden oder mitbestimmten. Bei fünf Teilnehmern war zusätzlich das gesamte Team involviert. Lediglich bei einem dieser Teilnehmer trug das Team die alleinige Verantwortung. Sechs der Teilnehmer gaben an, Reviews durchzuführen, um die Vorgaben der Makroarchitektur zu validieren. Hier wurde ausschließlich die Peer Review als eine konkrete Review-Methode genannt. Letztendlich gaben noch vier der Teilnehmer an, über Schulung oder Coaching, die Teammitglieder an die Vorgaben der Makroarchitektur heranzuführen.

Aufgaben der Makroarchitektur

In Abbildung 6-4 wird dargestellt, welche Aufgaben die Teilnehmer ihrer Makroarchitektur zuordnen. Bei über 70% der Teilnehmer zählten Service Discovery, Authentifizierung, Monitoring und Logging zu den Aufgaben. Über die Hälfte zählten außerdem das Deployment, Kommunikationsprotokolle und die Autorisierung dazu. Unter Sonstige gaben 13 Teilnehmer folgende weitere Aufgaben an:

- AWS Account-Struktur, Deployment-Prinzipien und -Implementierung
- Persistierung (Couchbase)

- Technical Capability Map
- Containerplattform
- Caching, Errorhandling
- Teilweise Autorisierung und Datenmanagement (aufgrund Autorisierung)
- Artefact Management, Provisionierung, Health-Check
- Keine Business-Logik in der Makroarchitektur
- Datenflussparadigma

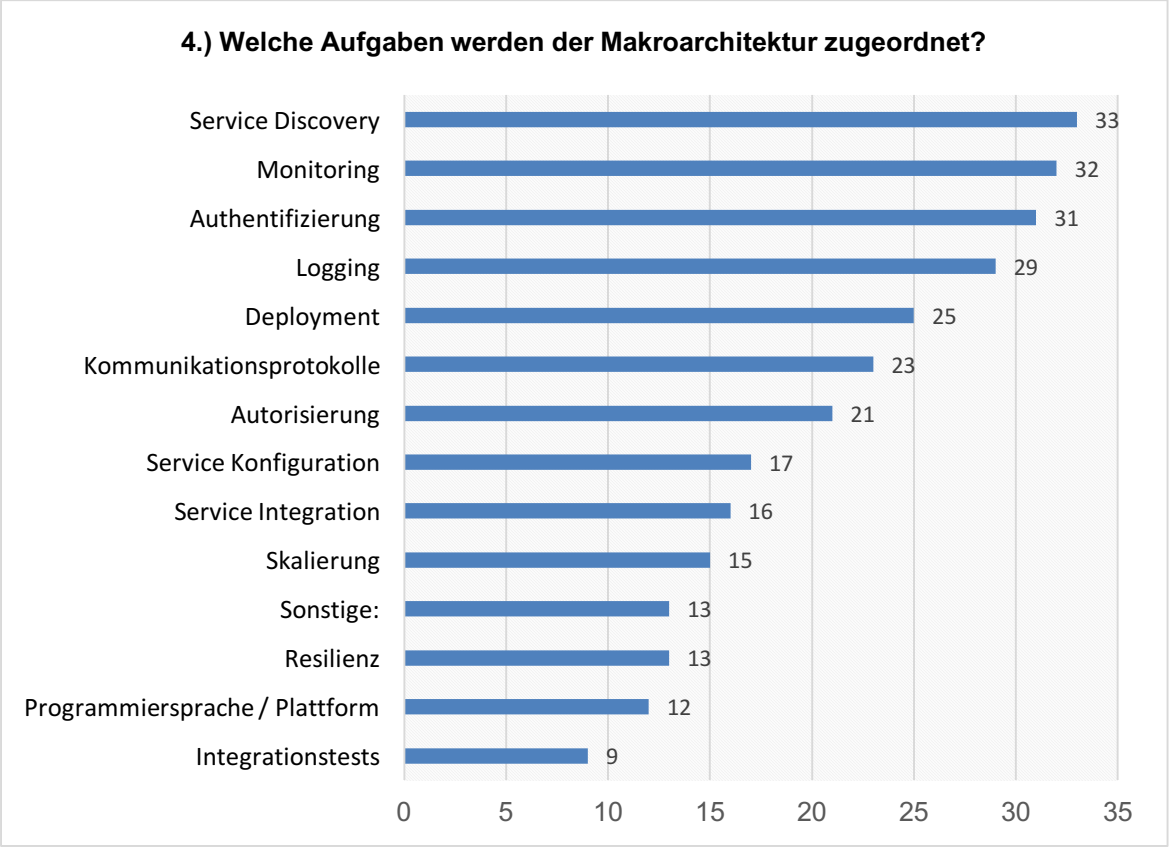

Abbildung 6-4: Aufgaben der Makroarchitektur

Deployment-Prinzipien und -Implementierung sowie Containerplattform lassen sich ebenfalls zum Deployment zuordnen. Health-Checks können Bestandteil einer Service Discovery sein, werden hier aber als einzelner Punkt angeführt. Das Datenmanagement kommt als weiterer Punkt in Frage, da sich Microservices für spezielle Aufgaben, hier die Autorisierung, gemeinsame Daten teilen können. Für die konkrete Persistierungstechnologie kann ebenfalls eine Zuordnung in der Makroarchitektur sinnvoll sein. Die Technologieunabhängigkeit ist somit zwar nicht mehr gegeben, aber ggf. können sich daraus andere Vorteile wie eine einheitliche Performance oder ein einfacher Wissenstransfer ergeben. Mit dem Artifact Management können verschiedene Abhängigkeiten bspw. in einer Continous Delivery Pipeline aufgelöst werden. Die Provisionierung adressiert die Bereitstellung von weiteren Ressourcen wie weitere Server. Auch das Error Handling und Caching sind weitere mögliche Aufgaben. Eine technische Capability Map, der Verzicht auf Businesslogik in der Makroarchitektur oder ein

Datenflussparadigma können als zusätzliche Regeln aufgefasst werden. Bei der Befragung wurden die Aufgaben von den allgemeinen Regeln getrennt, um eine Tendenz aus den zentralen Aufgaben ermitteln zu können.

Regeln der Makroarchitektur

31 Teilnehmer haben keine zusätzlichen Regeln für ihre Makroarchitektur definiert (Abbildung 6-5). 13 hingegen haben diese mit folgenden Regeln ergänzt:

- API-Richtlinien
- Dokumentation
- Orientierung an Bestandskunden und zukünftigen Kunden
- Einschränkung der Programmiersprachen (Java, GO)
- Einhaltung des Budgets
- Frameworks, Kommunikationsprotokolle, Datenformate
- Regeln über die Health-Checks
- Developer-Guide, Architektur-Board
- Höhere Testabdeckung, mehr Audits, härtere Commit-Strategien
- Services dürfen nur asynchron, d.h. nachrichtengetrieben kommunizieren

Außerdem wurden weitere Regeln ausführlicher beantwortet. Ein Teilnehmer gab an, dass die Teams die Verantwortung haben, die Grenzen der Mikroarchitektur zu kennen und potentielle Makroarchitekturthemen in die verantwortliche Gruppe mit einfließen zu lassen. Ein weiterer erklärte, dass alle Services in Containern laufen müssen und jeder Service nur maximal eine Datenbank bedienen darf. Für diese muss der Service schließlich alleine verantwortlich sein. Auch merkte ein Teilnehmer an, dass sich grundsätzlich alle Neuentwicklungen auf die definierte Zielarchitektur einzahlen müssen.

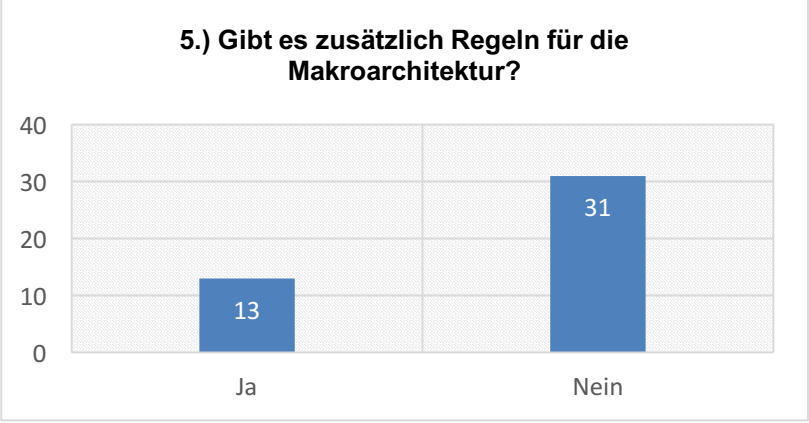

Abbildung 6-5: Regeln der Makroarchitektur

Entscheidungen auf Mikroebene

Für die Antworten bezüglich der Entscheidung auf Mikroebene wurden ebenfalls Gruppen gebildet. Aufgrund der sehr unterschiedlichen Antworten konnten allerdings nur wenige Teile zusammengefasst werden. Die größte Gruppe bilden Programmiersprache und Tools, die in 14 Antworten vorkamen. Danach folgen Datenbanken sowie Frameworks und Bibliotheken mit zehn und sechs Vorkommnissen. Vier Teilnehmer gaben an, dass alles, was in der

Makroarchitektur nicht definiert wurde, in der Verantwortung der Teams liegt. Abbildung 6-6 zeigt eine Übersicht aller Gruppen.

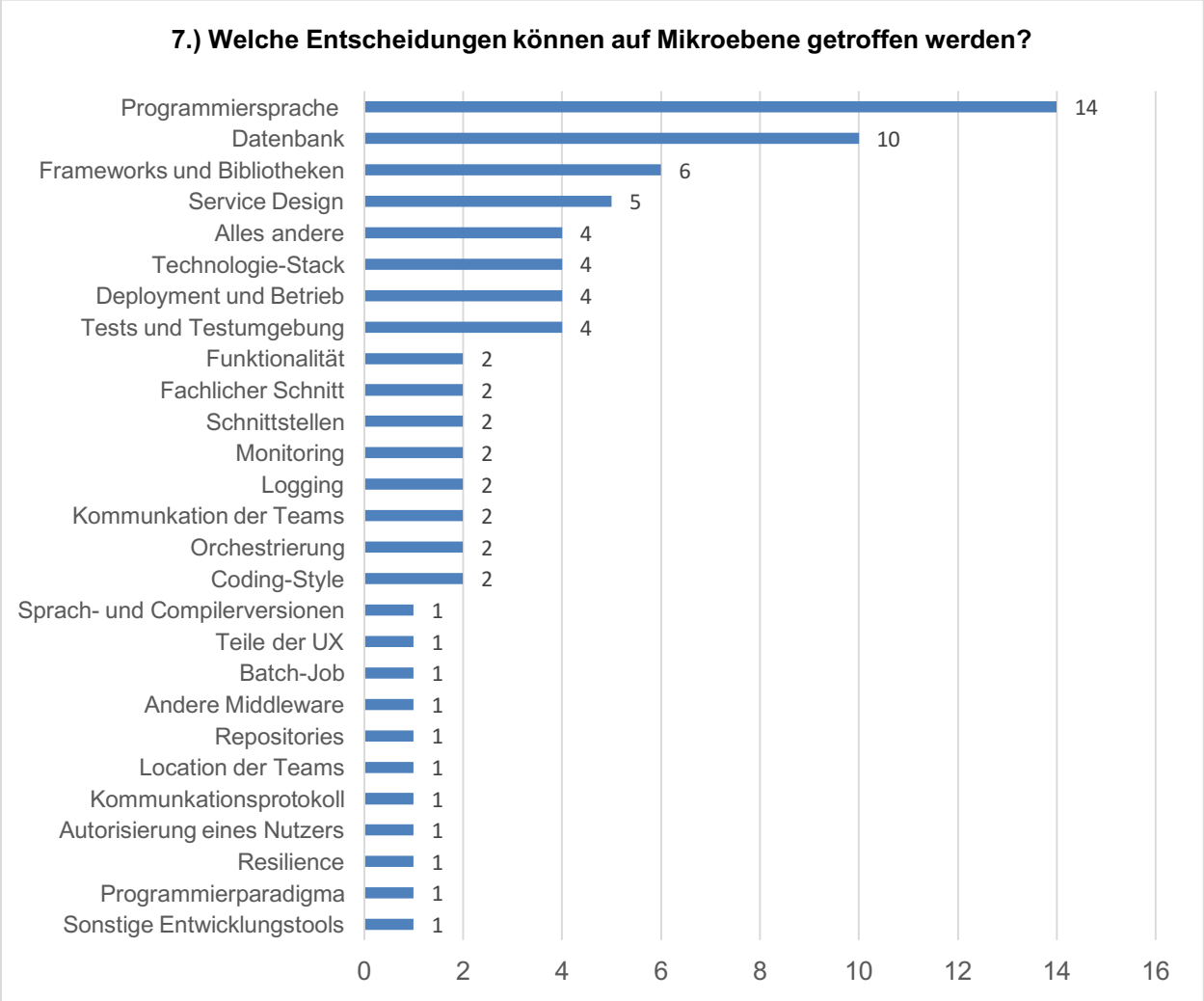

Abbildung 6-6: Entscheidungen der Mikroarchitektur

Überschneidung zwischen Mikro- und Makroarchitektur

Die Antworten bezüglich der Überschneidung gingen weit auseinander, weshalb sie hier vollständig aufgeführt werden:

- „Skalierung: Die Makroarchitektur definiert, wie skaliert wird und der Service selbst muss skalierbar sein."
- „Mit Tech Radar wird Guidance bzgl. Technologieentscheidungen in den Teams gegeben."
- „Die Makroarchitektur ist eine Empfehlung keine Vorschrift, damit überschneiden sich beide theoretisch in allen Bereichen."
- „Autorisierung: Rollen und Rechte müssen zwischen Makro- und Mikroarchitektur abgestimmt sein. Governance-Regeln der Makroarchitektur müssen/sollen eingehalten werden."
- „Deployment: Container auf Makroebene, Agents auf Mikroebene. Makroarchitektur kann teilweise nur eine Art „best practice"-Vorgabe sein, wobei es in der Mikroarchitektur

noch Möglichkeiten der Entscheidungsfreiheit gibt, um ggf. einen für den einzelnen Anwendungsfall besseren Weg zu finden."

- „Logging: Auf Makroebene einheitlich für serviceübergreifende Auswertungen, konkrete Umsetzung auf Mikroebene."
- „Makroarchitektur setzt sich aus Architekten der Mikroservice-Architekten zusammen."
- „UX, Fehlverhalten und Integration verschiedener anderer Microservices sind sowohl relevant für die Mikro- als auch für die Makroarchitektur. Dies geschieht oft bei der Einbindung/Nutzung zentral erstellter Bibliotheken oder bei der Integration der zentralen UX-Komponenten mit den teamspezifischen."
- „REST-API Modeling"

Informationen zum fachlichen Schnitt

Bezogen auf den fachlichen Schnitt gaben 41 Personen an, ihre Services nach einer bestimmten Vorgehensweise aufzuteilen (Abbildung 6-7). Lediglich drei hatten keine bestimmte Vorgehensweise.

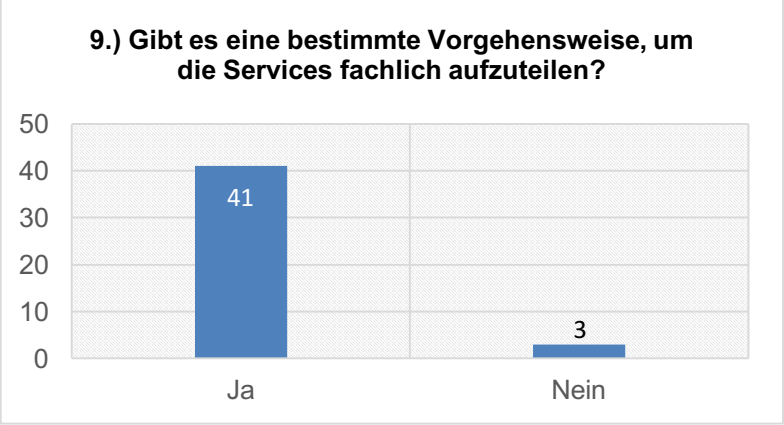

Abbildung 6-7: Vorgehensweise für die fachliche Aufteilung (Teil1)

36 der Teilnehmer verwenden Domain-driven Design für die Aufteilung der Services und fünf Teilnehmer eine andere Vorgehensweise (Abbildung 6-8).

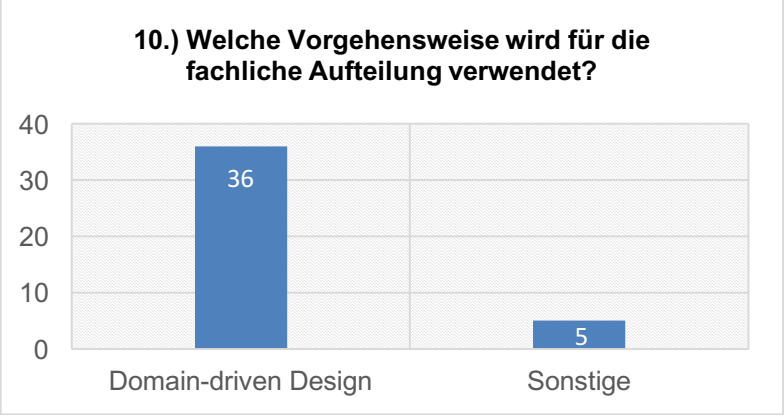

Abbildung 6-8: Vorgehensweise für die fachliche Aufteilung (Teil2)

Folgende weitere Vorgehensweisen wurden genannt:

- „Aufteilung zwar nach Bounded Context, aber auch Faktoren wie Skalierung und erwartete Lebenszeit eines Systems spielen eine Rolle."
- „Orientierung an SCS durch Migrationsstrategie von Legacy System nach Microservice Umgebung. Ausgangsbasis ist der Umzug ganzer Shop-Seiten hin zu Microservices. Allerdings findet dort, wo sinnvoll, eine zunehmende Granularisierung dieser großen Services in kleinere Bestandteile statt."
- „Webseiten werden in unabhängige Komponenten zerlegt, die möglichst exakt einen Anwendungsfall implementieren, welcher zugleich für verschiedene Webseiten wiederverwendet werden kann."
- „Jedem Service wird eine Abteilung zugeordnet, die wiederum eine Gruppe von Zielfunktionen umsetzt."
- „Fachliche Capability-Map"

Sammeln von Daten

Die Frage, ob Daten für ein Architekturmanagement gesammelt werden, und die darauf aufbauende Frage wurden teilweise anders ausgelegt. Einige Teilnehmer beantworteten den ersten Teil, ob Daten gesammelt werden mit nein und füllten dennoch den dritten Teil mit den konkreten Daten aus. Das kann so interpretiert werden, dass zwar Daten gesammelt werden, allerdings nicht, um diese für ein Architekturmanagement aufzubereiten. Somit wurden die Antworten in Abbildung 6-9 in drei Teile aufgeteilt. 23 Teilnehmer sammeln Daten für das Architekturmanagement, zwölf Teilnehmer sammeln keine Daten und neun Teilnehmer tun dies aus einem anderen Grund - höchstwahrscheinlich für ein klassisches Monitoring.

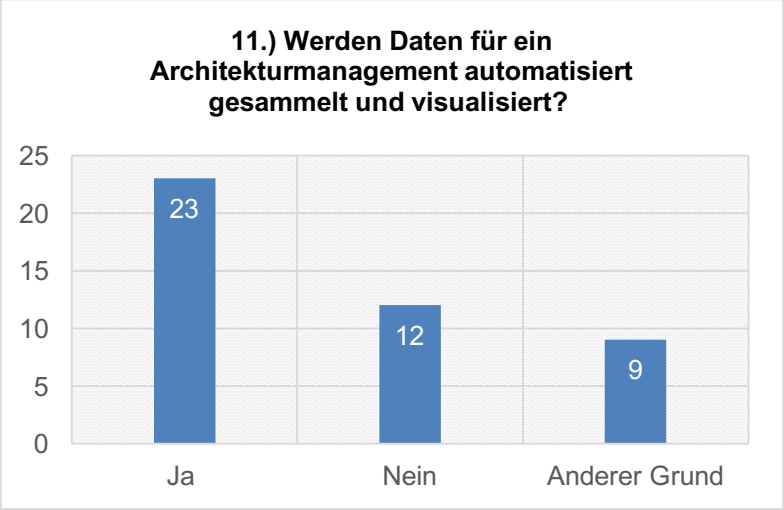

Abbildung 6-9: Sammeln von Daten für das Architekturmanagement (Teil 1)

Abbildung 6-10 zeigt, dass die Daten überwiegend über das Monitoring gesammelt werden. Hier wurden Technologien wie Grafana oder der ELK-Stack genannt. An zweiter Stelle steht die Service Discovery, mit der die Daten bspw. mit Consul zur Laufzeit gesammelt werden. Anschließend folgt das API Management bspw. mit Swagger. Zweimal wurden das Deployment z.B. über Deployment-Deskriptoren und Docker Container und das Durchsuchen der Repository Files genannt. Jeweils einmal genannt wurde eine zentrale Bibliothek, verteiltes Tracing mit

Zipkin, serviceübergreifende Modellierung mit UML und das Enterprise Datamanagement Tool *Goldensource*.

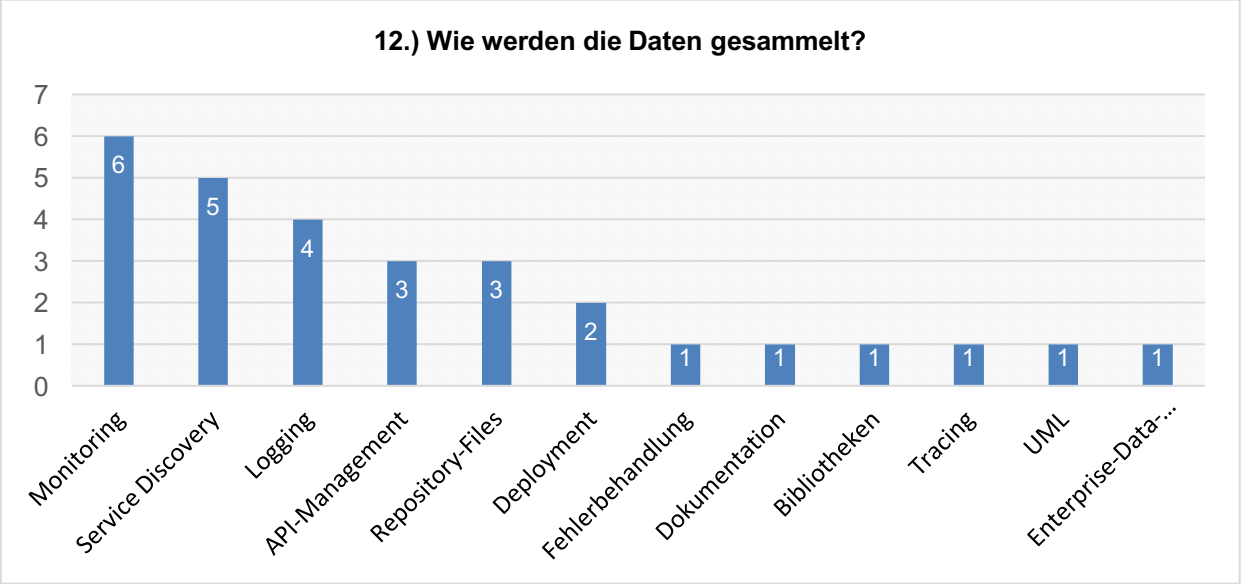

Abbildung 6-10: Sammeln von Daten für das Architekturmanagement (Teil 2)

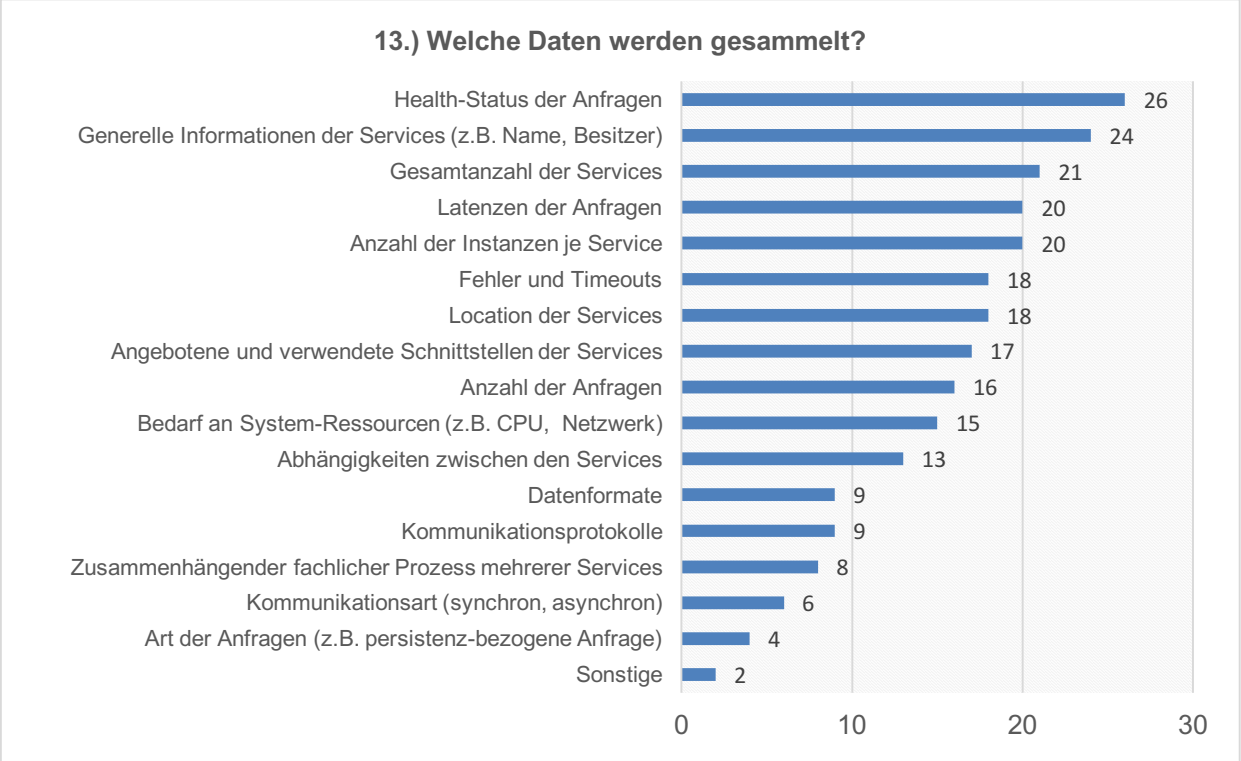

Abbildung 6-11: Sammeln von Daten für das Architekturmanagement (Teil 3)

Schließlich haben die Teilnehmer noch gewählt, welche Daten gesammelt werden (Abbildung 6-11). Die Ergebnisse zeigen, dass über 70 % der Teilnehmer den Health-Status der Anfragen und generelle Informationen über die Services sammeln. Über 60 % sammeln Latenzen der Anfragen, Gesamtzahl der Services, die Anzahl der Instanzen je Service sowie Informationen zu Fehlern und Timeouts. Mindestens die Hälfte sammeln außerdem die Location der Services, die Anzahl der Anfragen, den Bedarf an Systemressourcen als auch angebotene und

verwendete Schnittstellen der Services. Unter „Sonstige“ wurde noch Information Security (InfoSec) und Kritikalität angegeben.

6.2 Befragung mit vergleichbarem Charakter

Als Ergänzung zur eigenen Analyse wurde eine Umfrage aus der Masterarbeit von Aichinger (2017) herangezogen. Ziel dieser Arbeit war es, ein Dashboard für die Dokumentation dynamischer Softwaresysteme mit Schwerpunkt auf eine Microservice-Architektur zu entwickeln. Hierfür sollten Live-Daten aus unterschiedlichen Systemen herangezogen werden, um diese für unterschiedliche Stakeholder zu aggregieren und visualisieren. Das Ergebnis der Umfrage stellt die relevanten Informationen über Microservices als Informationsbasis für das Dashboard dar. Abbildung 6-12 zeigt die Ergebnisse der Befragung von Aichinger (2017) im Vergleich zu den Ergebnissen, die im Rahmen dieser Arbeit ermittelt wurden.

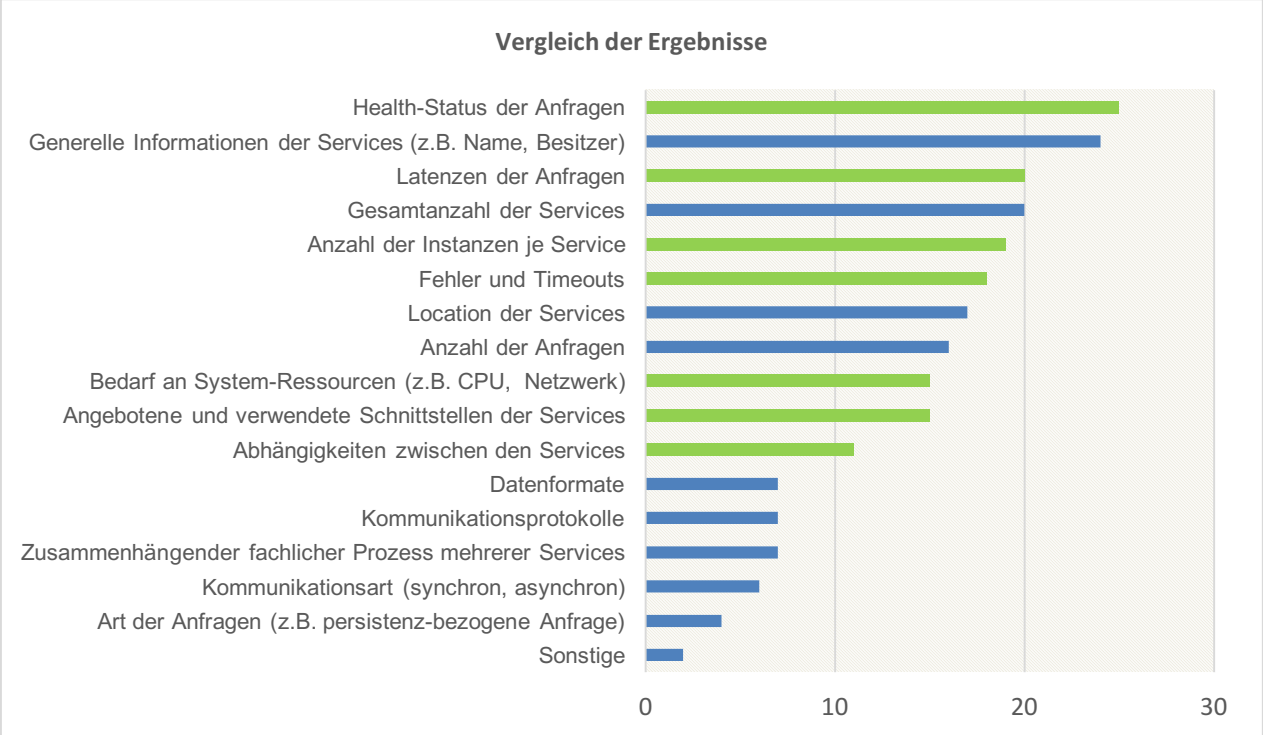

Abbildung 6-12: Vergleich der Ergebnisse

Die grünen Balken zeigen die Informationen, welche ebenfalls in der Arbeit von Aichinger (2017) vorkamen und von den Teilnehmern überwiegend als wichtig und sehr wichtig eingestuft wurden. Folgende Informationen wurden dabei ebenfalls als wichtig eingestuft, fehlen aber in der vorliegenden Arbeit:

- Version eines Microservices
- Unterschiede der Schnittstellen über verschiedene Versionen hinweg
- Technologie-Stack der Services
- Auslastung eines Microservices über einen bestimmten Zeitraum

6.3 Diskussion der Ergebnisse

Die getroffenen Annahmen aus Abschnitt 6.1.1 konnten mithilfe der Umfrage teilweise bestätigt werden. Auch wenn es einige Tendenzen bezüglich der Aufgaben der Makroarchitektur gibt, kann aufgrund der Überschneidungspunkte keine eindeutige Trennlinie definiert werden. Dies machten auch die zusätzlichen Regeln deutlich, welche sehr zwischen den Teilnehmern variierten. Auffällig war hierbei auch, dass es 21 Teilnehmer gab, die im Gegensatz zu der Empfehlung aus Wolff (2016b) die Autorisierung der Makroarchitektur zuordneten.

Bezüglich des Architekturmanagements gaben mehr als die Hälfte der Teilnehmer an, bereits automatisiert Daten zu sammeln. Aufgrund der Interpretationsschwierigkeiten bietet es sich hierbei an, bei weiteren Studien den Zusammenhang noch näher zu erläutern. Die ergänzende Studie aus Aichinger (2017) hat außerdem gezeigt, dass es hier noch weitere wichtige Daten gibt, welche als Antwortmöglichkeit bereitgestellt werden können. Das Vorhaben, weitere Daten zu ermitteln, kann als weniger erfolgreich eingestuft werden, da lediglich zwei Teilnehmer weitere Angaben machten.

Eine Reihe von Teilnehmern gab an, dass auch die Teams bei der Verantwortung über die Makroarchitektur involviert sind. Hierbei können aber keine hinreichenden Aussagen darüber getroffen werden, ob dies auf den Architekturansatz zurückzuführen ist, oder ob die Unternehmenskultur generell auf mehr Teamverantwortung ausgerichtet ist. Auch das JAXenter-Magazin fand in einer Umfrage heraus, dass bei 34% der Teilnehmer aus den DACH-Ländern, Architektur-Entscheidungen von bestimmten Teammitgliedern getroffen werden und 18% intuitive Architektur machen (JAXenter, 2016). 26% der Teilnehmer gaben an, dass spezielle Software-Architekten für diese Entscheidungen zuständig sind. Um hier verlässliche Aussagen machen zu können, wäre eine umfangreiche Studie nötig, die die Architekturansätze mitberücksichtigt und andere Einflussfaktoren systematisch ausschließt.

Mehr als 80 % der Teilnehmer verwendeten außerdem Domain-driven Design, um ihre Services fachlich aufzuteilen. Da Domain-driven Design genauso in monolithischen Architekturen Verwendung findet, wäre auch an dieser Stelle interessant, herauszufinden, ob sich die Vorgehensweise hingehend zu Microservices geändert hat.

7 DARSTELLUNG RELEVANTER INFORMATIONEN

In diesem Kapitel wird untersucht, wie sich die relevanten Aspekte einer Microservice-Architektur sinnvoll darstellen lassen. Zu diesem Zweck wird als erstes untersucht, ob sich eine Microservice-Architektur mit dem Architektur-Template arc42 beschreiben lässt und um welche Informationen dieses ergänzt werden kann. Auf Basis der Erkenntnisse wird anschließend ein entscheidungsbasiertes Modell vorgestellt, welches letztendlich mithilfe von qualitativen Interviews evaluiert wird.

arc42

Das arc42-Template ist eine standardisierte Gliederung für Architekturbeschreibungen (Starke & Hruschka, 2016). Das Template behandelt architekturrelevante Themen wie z.B. Architekturziele, Randbedingungen, Architektursichten, übergreifende technische Konzepte, Entwurfsentscheidungen und Bewertungsszenarien (Abbildung 7-1).

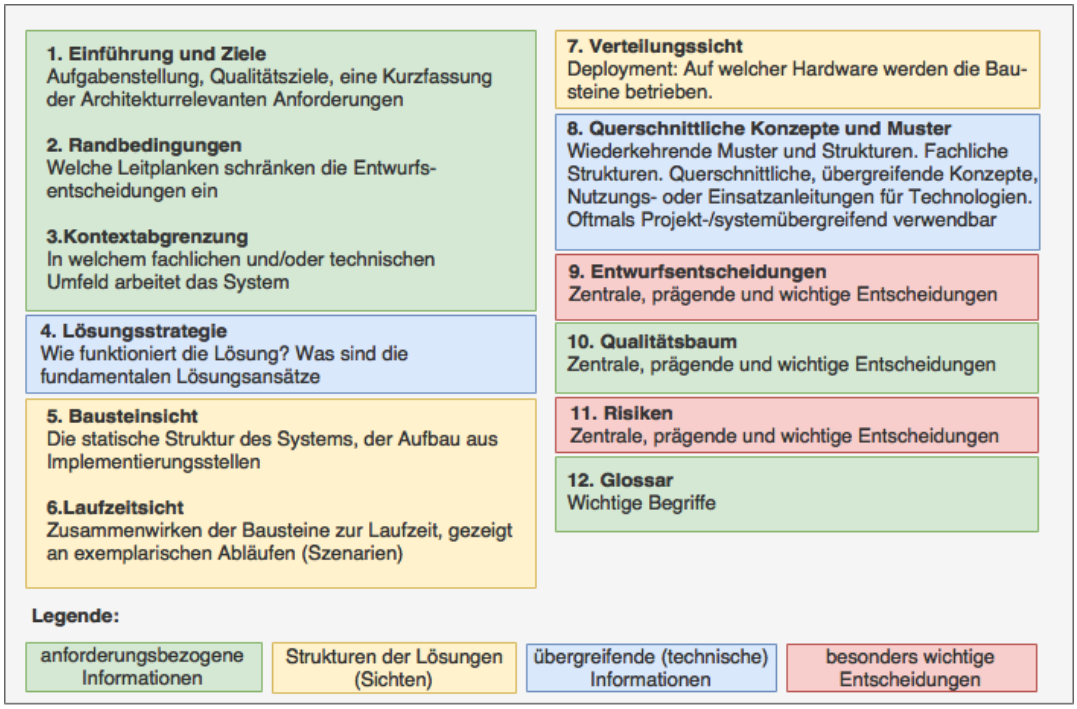

Abbildung 7-1: arc42-Template (Starke & Hruschka, 2016)

Punkt 1: Einführung und Ziele

Das Hauptziel des ersten Abschnittes ist es, allen Stakeholdern die Lösung verständlich zu machen, welche detailliert in den Punkten 3 bis 12 behandelt wird (Starke, Simons, & Zörner , 2017). Der Abschnitt zeigt die treibenden Kräfte für architektonisch relevante Entscheidungen sowie wichtige Anwendungsfälle und Merkmale in wenigen Sätzen zusammengefasst. Als Unterpunkte schlägt das Template Aufgabenstellung, Qualitätsziele und Stakeholder vor.

Anhand dieser Punkte kann bereits kritisch geprüft werden, ob der Microservice-Ansatz der richtige für das zu entwickelnde System ist. Hierbei helfen folgende Fragestellungen:

- Lässt sich die Aufgabenstellung am besten mit einer Microservice-Architektur bewältigen oder gibt es einen besseren Ansatz?
- Passen die geforderten Qualitätsziele zu den Qualitätsmerkmalen einer Microservice-Architektur?
- Hindern Personen, Rollen oder die Organisation den Strukturwandel, der mit einer Microservice-Architektur einhergehen kann?

Somit kann zu Beginn die Gefahr reduziert werden, eine Architektur zu entwerfen, welche keinen eindeutigen Mehrwert gegenüber weniger komplexen Ansätzen hat.

Punkt 2: Randbedingungen

In diesem Abschnitt wird beschrieben, welche Randbedingungen Freiheiten bezüglich der Designentscheidungen oder dem Entwicklungsprozess einschränken. Die Einschränkungen können in mehrere Ebenen wie technische und organisatorische Einschränkungen sowie Konventionen untergliedert werden. Bezogen auf Microservices können unter diesem Punkt die Designeinschränkungen auf Serviceebene, respektive die Makroarchitektur dokumentiert werden. Tabelle 7-1 und Tabelle 7-2 zeigen Beispiele für mögliche Einschränkungen.

Punkt 2.1: Technische Einschränkungen der Makroarchitektur

Einschränkung	Hintergrund / Motivation
Programmiersprache	Aufgrund vieler Spezialisten in den Teams sollen die frei wählbaren Programmiersprachen auf Java und GO beschränkt werden.
IPC-Mechanismus	Es soll einheitlich REST für die synchrone Kommunikation verwendet werden.
Monitoring	Die Daten sollen mit den Tool collectd gesammelt und zentral in Graphite zur Verfügung gestellt werden.
CDC	Für eine Rückwärtskompatibilität der Schnittstellen sollen CDCs mit Hilfe von Pact-Dateien vereinbart werden.

Tabelle 7-1: Randbedingungen: Technische Einschränkungen der Makroarchitektur

Punkt 2.2: Organisatorische Einschränkungen der Makroarchitektur

Einschränkung	Hintergrund / Motivation
Continuous Deployment	Die Anwendung soll nach dem Build direkt in die Produktionsumgebung gespielt werden.
Coding Guidelines	Serviceübergreifende Coding Guidelines sollen dafür sorgen, dass z.B. Namensschema und Kommentare bei allen Services einheitlich sind.
Team Leads	Jedes Team bestimmt eine Person, welche mit anderen Teammitgliedern und dem Projektleiter wöchentlich die Gesamtarchitektur diskutiert.

Tabelle 7-2: Randbedingungen: Organisatorische Einschränkungen der Makroarchitektur

Punkt 3: Kontextabgrenzung

Unter diesem Punkt wird der fachliche und technische Systemkontext dokumentiert (Abschnitt 2.1.5). Die Microservice-Architektur kann dabei wie herkömmliche Systeme als Blackbox in Abhängigkeit zu umliegenden Akteuren und Systemen dargestellt werden.

Punkt 3.1: Fachlicher Kontext

Abbildung 7-2 zeigt als Beispiel ein einfaches fachliches Kontextdiagramm. Auf das System, welches einen Webshop umsetzt, erhalten Nutzer und Shop-Administratoren Zugriff. Außerdem bindet das System für gewisse Bereiche ein Legacy-System ein. Die Zahlung befindet sich außerhalb des Systems und wird über einen externen Anbieter abgewickelt.

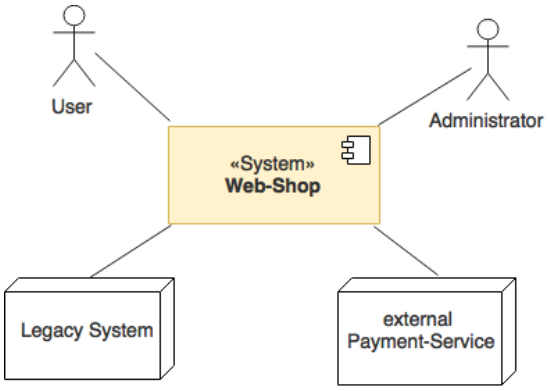

Abbildung 7-2: Fachliches Kontextdiagramm

Punkt 3.2: Technischer Kontext

Der technische Kontext zeigt die technische oder physische Infrastruktur des Systems. Hier kann bspw. ein Deployment-Diagramm (Punkt 7) zusammen mit einer Beschreibung der einzelnen Komponenten einen Überblick verschaffen.

Punkt 4: Lösungsstrategie

Hier werden die wesentlichen Lösungen, Ideen und Strategien niedergeschrieben, mit denen die Qualitätsziele des Systems erreicht werden sollen. Tabelle 7-3 zeigt Beispiele für Qualitätsziele in einer Microservice-Architektur.

Qualitätsziel	Architekturansatz
Sicherheit	Um die Nutzeridentitäten sicher über die Services zu verteilen, wird durchgängig das OAuth2-Protokoll verwendet.
Wartbarkeit	Da oft auch Entwickler zwischen den Serviceteams wechseln, wird ein einheitlicher Architekturstil für die interne Strukturierung der Services verwendet.
Zuverlässigkeit	Es wird das Circuit Breaker Pattern verwendet, damit bei Ausfall eines Service die Nutzer direkt Feedback bekommen, sobald ein Service wieder erreichbar ist.
Benutzbarkeit	Damit User über eine einheitliche Oberfläche navigieren können, werden die Services über eine gemeinsame Oberfläche integriert.

Tabelle 7-3: Lösungsstrategie: Qualitätsziele einer Microservice-Architektur

Punkt 5: Bausteinsicht

Die klassische Bausteinsicht zeigt die Struktur und Zusammenhänge zwischen den Bausteinen, wie Klassen, Komponenten, Subsystemen und deren Abhängigkeiten (Abschnitt 2.1.5). Die Bausteine einer Microservice-Architektur sind auf oberster Abstraktionsebene die Services. Da sich die Anzahl und Verbindungen der Services dynamisch ändern können, sollten hier Abhängigkeiten und Zusammenhänge automatisiert erfasst werden (Abschnitt 4.3). Die konkrete Zerlegung der Services erfolgt von den Teams. Hier gäbe es die Möglichkeit, dass die Teams in einer gesonderten Dokumentation die innere Struktur ablegen, welche unter diesem Punkt verknüpft wird.

Punkt 6: Laufzeitsicht

Die Laufzeitsicht zeigt das Verhalten, Interaktionen und Abhängigkeiten zur Laufzeit anhand von konkreten Szenarien. Auch hier sind Informationen auf Serviceebene relevant, bspw. die konkreten Aufrufe zwischen den Services innerhalb eines gemeinsamen fachlichen Prozesses zur Laufzeit. Die Befragung hat gezeigt, dass einige Unternehmen bereits solche Daten sammeln. Ergänzend wären an dieser Stelle auch klassische Sequenzdiagramme denkbar, bspw. um synchrone Eventabläufe darzustellen (Abbildung 7-3).

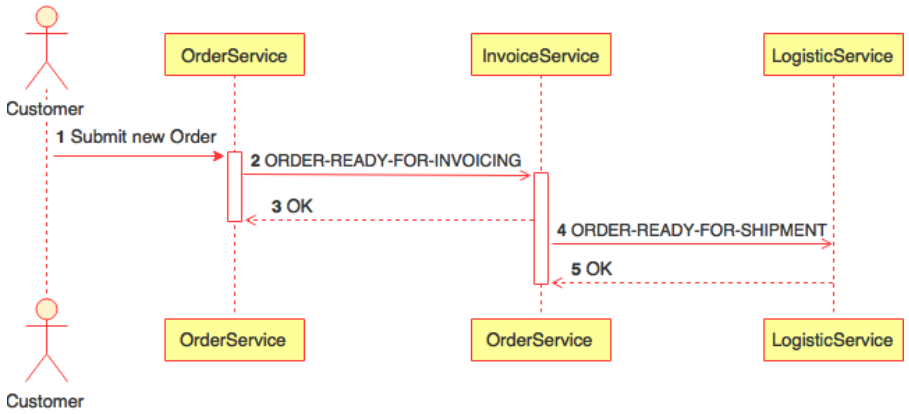

Abbildung 7-3: Sequenzdiagramm Laufzeitsicht (Soika, 2016)

Punkt 7: Verteilungssicht

In diesem Abschnitt wird die technische Infrastruktur dargestellt, auf der die Bausteine ausgeführt werden. Hierzu zählen u.a. Themen wie Skalierung, Clustering, automatisches Deployment, Firewalls und Load Balancing. Somit können hier relevante Informationen bezüglich der Infrastruktur und des Betriebs von Microservices festgehalten werden. Abbildung 7-4 zeigt als Beispiel ein Verteilungsdiagramm für eine Containerlösung mit Microservices.

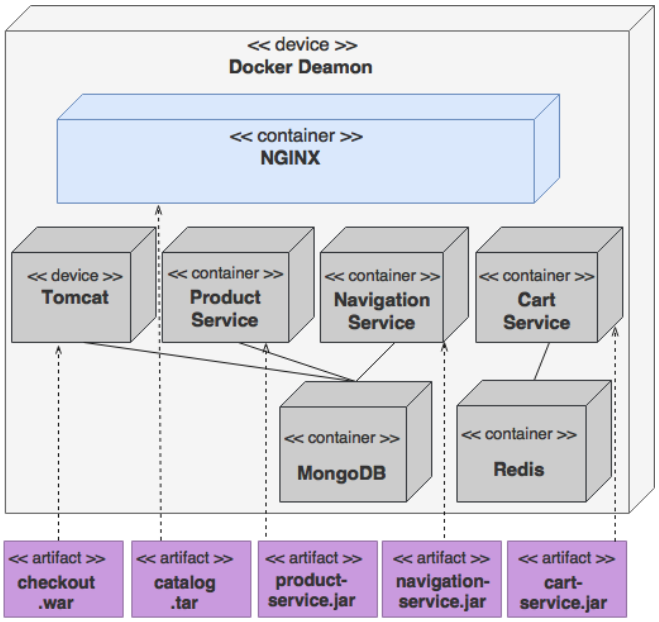

Abbildung 7-4: Verteilungssicht (Zuther, 2015)

Punkt 8: Querschnittliche Konzepte und Muster

Hier können Domänenmodelle, Architektur- und Entwurfsmuster, Regeln für die Verwendung bestimmter Technologien, Implementierungsregeln und übergreifende technische Entscheidungen beschrieben werden. Somit können in diesem Abschnitt bspw. folgende Fragen beantwortet werden:

- Wie sieht das Domänenmodell im Detail aus?
- Wie müssen übergreifende Architekturstile verwendet werden?
- Wie werden Querschnittsfunktionalitäten realisiert?
- Wie werden die Services integriert?
- Wie teilen sich die Services gemeinsame Daten?
- Wie kommunizieren die Services miteinander?
- Wie werden ggf. Templates, Blueprints oder Bibliotheken verwendet?

Handelt es sich bei dem zu beschreibenden System ausschließlich um eine Referenzarchitektur oder ein Framework, kann anstelle des letzten Punktes die Dokumentation auch insgesamt mit zielgruppenspezifischen Inhalten darauf ausgelegt werden (Zörner, 2012).

Punkt 9: Entwurfsentscheidungen

An dieser Stelle werden wichtige, risikoreiche, teure oder spezielle Architekturentscheidungen festgehalten. Idealerweise werden detailliert das Problem, Einschränkungen, Annahmen und Alternativen mitdokumentiert. Nachfolgend werden beispielhaft Entscheidungen angeführt, die für eine Microservice-Architektur getroffen werden müssen:

- Wie werden bestehende Legacy-Systeme integriert?
- Wie wird die Granularität der Services bestimmt?
- Wie wird bspw. Abwärtskompatibilität oder Zustandslosigkeit erreicht?

Da sich übergreifende Entscheidungen auf alle Services auswirken, können potentiell alle Entscheidungen angeführt werden, welche die Makroarchitektur betreffen. Entscheidungen können zudem schwerwiegendere Folgen haben als bei monolithischen Systemen, da die Refaktorisierung bei Microservices mit einem höheren Aufwand verbunden ist.

Qualitätsbaum

In diesem Abschnitt werden alle Qualitätsanforderungen innerhalb eines Qualitätsbaums mit Szenarien versehen (Abschnitt 2.1.7). Neben den wichtigsten Qualitätszielen aus Punkt 1 können hier auch niedriger priorisierte Ziele detailliert beschrieben werden. Abbildung 7-5 zeigt einen Qualitätsbaum, welcher mit Beispielszenarien aus Tabelle 7-4 verknüpft ist.

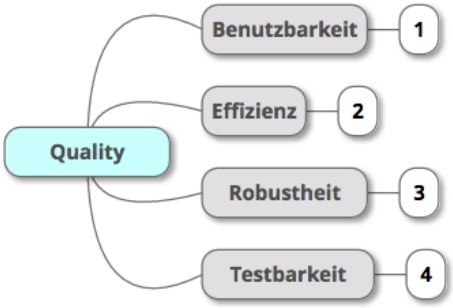

Abbildung 7-5: Qualitätsbaum

Nummer	Szenario
1	Ein Benutzer navigiert von der Produktdetailseite zur Kasse und bemerkt nicht, dass die Anfrage von mehreren Services bearbeitet wird.
2	Es werden 10.000 parallele Nutzer mit einer durchschnittlichen Antwortzeit von einer Sekunde gleichzeitig bedient.
3	Trotz Ausfall eines Service während der Zahlungsübermittlung können Benutzer den Vorgang mit einer zeitlichen Verzögerung von maximal fünf Sekunden abschließen.
4	Ein Entwickler bringt eine neue Serviceversion in Produktion und erhält Feedback, ob die Integration erfolgreich war.

Tabelle 7-4: Qualitätsszenarien einer Microservice-Architektur

Punkt 10: Risiken

Unter diesem Punkt können technische Risiken und potentielle Probleme priorisiert angeführt werden. Die Risiken können außerdem in technische Risiken und Geschäftsrisiken unterteilt werden. Tabelle 7-5 zeigt ein Beispiel, inklusive einer Beschreibung und Maßnahmen, um die Risiken zu minimieren.

Risiko	Beschreibung	Minimierung
Sinkende Robustheit	Mit einer hohen Anzahl an synchronen Kommunikationsaufrufen steigt auch die Wahrscheinlichkeit von kaskadierenden Fehlern.	Konsequente Umsetzung des Circuit Breaker Patterns
Team Leads	Das Wissen der einzelnen Team Leads über die Gesamtarchitektur kann zum Flaschenhals werden, wenn es nicht regelmäßig im Team geteilt wird.	Regelmäßige Wissens-transfers / Rotation der Teamleads
Sinkende Servicequalität	Übergreifende Themen wie CDC, Architekturstil und Sicherheit werden nicht von jedem Entwickler beachtet.	Jedes Team hat einen einheitlichen Rahmen für Architekturrichtlinien.

Tabelle 7-5: Risiken

Punkt 11: Glossar

In diesem letzten Punkt werden die wichtigsten Fachbegriffe und Ausdrücke, die Stakeholder bei der Kommunikation verwenden, erklärt.

Zusammenfassung

Grundsätzlich kann eine Microservice-Architektur mit arc42 beschrieben werden. Bezogen auf die Baustein- und Laufzeitsicht, müssen hier anstelle von Klassen, Komponenten und Objekten die Services in den Fokus rücken. Aufgrund der Dynamik innerhalb einer Microservice-Architektur bieten sich hier automatisierte Lösungen an. Das Domänenmodell wird bei Microservices umso wichtiger. Kontextgrenzen und Abhängigkeiten bestimmen die spätere Aufteilung und Abhängigkeiten der Services und somit nach Conways Law auch die der Teams. Obwohl die Grenze zwischen Makro- und Mikroarchitektur in verschiedenen Punkten untergebracht werden kann, wäre hier eine Hervorhebung von Vorteil. Unter einem gesonderten Abschnitt könnte hier neben dem *was* und dem *wie* auch das *wer* dokumentiert werden, um die Verantwortlichkeiten transparent zu machen. Da das arc42-Template auf keinen bestimmten Architekturansatz spezialisiert ist, fehlen außerdem konkrete Informationen einer Microservice-Architektur, welche bei der Dokumentation unterstützen.

7.1 Entwicklung eines prototypischen Modells

Um diese Lücke zu schließen, soll ein prototypisches Modell entwickelt werden, welches die relevanten Informationen aus dem theoretischen Teil und den Ergebnissen aus der Befragung zusammenfasst. Dabei soll es sich nicht auf ein bestimmtes Szenario festlegen, sondern verschiedene Alternativen abbilden, mit denen eine Microservice-Architektur umgesetzt werden kann. Zielgruppe sind Projektbeteiligte, welche vor der Herausforderung stehen, an den Entwurfsentscheidungen einer Microservice-Architektur mitzuwirken. Hierbei soll das Modell einen Überblick verschaffen und bei der weiteren Kommunikation und Entscheidungsfindung unterstützen.

Modellbeschreibung

Als Ergebnis wurde ein vorwiegend textuelles Modell entwickelt, welches die wichtigen Entscheidungen darstellt, die in einer Microservice-Architektur getroffen werden müssen. Dieses bezieht sich ausschließlich auf die wichtigen Fragestellungen und liefert keine konkreten Vorgaben für die weitere Dokumentation. Somit kann es als Unterstützung der *Struktur und Leitfragen zur Bearbeitung einer Architekturentscheidung* nach Zörner (2012) angesehen werden (Abbildung 7-6), wobei es anstelle des konkreten Systems die allgemeinen Entscheidungsfragen innerhalb einer Microservice-Architektur hervorhebt.

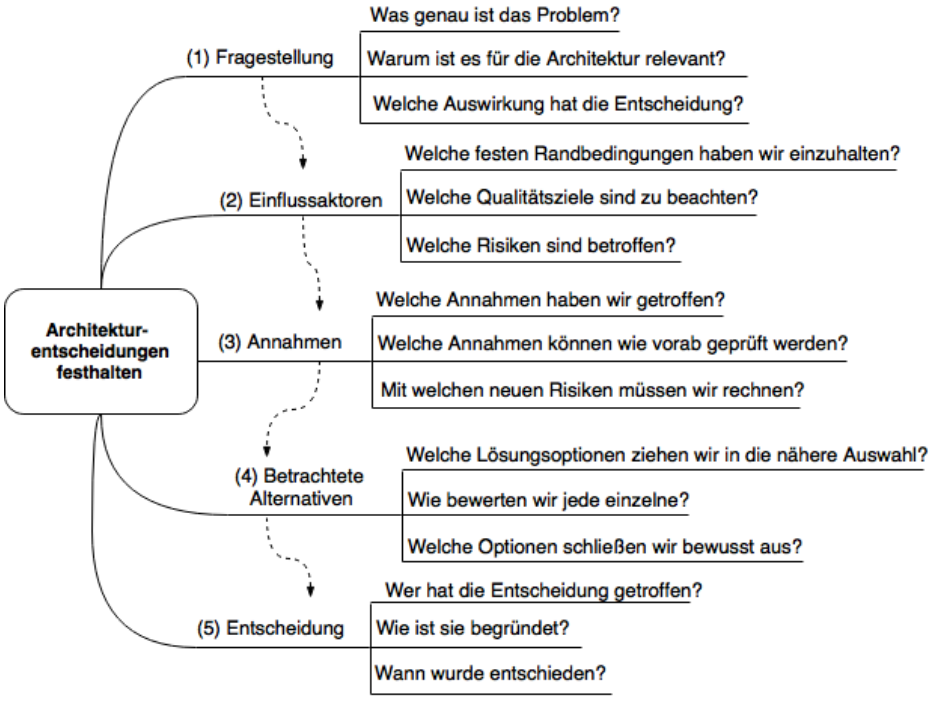

Abbildung 7-6: Struktur und Leitfragen zur Bearbeitung einer Architekturentscheidung (Zörner, 2012)

Entscheidungsebenen

Die Entscheidungen wurden in folgende Ebenen gruppiert (Abbildung 7-7):

- **Organisation:** Hier befinden sich die organisatorischen Fragestellungen. Diese betreffen die technischen und organisatorischen Rahmenbedingungen sowie Entscheidungen bezüglich der Makro- und Mikroarchitektur.

- **Fachliche Architektur**: Auf dieser Ebene befinden sich die Fragestellungen bezüglich der Vorgehensweise bei der fachlichen Aufteilung, der Servicegranularität und den Abhängigkeiten zwischen den Services.
- **Infrastruktur:** Diese Ebene beschreibt alle infrastrukturellen Entscheidungen, bspw. in Bezug auf Legacy-Systeme, Plattform und Produkte sowie Deployment und Interprozesskommunikation.
- **Querschnittsfunktionalitäten:** Hier befinden sich die Entscheidungen bezüglich der Querschnittsfunktionalitäten wie Monitoring, Logging, Authentifizierung, Autorisierung, Fehlerbehandlung und Datenhaltung.
- **Service:** Auf dieser Ebene werden die individuellen Entscheidungen auf Serviceebene getroffen, weshalb sie im Modell keine Verwendung finden.

Für jede Ebene wurde eine Tabelle erstellt, welche die wichtigen Fragestellungen zusammen mit einer Beschreibung beinhaltet. Auf Ebene der Organisation und fachlichen Architektur wurden zusätzlich mögliche Auswirkungen beschrieben, da sich die Entscheidungen in dieser frühen Phase stark auf die weiteren Ebenen auswirken können.

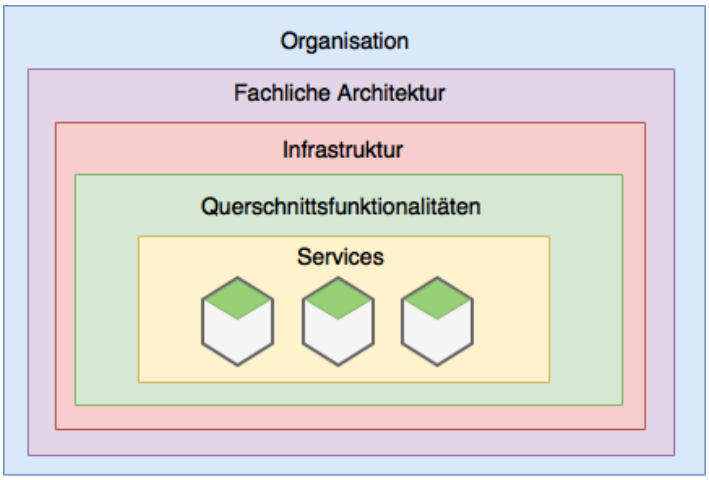

Abbildung 7-7: Entscheidungsebenen

7.2 Evaluation

Um zu untersuchen, inwieweit das Modell die für die Zielgruppe relevanten Informationen bereitstellt, wurde eine Evaluation durchgeführt. In diesem Abschnitt werden Aufbau, Vorgehen und Ergebnisse dieser Evaluation erläutert und abschließend diskutiert.

7.2.1 Aufbau und Zielgruppe

Für die Evaluation wurden mit sechs Probanden *fokussierte Interviews* durchgeführt. Hierbei sollte ermittelt werden, ob die vorab bereitgestellten Informationen einen Überblick über die wichtigen Entscheidungen einer Microservice-Architektur geben oder ob ggf. weitere

Informationen benötigt werden. Vor den Interviews wurde ein Leitfaden (Anhang C) erstellt, welcher in folgende Abschnitte strukturiert wurde:

- **Gesprächseröffnung:** Für den Einstieg in das Gespräch wurden Fragen bezüglich der Rolle und Aufgaben im Unternehmen gestellt. Weitere Fragen bezogen sich bspw. auf die Erfahrungen mit Softwarearchitektur und Modellierung.
- **Allgemeine Fragen:** In diesem Teil wurden explizit Fragen zu dem Modell gestellt. Hierbei sollte herausgefunden werden, inwieweit die Aufteilungen der Ebenen sowie die einzelnen Informationen innerhalb der Ebenen verständlich und hilfreich sind und welche weiteren Informationen ggf. benötigt werden.
- **Ergänzende Fragen:** Je nach Wissensstand wurden weitere Fragen definiert, die zur Vertiefung in die Thematik dienen.

Trotz Leitfaden wurde das Interview bewusst offengehalten, damit sich ggf. neue Gesichtspunkte ergeben können und sich die Fragen nach dem Kenntnisstand des Teilnehmers richten konnten.

7.2.2 Vorgehen

Vor dem Interview wurde den Teilnehmern das Modell aus Anhang B und eine Datenschutzvereinbarung (Anhang C) zugesendet. Die Interviews wurden anschließend telefonisch durchgeführt und aufgezeichnet. Für die Auswertung wurde das Interview transkribiert (Anhang C). Hierbei wurden die für die Arbeit relevanten Fragen und Antworten übertragen. Einleitung, Schluss- und Randgespräche sowie parasprachliche Merkmale wurden ausgelassen. Anschließend wurden die Ergebnisse thematisch eingeordnet, verglichen und interpretiert. Als Ergänzung wurde mit *Eberhard Wolff* ein Experte herangezogen, welcher sich bereiterklärte, das Dokument zu kommentieren.

7.2.3 Ergebnisse

Allgemeine Teilnehmerinformationen

Für den Einstieg in das Gespräch, wurden den Teilnehmern allgemeine Fragen zum Unternehmen und ihrer Rolle gestellt. Tabelle 7-6 zeigt eine Übersicht der Teilnehmer zusammen mit ihren Erfahrungen mit Microservices und das Interesse an dem Thema.

Der Softwarearchitekt konnte bereits erste Erfahrungen mit Microservices sammeln. Hierbei bezieht er bei Entscheidungen, z.B. Microservices umsetzen zu wollen, die Verantwortlichen wie Geschäftsführer und Inhaber mit ein. Für die konkrete Umsetzung der Services stimmt er sich außerdem mit den Teamleitern und Vertretern aus der Entwicklung oder dem gesamten Entwicklungsteam ab, um bspw. den Bounded Context der Services festzulegen. Bei der Modellierung werden dabei keine speziellen Werkzeuge verwendet. Die Dokumentation erfolgt anhand von arc42 mit Fokus auf Sequenzdiagrammen, um die Kommunikation zwischen den Services abzubilden.

Rolle	Erfahrung mit Microservices	Interesse
Softwarearchitekt	Erste praktische Erfahrungen	Einführung von Microservices im Unternehmen
Testmanager / Entwickler	Theoretische Kenntnisse	Masterarbeit zum Thema Microservices
Business Analystin / Spezifikateurin	Keine	Entwicklung einer Microservice-Architektur im zukünftigen Projekt
Softwareentwickler A	Theoretische Kenntnisse	Allgemeines Interesse, Erfahrungen mit SOA
Softwareentwickler B	Theoretische Kenntnisse	Masterarbeit zum Thema Microservices / Zukünftiges Projekt
Webentwickler	Keine	Potentielle zukünftige Projekte

Tabelle 7-6: Kenntnisstand und Interesse an Microservices

Allgemeine Anmerkungen zum Modell

Der Business Analystin fehlte grundsätzlich als Einstieg die Definition einer Microservice-Architektur. Softwareentwickler B bemerkte, dass er das Modell nur verstehe, weil er bereits das entsprechende Hintergrundwissen erworben hat. Für Softwareentwickler A deckt das Modell die wesentlichen Informationen ab. Er verweist allerdings auf die Architektursichten (Abschnitt 2.1.5), mit denen weitere Perspektiven aus unterschiedlichen Blinkwinkeln eingenommen werden müssen, da diese nicht mit einer einzigen Darstellung abgedeckt werden können. Für die Business Analystin ist hier bspw. interessant, welche Informationen zwischen ihr und den Entwicklern geteilt werden müssen, damit für beide Seiten klar ist, welcher Service bei der Spezifikation oder Entwicklung betroffen ist. Außerdem würde sie gerne wissen, was sie, in Bezug auf die bei Microservices verwendeten Technologien, beachten muss und welche Auswirkung diese auf ihre Spezifizierung oder Anforderungsanalyse haben.

Anmerkungen zur Darstellung der Entscheidungsebenen

Die Entscheidungsebenen stellten für Softwareentwickler B einen guten roten Faden dar, an dem sich Entscheider orientieren können. Zusätzlich war dieser der Meinung, dass durch die Darstellung der Ebenen klar wird, dass man mit einer Microservice-Architektur nicht sofort beginnen könne, sondern deren Einsatz planen muss. Der Testmanager fragte, ob es einen direkten Bezug zwischen den Ebenen und zur Makro- und Mikroarchitektur gibt. Er wies zudem darauf hin, dass die fachliche Architektur auch ein Teil der Makroarchitektur sein müsste und daher auf der Organisationsebene mitaufgenommen werden sollte.

Anmerkungen zur Organisationsebene

Bezogen auf die Organisation, merkte der Testmanager an, dass man sich die Frage nach der Teamorganisation zu Beginn stellen sollte, bspw. ob die Teamstruktur klar ist oder abgestimmt wurde. Außerdem wollte dieser wissen, wie die Aufteilung in dem Praxisbeispiel aus Tabelle 2 (Makro- und Mikroarchitektur) entschieden wurde. Beim Lesen kam ihm der Gedanke, dass er die Programmiersprache der Makroarchitektur zuordnen würde, „um die notwendigen Skills zur

Wartung im Rahmen zu halten“. Für die Business Analystin war nicht verständlich, warum sich die Organisationsstruktur in einer Microservice-Architektur ändert. Auch hat ihr bei der Entscheidung, ob man mit einem Monolithen oder mit Microservices startet, ein Beispiel gefehlt, welches die Auswirkungen verdeutlicht. Für Softwareentwickler B ist zudem der Unterschied zwischen Makro- und Mikroarchitektur nicht klargeworden.

Anmerkungen zur fachlichen Architektur

Softwareentwickler B bewertet die Tabelle 5 (Servicegranularität) als sehr gut, da sie „große, kleine Services gegenüberstellt“ und dabei zeigt „was das für Konsequenzen hat“. Auch der Webentwickler bewertet diese Tabelle als hilfreich. Für die Business Analystin ist die fachliche Architektur am relevantesten, da ihrer Meinung nach, ihr Einfluss auf die organisatorischen Aspekte gering ist. Besonders interessiert sie dabei „die Vorgehensweise, wie die Services fachlich aufgeteilt werden, da sich daraus vermutlich die Anforderungen ableiten lassen“ oder weil sie „bei der Spezifikation die Aufteilung berücksichtigen würde“. Auch ist für sie interessant, wer für das Erstellen der Context Map verantwortlich ist und ob es neben DDD noch andere erprobte Vorgehensweisen gibt. Sie bewertet es außerdem als gut, dass die Erklärungen zu DDD mit Abbildungen ergänzt wurden.

Anmerkungen zur Infrastruktur

Der Softwarearchitekt merkt zur Abbildung 5 (Infrastruktur) an, dass es so aussieht „als ob von außen durch die Service Registry mit den Services kommuniziert wird“. Hierbei stellt er klar, dass die Services Registry nur für die Discovery genutzt wird und die Kommunikation direkt über die API Gateways erfolgt. Softwareentwickler B bemerkt ebenfalls die Abwesenheit des API Gateways, sieht diesen aber als alternativen Ansatz zu einer Backend-/Frontendlösung, bei der die Services über die Edge Services mit dem Client kommunizieren. Der Testmanager empfand es außerdem als wichtig, die Tabelle zu visualisieren, um zu sehen „wo dann auch die Load Balancer stehen“ und um den Begriff Edge Service näher zu erläutern. Softwareentwickler B hätte gerne mehr Informationen über die Lokalisierung, z.B. mithilfe eines praktischen Beispiels. Die Entscheidungsfragen bewertet dieser aber insgesamt als „ziemlich gut“ und ergänzt, dass diese die wichtigsten Dinge beinhalten. Bei Tabelle 8 (Deployment-Varianten) stellte dieser außerdem fest, dass die VM-Varianten eigentlich nicht als Information benötigt werden, da sie deutliche Nachteile mit sich bringen. Er verweist hier auf die Containerlösungen, die sich inzwischen etabliert haben, und sieht die ersten Varianten eher als Evolutionsschritte.

Anmerkungen zu Querschnittsfunktionalitäten

Softwareentwickler B stellte sich bei den Querschnittsfunktionalitäten die Frage nach der Funktionsweise von Event Sourcing. In Bezug auf die Datensammlung wäre für Softwareentwickler A ein Überblick interessant, welche Services kombiniert werden können und welche Auswirkungen auf die Latenz zu erwarten sind. Hierfür schlägt er vor, SLAs für Services zu definieren und sich über diese einen Überblick zu verschaffen. Für ihn ist außerdem interessant, wie viele Requests die Services verarbeiten können, bis es zu einem Ausfall kommt. Zur Tabelle 10 (Datensammlung) merkt der Testmanager an, dass der Punkt Datenformate „in die Konzepte der Anwendung und in die Architekturdokumentation“ gehört,

weil in der Regel bekannt ist, welche Datenformate gesendet werden. Abbildung 7 (Monitoring) bringt diesem außerdem keinen eindeutigen Mehrwert. Die Business Analystin nennt als relevante Daten „die maximale Response Time und, ob die Schnittstelle synchron oder asynchron läuft. Auch nennt sie „das Datenvolumen, welches verarbeitet werden kann, und die Direction of Transmisson“.

Qualitative Verbesserungsvorschläge

Neben den inhaltlichen Anmerkungen haben die Teilnehmer außerdem Verbesserungsvorschläge bezüglich der Qualität des Modells gemacht. Diese können in folgende Punkte zusammengefasst werden:

- Bessere Verknüpfung der Inhalte
- Deutlichere Kennzeichnung von Daten der Umfrage und sonstigen Quellen
- Deutlichere Kennzeichnung von Beispielen
- Aufteilung von Tabelle 2 in zwei Spalten
- Matrizendarstellung für Tabelle 5
- Erläuterung der Abkürzungen

Expertenkommentar

Die Aufteilung der Ebenen empfand Eberhard Wolff als sinnvoll und merkt an, eine solche Aufteilung noch nicht gesehen zu haben.

Wolffs Meinung nach, ist der Start eines Monolithen allerdings wenig sinnvoll. Er verweist hier auf Tilkov (2015). Dieser behauptet, dass es in den meisten Fällen schwer bis unmöglich ist, einen Monolithen zu zerschneiden. Hierbei bezieht er sich auf Newman (2015), der empfiehlt, mit einem sogenannten Brownfield-System zu starten. Dennoch ist Tilkov (2015) auch der Meinung, dass man vor dem Start mit Microservices, die Domäne gut kennen sollte und schlägt daher als Idealszenario vor, den Monolithen komplett zu ersetzen (auch *Sacrificial Architecture* (Fowler M. , 2014b)).

In Bezug auf die Verantwortlichkeiten zwischen Makro- und Mikroarchitektur verweist Wolff auf sein Werk (Wolff, 2016b). Dort beschreibt er, dass die Makroarchitektur von Vertretern der Serviceteams festgelegt werden kann, mit den Nachteilen, dass diese eine zu fokussierte Sicht auf die eigenen Services haben oder das Komitee bei vielen Teams ggf. zu groß wird. Als Alternative nennt er Architekturteams, mit dem Risiko, dass sich deren Sicht auf das System zu weit von denen der Teams entfernt. Er stellt in seinem Kommentar auch nochmal klar, dass er die Teams immer mit einbeziehen würde, da dies ansonsten in Widerspruch mit der Selbstorganisation steht.

Wolff sieht außerdem die organisatorischen Voraussetzungen aus Tabelle 1 kritisch. Dies ist nach ihm „nicht notwendig, wenn man Microservices nur wegen technischer Gründe wie Skalierung einführt“. Außerdem stellt er sich an dieser Stelle die Frage, ob es eine signifikante Auswirkung hat, wenn es nur ein Scrum-Team gibt.

Bezüglich Tabelle 4 (Durchsetzung der Makroarchitektur) kommentiert Wolff, dass er die Regeln durch automatisierte Tests durchsetzen würde (Abschnitt 3.6).

Bei der Granularität der Services würde Wolff zwischen fachlicher und technischer Aufteilung unterscheiden. So kann die fachliche Aufteilung der Teams anhand von Bounded Context in der Makroebene entschieden werden und eine weitere Zerlegung der Services, bspw. aufgrund von Skalierbarkeit, auf Mikroebene erfolgen. Des Weiteren würde er über die internen Strukturen wie Aggregate keine Aussagen machen, da sich diese je nach Microservice unterscheiden können. In Bezug auf die fachliche Aufteilung fügte Wolff noch hinzu, dass ihm kein anderes Vorgehen neben DDD bekannt sei.

Bei der Integration von Legacy-Systemen merkte Wolff an, dass der Zugriff auf die Datenbank über einen Anticorruption Layer zu einer engen Koppelung führt. Auf Nachfrage, ist dies seiner Meinung nach auch der Fall, wenn es sich dabei nur um einen lesenden Zugriff handelt. Er verweist hier erneut auf sein eigenes Werk (Wolff, 2017), wo er beschreibt, dass auch die Koppelung über die View oft zu einer engen Koppelung mit denselben Problemen führt. Die Daten sollten demnach im Sinne des Information Hidings und nach der Definition von Modulen aus Parnas (1971) in den Modulen oder Microservices versteckt bleiben.

Die Frage nach der Konfiguration der Services kann nach Wolff in zwei Themen aufgeteilt werden. Zum einen stellt sich die Frage, wie der Service seine Konfiguration bekommt, bspw. über eine Umgebungsvariable oder Konfigurationsdatei, und zum anderen, wo die Konfiguration abgespeichert wird, z.B. in einer Datenbank.

Bezogen auf die Lokalisierung der Services (Abbildung 5), stellt Wolff die Frage, ob eine Lokalisierung mit einer Discovery notwendig sei, wenn Messaging benutzt wird (Abschnitt 3.6). Er merkt außerdem an, dass in Abbildung 5 ein synchrones Modell dargestellt wird, welches nicht zwingend verwendet werden muss. Auch informiert er darüber, dass REST auch asynchron verwendet werden kann und verweist auf Westheide (2016). Dieser macht dort deutlich, dass das Problem nicht die Auswahl von asynchroner oder synchroner Kommunikation zwischen den Services sei, sondern dass Services überhaupt während eines User Requests kommunizieren und damit die Robustheit gefährden (Abschnitt 5.4).

Neben den dargestellten Deployment-Varianten unter Tabelle 8 fehlt nach Wolff noch eine PaaS-Lösung wie OpenShift oder Kubernetes. Bezüglich der virtuellen Maschinen oder Container stellt er zusätzlich die Frage, ob man hier nicht Technologien wie Linux Packages, Puppet und Chef näher betrachten sollte.

Die Autorisierung gehört nach Wolff auf Mikroebene, da jede Komponente am besten weiß, was fachlich erlaubt ist (Abschnitt 3.5). Auch wurde die Fehlerbehandlung als Querschnittsfunktionalität hinterfragt, bspw. wenn die Teams innerhalb der Services selbständig einen Circuit Breaker implementieren.

Abschließend stellte Wolff noch richtig, das Event Sourcing die Inkonsistenzen nicht vermeidet, sondern repariert.

7.3 Diskussion der Ergebnisse

Die Ergebnisse haben gezeigt, dass das Modell ein Vorwissen über Microservices voraussetzt. Um einigen Stakeholdern den Einstieg zu erleichtern, können hier Themen wie Makro- und Mikroarchitektur näher erläutert werden. Auch in Bezug auf Conways Law können Beispiele helfen, welche die Auswirkungen des fachlichen Schnitts auf die Teams und umgekehrt deutlich machen. Dass dabei nach Wolff ein Strukturwandel nicht immer erfolgen muss, bspw. bei technischer Motivation oder einem bestehenden crossfunktionalen Team, kann ebenfalls mit eingearbeitet werden.

Es wurde außerdem deutlich, dass eine konkrete Zuordnung der Entscheidungsfragen zur Makroarchitektur nicht immer treffend ist. Je nach Definition können bestimmte Fragen wie die Fehlerbehandlung und Autorisierung auf Makro- und Mikroebene gestellt werden. Hier müssten sich die Fragen entweder von den Architekturebenen abgrenzen oder es muss eine mehrstufige Zuordnung erfolgen (Abschnitt 4.2).

Auch wurde klar, dass die Informationen zwar einen guten Überblick verschaffen, aber noch weiter detailliert werden können. So können z.B. die Fragen auf weitere Unterfragen heruntergebrochen werden oder Konzepte näher erläutert werden. Ebenfalls können weitere teilnehmerspezifische Informationen erfragt werden, um diese für bestimmte Zielgruppen aufzubereiten. Mögliche Informationen, die hier gewonnen werden können, deuteten sich bereits mit der Angabe der zielgruppenspezifischen Metadaten an.

Eine weitere Erkenntnis ist, dass die Auswirkungen der Entscheidungen auch auf den unteren Ebenen beschrieben werden sollten. Hier könnte bspw. die Kommunikation zwischen den Services während User Requests zur Infrastrukturebene eingeordnet werden. Auch wäre es sinnvoll, bekannte Alternativen wie z.B. in diesem Fall die Datenreplikation mit Atom Feeds anzugeben.

Es kann ebenfalls diskutiert werden, ob der Start mit einem Monolithen eine erwähnenswerte Entscheidung ist. Man könnte argumentieren, dass sich die Frage nach einem Monolithen nicht stellt, wenn man sich bereits für Microservices entschieden hat. Auf der anderen Seite können Ansätze wie eine *Sacrificial Architecture* Teams helfen, erste Erfahrungen mit der Domäne zu sammeln. Auch hier wäre es wichtig, weitere Auswirkungen und Alternativen näher zu beschreiben, um auf Basis dieser Informationen den vorher gewählten Ansatz vielleicht nochmal zu überdenken.

Tabelle 7-7 zeigt eine Zusammenfassung der Erweiterungsmöglichkeiten des Modells.

Element	Optimierung / Erweiterung
Tabellen	Kennzeichnen von Beispielen und Umfrageergebnissen
Abkürzungen	Glossar für Abkürzungen
Verknüpfungen	Verdeutlichung des Gesamtzusammenhanges mit weiteren Verknüpfungen
Konzepte	Weitere Erläuterungen der Konzepte, um das benötigte Vorwissen in Grenzen zu halten
Granularität	Unterscheidung von fachlicher und technischer Strukturierung
Infrastruktur (Abbildung 5)	Erweiterung um weitere Alternativen wie API Gateway, Backend for Frontend oder asynchrone Kommunikation
Lokalisierung	Messaging als Alternative zur Service Discovery
Deployment-Varianten (Tabelle 8)	Erweiterung um PaaS-Lösungen, wie OpenShift oder Kubernetes. Bei Container und VM-Technologien wie Linux Packages, Puppet und Chef betrachten
Metadaten	Erweiterung servicerelevanter Daten, wie Datenvolumen, maximale response time, Anzahl bearbeitbarer Requests, Direction of Transmission, ggf. geordnet nach Zielgruppe
Mehrstufige Entscheidungen	Weitere Aufteilung der Fragen. Beispiel Service Discovery: Wie bekommt der Service seine Konfiguration? Wo wird die Konfiguration gespeichert?

Tabelle 7-7: Erweiterung des Entscheidungsmodells

8 FAZIT UND AUSBLICK

Ziel dieser Arbeit war es, herauszufinden, welche Informationen für eine Microservices-Architektur von Relevanz sind und daher als Modell sichtbar gemacht werden sollten. Zu diesem Zweck wurden zunächst die relevanten Informationen aus der Literatur in infrastrukturelle und querschnittsfunktionale Themen sowie fachliche Architektur aufbereitet. Anschließend wurden erste Ansätze der Modellierung aus der Literatur untersucht. Dabei wurde deutlich, dass eine Microservice-Architektur im Sinne eines emergenten Systems auch äußeren Einflüssen wie Organisation, Prozesse und Kultur unterliegt. Wesentlicher Treiber ist dabei die Organisation, welche anhand von Richtlinien auf der Makroarchitekturebene die Freiheitsgrade auf Serviceebene einschränkt. Zudem kann sich die Organisation nach Conways Law direkt auf die Systemstruktur auswirken. Hier reifte die Erkenntnis, dass die anfängliche Motivation nur infrastrukturelle und fachliche Aspekte zu betrachten, wichtige Informationen auslassen würde.

In einer quantitativen Befragung wurde daher auch untersucht, wie die Makro- und Mikroarchitektur in der Praxis definiert ist und für die Teams transparent gemacht wird. Die Ergebnisse haben gezeigt, dass es eine Tendenz bezüglich der Aufgaben der Makroarchitektur gibt, aber aufgrund einiger Überschneidungspunkte keine klare Trennlinie gezogen werden kann. Auf Mikroarchitekturebene hatten die Teams mehrheitlich Freiheiten bezüglich Programmiersprache und Datenbank und wurden ebenfalls bei den Entscheidungen der Makroarchitektur mit einbezogen. Inwieweit sich der Microservice-Ansatz auf die Verantwortlichkeiten einer Architektur auswirkt, wurde jedoch nicht untersucht und könnte einen weiteren interessanten Untersuchungsgegenstand darstellen.

Weiter wurde mit Domain-driven Design eine gängige Möglichkeit vorgestellt, mit der die Kontextgrenzen einer Microservice-Architektur modelliert werden können. Hier stellte sich die Frage, wie das Modell in einer sich dynamisch ändernden Servicelandschaft auch langfristig konsistent gehalten werden kann. Die Teilnehmer der Befragung nutzen bereits automatisierte Lösungen, um über verschiedene Kanäle wie Monitoring, Service Discovery und Logging ihre Daten zu sammeln. Einige relevante Daten, welche jedoch von großen Unternehmen wie Netflix und Soundcloud gesammelt werden, befanden sich dabei im unteren Bereich. Hier werden sich sicherlich in den nächsten Jahren, mit steigender Erfahrung und weiteren Frameworks, noch Verbesserungen ergeben.

Um einen Vergleich mit konventioneller Modellierung zu ziehen, wurde außerdem untersucht, ob sich Microservices mit einem klassischen Template wie arc42 beschreiben lassen. Im Unterschied zu klassischen Architekturen ist bei Microservices eine manuelle Beschreibung nur auf der Serviceebene sinnvoll. Zusätzlich fehlen konkrete Hilfestellungen bei den Entscheidungen, die für eine Microservice-Architektur zu treffen sind.

Aus diesem Grund wurde ein prototypisches Modell entwickelt, welches Projektbeteiligte bei diesen Entscheidungen unterstützt. Über qualitative Interviews und einen Expertenkommentar sollte herausgefunden werden, wie sich das Modell perspektivisch verbessern lässt.

Hierbei stellte sich heraus, dass einige Konzepte noch näher erläutert werden müssen. Auch kann das Modell sukzessive verfeinert werden. So können z.B. weitere Fragestellungen ergänzt und Auswirkungen auch auf den unteren Ebenen beschrieben werden. Laut einigen Teilnehmern transportiert das Modell aber grundsätzlich die wichtigen Aspekte auf niedrigem Detailgrad.

Um hier konkreter zu werden, können die Informationen zielgruppenspezifisch aufbereitet werden. So könnten bspw. für einen Projektmanager mehr Informationen auf der organisatorischen Ebene bereitgestellt werden, während ein Entwickler mit Schwerpunkt Betrieb eher infrastrukturelle Fragestellungen und Auswirkungen benötigt.

Für eine detaillierte Darlegung der Sichten sind allerdings Befragungen mit weiteren Rollen in einer repräsentativen Anzahl notwendig, welche im Rahmen dieser Arbeit nicht durchgeführt wurden. Dies könnte eine weitere interessante Forschungsarbeit darstellen, um noch weitere relevante Informationen einer Microservice-Architektur zu erhalten.

ANHANG A - Umfrage

Umfrage: Modellierung von Microservices

UNIVERSITY OF APPLIED SCIENCES, Campus02 Graz, Austria

Ziel:
Die Umfrage ist Teil der Masterarbeit zum Thema „Modellierung von Microservices“ an der FH Campus02 in Graz. Ziel dieser Arbeit ist es herauszufinden, welche Aspekte für eine Microservice-Architektur von Relevanz sind und als Modell sichtbar gemacht werden sollten.

Empfängergruppe:
Die Umfrage richtet sich an alle, die in einem Projekt mit einer Microservice-Architektur mitwirken oder mitgewirkt haben.

Datenschutz:
Die Datenerhebung erfolgt anonym. Persönliche Daten zum Unternehmen oder zu Personen werden nicht erhoben. Die Ergebnisse werden am Ende veröffentlicht.

Kontakt:
olcay.tuemce@edu.campus02.at

* Erforderlich

Allgemeine Informationen

1.) Größe des Unternehmens *

- ◯ < 20 Mitarbeiter
- ◯ 21-50 Mitarbeiter
- ◯ 51-500 Mitarbeiter
- ◯ > 500 Mitarbeiter

Makro- und Mikroarchitektur

Bei der Makroarchitektur werden Entscheidungen getroffen, die das gesamte System betreffen. Im Gegensatz dazu werden bei der Mikroarchitektur die Entscheidungen den Teams überlassen.
Zur Makroarchitektur zählen Infrastruktur und Querschnittsfunktionalitäten eines Systems. Übergeordnete Regeln und Richtlinien des Unternehmens können ebenfalls Vorgaben der Makroarchitektur sein.

2.) Gibt es eine definierte Makroarchitektur? *

- ◯ Ja
- ◯ Nein

3.) Falls ja, wie wird diese durchgesetzt?

Meine Antwort

4.) Welche Aufgaben werden der Makroarchitektur zugeordnet?

- [ ] Monitoring
- [ ] Logging
- [ ] Resilienz
- [ ] Authentifizierung
- [ ] Autorisierung
- [ ] Programmiersprache / Plattform
- [ ] Deployment
- [ ] Service Konfiguration
- [ ] Skalierung
- [ ] Kommunikationsprotokolle
- [ ] Service Discovery
- [ ] Service Integration
- [ ] Integrationstests
- [ ] Sonstiges: _____

5.) Gibt es zusätzlich Regeln für die Makroarchitektur? *

- ○ Ja
- ○ Nein

6.) Falls ja, welche Regeln?

Meine Antwort

7.) Welche Entscheidungen können auf Mikroebene getroffen werden? *

Meine Antwort

8.) Falls sich Verantwortlichkeiten zwischen Mikro- und Makroarchitektur überschneiden, welche sind das und wie überschneiden sie sich?

Meine Antwort

Fachliche Architektur

Betrifft die fachliche Aufteilung der Services und das Architekturmanagement.
Fachliche Aufteilung: Vorgehensweise um die Servicegrenzen zu bestimmen.
Architekturmanagement: Das Sammeln und Verwalten von Daten, um einen Gesamtüberblick der Architektur zu erhalten.

9.) Gibt es eine bestimmte Vorgehensweise, um die Services fachlich aufzuteilen? *

- ◯ Ja
- ◯ Nein

10.) Welche Vorgehensweise wird für die fachliche Aufteilung verwendet?

- ◯ Domain-driven Design
- ◯ Sonstiges:

11.) Werden Daten für ein Architekturmanagement automatisiert gesammelt und visualisiert? *

- ◯ Ja
- ◯ Nein

12.) Falls ja, wie werden diese Daten gesammelt?

Meine Antwort

13.) Welche Daten werden gesammelt?

- [ ] Generelle Informationen der Services (z.B. Name, Besitzer)
- [ ] Gesamtanzahl der Services
- [ ] Anzahl der Instanzen je Service
- [ ] Location der Services
- [ ] Abhängigkeiten zwischen den Services
- [ ] Angebotene und verwendete Schnittstellen der Services
- [ ] Zusammenhängender fachlicher Prozess mehrerer Services
- [ ] Kommunikationsart (synchron, asynchron)
- [ ] Kommunikationsprotokolle
- [ ] Datenformate
- [ ] Bedarf an System-Ressourcen (z.B. CPU, Netzwerk)
- [ ] Anzahl der Anfragen
- [ ] Art der Anfragen (z.B. persistenz-bezogene Anfrage)
- [ ] Health-Status der Anfragen
- [ ] Latenzen der Anfragen
- [ ] Fehler und Timeouts
- [ ] Sonstiges: _____

SENDEN

Geben Sie niemals Passwörter über Google Formulare weiter.

ANHANG B - Modell

Ziel:

Das Interview ist Teil der Masterarbeit zum Thema „Modellierung von Microservices“ an der Fachhochschule der Wirtschaft Campus02 in Graz und der oose Informatik GmbH. Ziel dieser Arbeit ist es, herauszufinden, welche Aspekte für eine Microservice-Architektur von Relevanz sind und als Modell sichtbar gemacht werden sollten.

Zielgruppe sind Personen, welche ein Projekt mithilfe einer Microservice-Architektur umsetzen wollen. Anhand der Interviews soll herausgefunden werden, ob die bereitgestellten Informationen die Personen ausreichend bei der Umsetzung unterstützen.

Datenschutz:

Eine Datenschutzvereinbarung wird in einem separaten Dokument vor Beginn des Interviews übermittelt.

Modellbeschreibung:

Das vorliegende Modell stellt die wichtigen Entscheidungen dar, welche in einer Microservice-Architektur getroffen werden müssen. Hierfür wurden die Entscheidungen in verschiedene Ebenen aufgeteilt (Abbildung 1). Für jede Ebene wurde eine Tabelle aufbereitet, welche die Entscheidungen inklusive Beschreibung und Beispielen bereithält. Auf der Organisationsebene und Ebene der fachlichen Architektur wurden zusätzlich die möglichen Auswirkungen auf weitere Ebenen beschrieben. Für die Erläuterung einzelner Punkte wurden weitere Tabellen ergänzt, welche Informationen aus der Literatur und einer im Rahmen dieser Arbeit durchgeführten empirischen Analyse beinhalten.

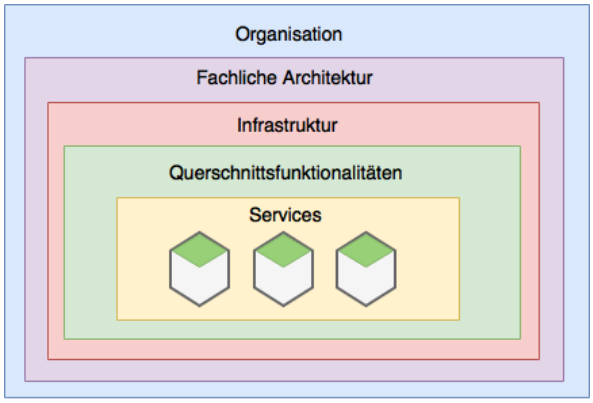

Abbildung 1: Entscheidungsebenen einer Microservice-Architektur

Organisation:

Entscheidung	Beschreibung	Auswirkung
Sind die Voraussetzungen für eine Microservice-Architektur erfüllt?	Um mit Microservices beginnen zu können, sollten die organisatorischen und infrastrukturellen Rahmenbedingungen geprüft werden (Tabelle 1).	Die Entscheidung für einen unpassenden Architekturansatz kann negative Auswirkungen auf alle weiteren Entscheidungen haben.
Wird mit einer Microservice-Architektur oder einem Monolithen gestartet?	Es besteht die Möglichkeit direkt mit einer Microservice-Architektur zu beginnen oder einen Monolithen im Anschluss zu zerlegen.	Der direkte Einstieg mit Microservices kann sich aufgrund der Komplexität schwieriger gestalten. Auch die Refaktorisierung ist bei Microservices aufwendiger. Auf der anderen Seite kann die späte Aufteilung eines Monolithen ebenfalls mit hohem Aufwand verbunden sein.
Was darf auf Makro- und was auf Mikroebene entschieden werden?	In einer Microservice-Architektur werden übergreifende Entscheidungen getroffen, welche alle Services betreffen (Makroarchitektur), und individuelle Entscheidungen auf Serviceebene (Mikroarchitektur) (Tabelle 2). Die Trennlinie zwischen den Ebenen ist nicht klar definiert, weshalb der Übergang oft fließend ist.	Tabelle 3 zeigt Vorteile von Standardisierung und individuellen Lösungen.
Wie wird die Makroarchitektur durchgesetzt?	In der Praxis gibt es verschiedene Methoden die Makroarchitektur durchzusetzen. Tabelle 4 zeigt mögliche Methoden inklusive Beispielen.	Halten sich Teams nicht an die Vorgaben, kann das negative Auswirkungen auf die Qualität haben. Beispiel: Ein neuer Service wird nicht an das standardisierte Monitoring angebunden und erschwert die Fehlersuche.
Wer verantwortet die Makroarchitektur?	In der Praxis gibt es klassische Architekten oder andere Führungsrollen, welche die Makroarchitektur verantworten. Oft werden dabei die Teams mit einbezogen.	Das Mitwirken der Serviceteams an der Gesamtarchitektur kann sich positiv auf die Produktivität auswirken. Hierzu ist es notwendig, dass sich die Teams das nötige Architekturwissen aneignen.

Voraussetzung	Beschreibung
Schnelle Bereitstellung von Ressourcen	Server müssen in kurzer Zeit hinzugefügt werden können. Hierfür wird langfristig ein hoher Grad an Automatisierung benötigt.
Überwachung der Systemlandschaft	Um in einer komplexen Umgebung schnell Fehler identifizieren zu können, muss ein umfangreiches Monitoring erfolgen, welches ggf. für verschiedene Zielgruppen Daten bereitstellt.
Schelle Bereitstellung der Anwendung	Die Anwendung muss in kurzer Zeit mit Hilfe einer Deployment-Pipeline bereitgestellt werden können.
Organisatorische Voraussetzungen	In einer Microservice-Architektur ändert sich die Organisationstruktur. Es entstehen crossfunktionale Teams, die nach Fachlichkeit getrennt sind und neben der Entwicklung gleichzeitig für den Betrieb zuständig sind.

Tabelle 1: Rahmenbedingungen einer Microservice-Architektur, angelehnt an Fowler (2014c)

Aufgaben Makroarchitektur	Richtlinien Makroarchitektur	Mikroarchitektur
Service Discovery	API-Richtlinien	Programmiersprache
Monitoring	Dokumentation	Datenbank
Authentifizierung	Developer-Guide	Frameworks und Bibliotheken
Logging	Höhere Testabdeckung	Alles andere
Deployment	Asynchrone Kommunikation	Service-Design
Kommunikationsprotokolle	Keine Businesslogik auf Makroebene	Technologie-Stack
Autorisierung	Einhaltung von Budgets	Deployment und Betrieb
Service Konfiguration	Kundenorientierung	Tests- und Testumgebung
Service Integration	Container und eine DB je Service	Funktionalität

Tabelle 2: Makro- und Mikroarchitektur in der Praxis

Mehr Standardisierung	Individuelle Lösungen
Entwickler können leichter zwischen den Teams wechseln, Funktionalität zwischen den Services kann aufgrund der gleichen Technologie leichter verschoben werden.	Es können optimale Lösungen für spezifische Probleme eingesetzt werden.
Es ist leichter sich auf Applikationsspezifika (z.B. Fachlichkeit) zu konzentrieren.	Neue Trends können schneller umgesetzt werden.
Durch erprobte Konzepte in kritischen Bereichen können Fehler vermieden werden.	Fehlentscheidungen haben eine geringere Relevanz, da sie sich nur auf Serviceebene auswirken.
Es entstehen geringere Kosten, dadurch dass weniger Produkte benötigt werden.	Es gibt eine geringere Abhängigkeit von einzelnen Lieferanten.

Tabelle 3: Vorteile von Standardisierung und individuellen, angelehnt an Zörner (2016)

Methode	Beispiele
Dokumentation	Allgemeine Regeln, Architektur-Richtlinien, Definition of Done
Kommunikation	Abstimmung zwischen den Teams, Schulungen, Coaching, Meetings, Austauschgruppen
Reviews	Peer Review, Quality Gates
Vorlagen	Blueprints, Referenz-Architekturen, Templates
Infrastruktur	Über die Build Pipeline mit vordefinierten Gradle-Tasks, Bereitstellung von Infrastrukturkomponenten wie Monitoring und Logging
Verantwortliche	(Enterprise)-Architekten, Architekturgremium, Team-Leads, Software-Platform-Director, Spezielles Dev-Ops-Team, CTO, Entwicklerteams

Tabelle 4: Durchsetzung der Makroarchitektur

Fachliche Architektur:

Entscheidung	Beschreibung	Auswirkung
Nach welcher Vorgehensweise werden die Services fachlich aufgeteilt?	Viele Unternehmen verwenden Domain-driven Design, um ihre Services fachlich aufzuteilen. Hierbei finden die verschiedenen Konzepte des strategischen und technischen Designs Verwendung.	Eine erprobte Vorgehensweise hilft dabei die Anwendungsdomäne sauber für alle Projektbeteiligten zu modellieren.
Wie wird die Granularität der Services bestimmt?	Die Granularität der Services kann sich bspw. an den identifizierten Bounded Contexts oder Aggregaten orientieren. Zur Unterstützung können Komplexität und Kommunikation der Services betrachtet werden. Eine Möglichkeit Services grobgranular zu strukturieren, sind Self Contained Systems.	Tabelle 5 zeigt die Auswirkungen von kleinen und großen Services auf die Architektur im Vergleich.
Welche Abhängigkeiten bestehen zwischen den Services?	Aufgrund gemeinsamer Schnittstellen müssen die Abhängigkeiten zwischen den Services explizit gemacht werden. Tabelle 6 zeigt hierfür die Konzepte von Domain-driven Design und deren Anwendbarkeit auf Microservices.	Starke Abhängigkeiten haben negative Auswirkungen auf die Qualitätsmerkmale der Architektur. Sie erhöhen bspw. das Ausfallrisiko, erschweren die Testbarkeit und führen zu mehr Koordination zwischen den Teams.

Große Services	Kleine Services
Geringe Flexibilität	Hohe Flexibilität
Geringe Kommunikation	Hohe Kommunikation
Geringe Skalierbarkeit	Hohe Skalierbarkeit
Geringe Ersetzbarkeit	Hohe Ersetzbarkeit
Geringer infrastruktureller Aufwand	Hoher infrastruktureller Aufwand
Hohes Ausfallrisiko des Systems	Geringeres Ausfallrisiko des Systems

Tabelle 5: Vergleich der Service-Granularität, angelehnt an Wolff (2016b) *und Ghadir* (2016)

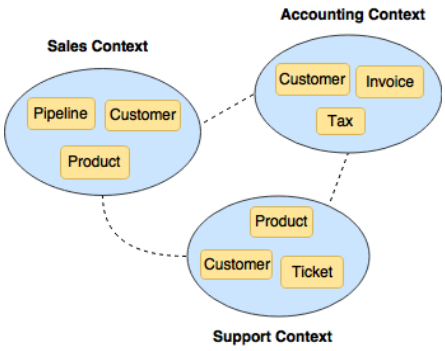

Abbildung 2: Strategisches Design mit Context-Maps, angelehnt an Vernon (2017)

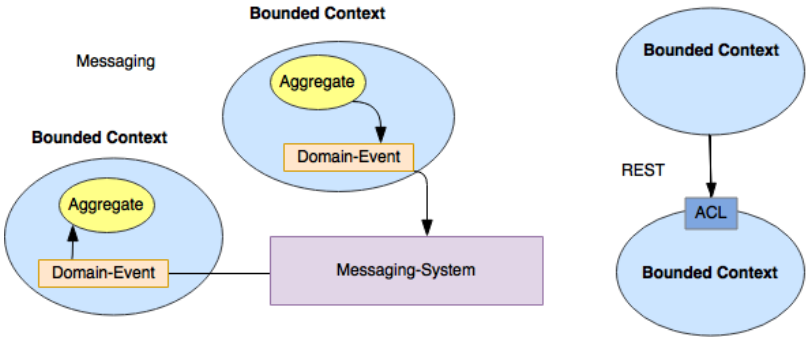

Abbildung 3: Integration mit Domain-Events und Context-Relationships, angelehnt an Vernon (2017)

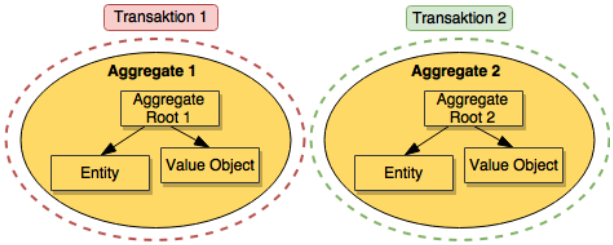

Abbildung 4: Taktisches Design mit Aggregaten, angelehnt an Vernon (2017)

Konzept	Anwendbarkeit
Bounded Context	Aufteilung der Fachlichkeit nach Bounded Context, um die Unabhängigkeit der Services zu gewährleisten. Jeder Bounded Context kann als eigener Service implementiert werden, wird von einem Team betreut und beinhaltet ein eigenes Datenmodell.
Context Map	Stellt die verschiedenen Bounded Context mit ihren Beziehungen dar und beschreibt somit die Interaktionen und Abhängigkeiten zwischen den Services (Abbildung 2).
Context Relationships	Beschreiben die konkreten Abhängigkeitsformen des Systems (Abbildung 3). Teilen sich die Services Teile des Domänenmodells mit einem Shared Kernel bspw. auf Datenebene, führt dies zu einer hohen Abhängigkeit der Services und demzufolge zu einer höheren Koordination zwischen den Teams.
Building-Blocks	Unterstützen bei der internen Strukturierung der Services. Hierbei gruppieren Aggregate die Entitäten und Value-Objects in fachliche Teilbereiche und bilden eine transaktionelle Konsistenzgrenze (Abbildung 4). Aggregate können somit als Untergrenze von Microservices verwendet werden. Aggregate referenzieren sich gegenseitig über Identitäten und verwenden Domain-Events, um Eventually Consistency zu erreichen (Abbildung 3).

Tabelle 6: Konzepte des Domain-driven Design, angelehnt an Wolff (2016b) *und Plöd* (2016)

Infrastruktur:

Entscheidung	Beschreibung
Wie werden Legacy-Systeme integriert?	Die Integration kann bspw. über einen Message-Broker erfolgen, der die Nachrichten an die entsprechenden Systeme verteilt. Eine weitere Möglichkeit ist der Zugriff über einen Anticorruption-Layer, über den eine Remote-API aufgerufen, die Datenbank synchronisiert oder direkt auf die Daten des Legacy-Systems zugegriffen werden kann.
Welche Produkte werden verwendet?	Es gibt bereits eine Vielzahl an Produkten, welche zentrale Aufgaben einer Microservice-Architektur übernehmen können. Tabelle 7 zeigt eine Zusammenfassung von verfügbaren Frameworks, Tools und Bibliotheken.
Auf welcher Plattform laufen die Services?	Es muss entschieden werden, ob auf einer eigenen Plattform oder einer fertigen IaaS-, CaaS- oder PaaS-Lösung aufgesetzt wird.
Wie werden die Services deployed?	Es müssen Entscheidungen bezüglich der Deployment-Art (Tabelle 8), des Aufbaus einer Deployment-Pipeline und erweiterter Deployment-Szenarien wie A/B-Testing und Canary-Releases getroffen werden.
Wie werden die Services konfiguriert?	Es muss festgelegt werden, ob die Konfiguration während des Deployments oder zur Laufzeit stattfindet. Für die Konfiguration zur Laufzeit wird eine Infrastrukturkomponente benötigt. Werden Konfigurationen mehrmals am Tag geändert, können Konfigurationsdatenbanken verwendet werden.
Wie erfolgt die Skalierung?	Da Microservices einzeln skaliert werden können, muss entschieden werden welche Systemteile auf welche Instanzen verteilt werden müssen. Für die Skalierung können z.B. proxybasierte Load-Balancer, client-seitige Loadbalancer oder Load-Balancer im Zusammenspiel mit einer Service-Discovery eingesetzt werden. Zentrale Loadbalancer sollten vermieden werden, da sie einen Single-Point-Of-Failure darstellen.
Wie erfolgt der Zugriff auf das System?	Der Einstieg in eine Microservice-Architektur wird in der Regel über Edge-Services realisiert. Edge-Services stellen zusätzliche Funktionen wie Load-Balancing, Caching, Reverse Routing, SSL-Terminierung und Service-Integration bereit (Abbildung 5). Zusätzlich können mit Techniken wie Edge Side (ESI) oder Server Side Includes (SSI) Referenzen aufgelöst werden, damit Nutzer zusammengesetzte Antworten von mehreren Services erhalten können.
Wie werden die Services integriert?	Services können auf Benutzerschnittstellen-Ebene integriert werden, bspw. mit jeweils eigenem Userinterface (UI), einem übergeordneten UI-Rahmen oder gänzlich ohne UI. Weitere Möglichkeiten sind eine Integration auf Logikebene bspw. mit REST, SOAP, RPC oder Messaging oder die Integration auf Datenbankebene. Letzteres führt zu einer starken Abhängigkeit zwischen den Services.
Wie werden die Services lokalisiert?	Die Lokalisierung der Services erfolgt in der Regel über eine Service Registry mit einer clientseitigen oder serviceseitigen Discovery (Abbildung 5). Zusätzlich kann ein API-Gateway verwendet werden, welches die Anfragen der Clients an die entsprechenden Services routet.
Wie kommunizieren die Services untereinander?	Es muss entschieden werden, wie die Inter-Prozesskommunikation zwischen den Services erfolgt und welche Protokolle und Datenformate dabei verwendet werden (Abbildung 6). Viele synchrone Aufrufe zwischen Services können zu einer hohen Latenz führen und sollten daher vermieden werden. Tabelle 9 zeigt die verschiedenen Interaktionsmöglichkeiten inklusive Beispielprotokolle.
Wie werden die Schnittstellen getestet?	Es muss gewährleistet werden, dass Servicenutzer auch bei Änderungen an den Schnittstellen die Services weiterverwenden können. Eine weitverbreitete Lösung hierfür sind Consumer-Driven-Contracts.
Wie wird Konsistenz oder Verfügbarkeit erreicht?	In verteilten Anwendungen muss nach dem CAP-Theorem zwischen Konsistenz und Verfügbarkeit entschieden werden. Damit eine strikte Erreichung der Konsistenz nicht zu Lasten der Verfügbarkeit geht, wird oft eine schwache Konsistenzform (eventual consistency) gewählt.

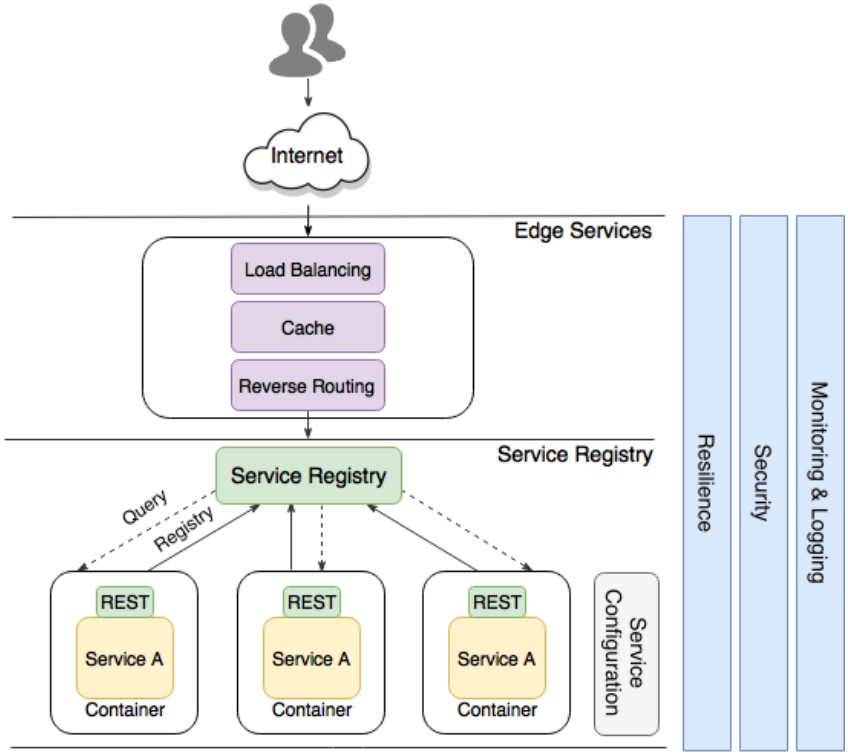

Abbildung 5: Beispiel einer Microservice-Infrastruktur, angelehnt an Richardson (2015) und Vitz (2016a)

Muster	Produkt
API Gateway	AWS, Netflix, NGINX
Circuit Breaker	Hystrix, Akka
Load Balancing & Service Discovery	NGINX, Ribbon, ELB, Eureka, etcd, Zoekeeper, Marathon, Consul
Monitoring, Metrics & Logging	Docker, Hystrix, Lightbend, Marathon, Graphite, Grafana, Zipkin, Elastic-Stack,
Edge-Service	Apache HTTPD, Apache Traffic Server, HA Proxy, NGINX, Træflk, Varnish, Zuul
Konfiguration	Consul, Vault, Zookeeper, Keywhiz, etcd, Chef, Puppet

Tabelle 7: Technologien für Microservice-Architekturen, angelehnt an Vitz (2016) *und Montesi & Weber* (2016)

Deployment	Vorteile	Nachteile
Eine Service-Instanz pro Host	Keine Konflikte zwischen Ressourcenanforderungen oder Abhängigkeiten der Versionen. Einfache Auslieferung, Verwaltung und Überwachung.	Verwaltung mehrerer Server und daher steigende Kosten beim Hinzufügen neuer Ressourcen.
Mehrere Service-Instanzen pro Host	Effizientere Ressourcennutzung als bei einzelnen Instanzen.	Mögliche Konflikte zwischen Ressourcenanforderungen oder Abhängigkeiten der Versionen. Da sich die Services gegenseitig beeinflussen, wird die Unabhängigkeit beeinträchtigt.
Service-Instanz pro Container	Bessere Performance als virtuelle Maschinen, z.B. beim Starten eines Containers oder Bauen eines Paketes.	Mehr Aufwand beim Aufbau gegenüber serverlosen Ansätzen.
Serverloses Deployment	Fertige Deployment-Infrastruktur. Nutzung optimierter Dienste.	Einschränkung von Programmiersprachen. Kontrolle wird teilweise abgegeben.

Tabelle 8: Deployment-Varianten, angelehnt an Newman (2015)

	one-to-one	one-to-many	Protocols (e.g)
synchronous	Request/response	-	HTTP (REST, Thrift)
asynchronous	Notification	Publish/subscribe	AMQP, STOMP, MQTT
	Request/async response	Publish/async responses	

Tabelle 9: Interaktionsmöglichkeiten, angelehnt an Richardson & Smith (2016)

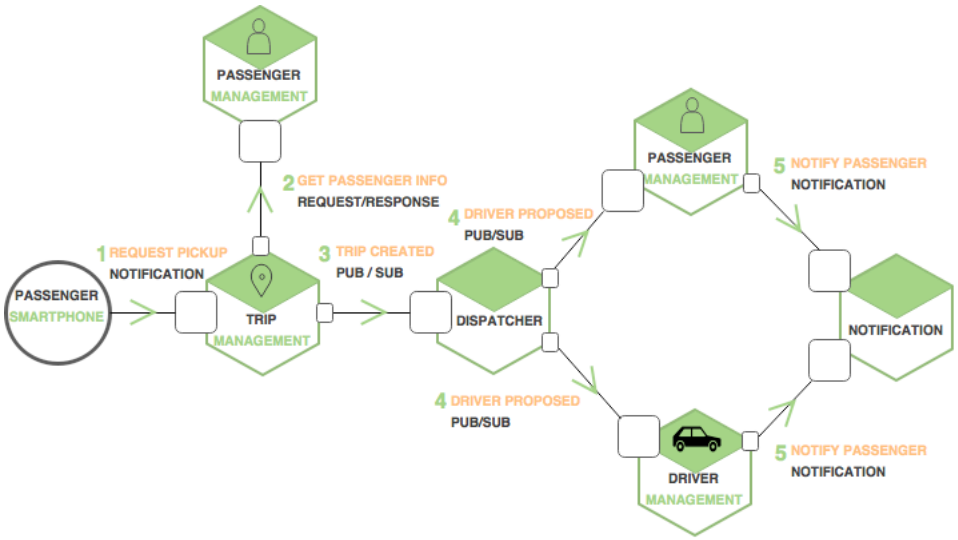

Abbildung 6: Interprozess-Kommunikation (Richardson & Smith, 2016)

Querschnittsfunktionalitäten:

Entscheidung	Beschreibung
Wie werden servicerelevante Informationen gesammelt und dargestellt?	Das Sammeln und Darstellen von servicerelevanten Informationen ist essentiell, um sich einen Überblick zu verschaffen. Hierzu muss entschieden werden, welche Daten (Tabelle 10) für welche Zielgruppen über welchen Kanal (Tabelle 11) gesammelt werden. Abbildung 7 zeigt ein Beispiel einer Monitoring Landschaft mit Grafana, Graphite und Zipkin.
Wie erfolgen Authentifizierung und Autorisierung?	Es wird ein Sicherheitskonzept benötigt, um die vielen verteilten Services vor unberechtigten Zugriffen zu schützen. Als technische Umsetzung kann z.B. das OAuth2-Protokoll mit diversen Ansätzen oder als Alternative Kerberos, SAML und SAML 2.0 verwendet werden.
Wie wird mit Fehlern umgegangen?	Da Ausfälle in einem verteilten System nicht ausgeschlossen werden können, muss der Fokus auf Fehlerbehandlung anstelle von Vermeidung liegen. So können bei Ausfällen z.B. Weiterleitungen auf reduzierte Services eingerichtet werden. Mithilfe des Circuit-Breaker-Patterns kann zusätzlich verhindert werden, dass Nutzer weiterhin auf ausgefallene Services geroutet werden.
Wie teilen sich die Services gemeinsame Daten?	Häufig müssen sich Services bspw. aufgrund eines gemeinsamen fachlichen Prozesses Daten teilen. Für eine lose Koppelung unterstützen hier event-getriebene Ansätze. Um Inkonsistenzen z.B. bei Ausfällen zu vermeiden, kann zusätzlich Event-Sourcing verwendet werden.

Informationen
Health-Status der Anfragen
Generelle Informationen der Services (z.B: Name, Besitzer, Version)
Gesamtzahl der Services
Latenzen der Anfragen
Anzahl der Instanzen je Services
Fehler und Timeouts
Location der Services
Angebotene und verwendete Schnittstellen
Anzahl der Anfragen
Bedarf an Systemressourcen
Abhängigkeiten zwischen den Services
Datenformate
Kommunikationsprotokolle
Zusammenhängender fachlicher Prozess mehrerer Services
Kommunikationsart (synchron / asynchron)
Art der Anfragen (z.B. Persistenz bezogene Anfrage)
Unterschiede der Schnittstellen über mehrere Versionen hinweg
Technologie-Stack der Services
Information Security (infosec)
Kritikalität

Tabelle 10: Datensammlung aus der Praxis

Kanal	Beispiel
Monitoring	Grafana
Logging	ELK-Stack
Service Discovery	Consul
API-Management	Swagger
Repository-Files	Sammeln von YAML Files
Deployment	Deployment-Deskriptoren
Fehlerbehandlung	Hystrix
Tracing	Zipkin
Modellierung	UML

Tabelle 11: Kanäle der Datensammlung

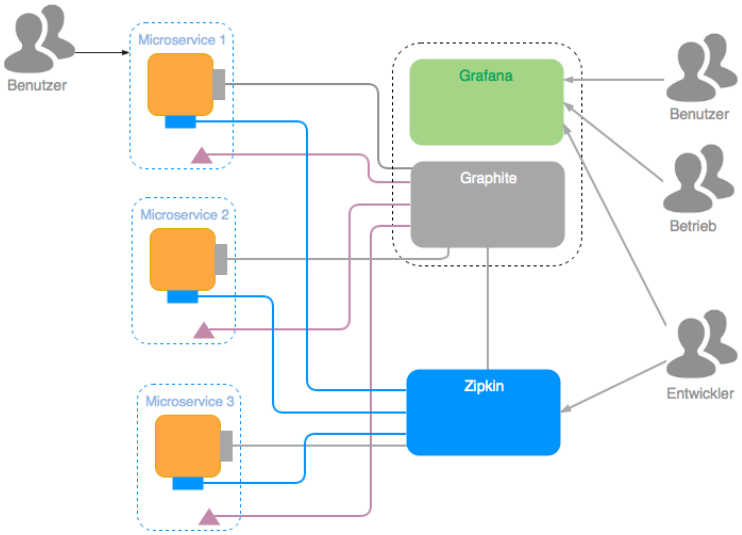

Abbildung 7: Monitoring mit Grafana, Graphite und Zipkin, (Fichtner, 2016)

ANHANG C - Digitale Form

Umfrage:

Ergebnisse der Umfrage (Excel)

Interview:

Interviewprotokolle 1-6 (PDF)

Expertenkommentar (PDF)

Datenschutzvereinbarungen (PDF)

Interviewleitfaden (PDF)

ABBILDUNGSVERZEICHNIS

TABELLENVERZEICHNIS

LITERATURVERZEICHNIS

Abbott, M., & Fischer, M. (2015). *The Art of Scalability.* Boston: Addison-Wesley.

Adersberger, J., Siedersleben, J., & Weigend, J. (Nov/Dez 2016). Robustheit und Antifragilität: Eignen sich Microservices für die Systeme der Zukunft? *OBJEKTSpektrum*, 45-49.

Aichinger, P. (2017). *Ein Dashboard für die dynamische Dokumentation microservice-basierter Softwaresysteme.* Österreich: Johannes Kepler Universität Linz.

Annenko, O. (08. 06. 2016). *Breaking Down a Monolithic Software: A Case for Microservices vs. Self-Contained Systems*. Abgerufen am 08. 12. 2016 von elastic.io: https://www.elastic.io/breaking-down-monolith-microservices-and-self-contained-systems/

Apple, L. (05. 02. 2016). *Zalando Tech's Rules of Play*. Abgerufen am 22. 01. 2017 von GitHub: https://github.com/lmineiro/zalando-rules-of-play

Bass, L., Clements, P., & Kazman, R. (2013). *Software Architecture in Practice* (3. Ausg.). New Jersey: Pearson Education, Inc.

Becker, J., Probandt, W., & Vering, O. (2012). *Grundsätze ordnungsgemässer Modellierung - Konzeption und Praxisbeispiel für ein effizientes Prozessmanagement.* Springer Verlag.

Bernroider, E., & Stix, V. (2006). *Grundzüge der Modellierung.* Wien: Facultas Verlags- und Buchhandels AG.

Bonér, J. (2016). *Reactive Microservices Architecture - Design Principles for Distributed Systems* (1. Ausg.). California: O'Reilly Media, Inc.

Bryant, D. (03. 02. 2016). *AutoScout24's Journey to Microservices: Christian Deger on Transformation, Principles and Technology*. Abgerufen am 12. 03. 2017 von InfoQ: https://www.infoq.com/news/2016/02/autoscout-microservices

Calçado, P. (11. 06. 2014). *Building Products at SoundCloud —Part I: Dealing with the Monolith*. Abgerufen am 08. 12. 2016 von SoundCloud: https://developers.soundcloud.com/blog/building-products-at-soundcloud-part-1-dealing-with-the-monolith

Conway, M. E. (04. 1968). *How Do Committees Invent?* Abgerufen am 05. 10. 2016 von Mel Conway's Home Page: http://www.melconway.com/research/committees.html

Dragoni, N., Giallorenzo, S., Lluch Lafuente, A., Mazzara, M., & Montesi, F. (13. 06. 2016). *Microservices: yesterday, today, and tomorrow.* Abgerufen am 06. 10. 2016 von Cornell University Library: https://arxiv.org/abs/1606.04036

Evans, E. (2003). *Domain-Driven Design: Tackling Complexity in the Heart of Software.* Boston: Addison-Wesley.

Fichtner, V. (2016). Microservices brauchen ein neues Monitoring. *Entwickler Spezial - Microservices die andere Art der Modularisierung*, 62-68.

Fildebrandt, U. (2016). Solide Microservices. *Entwickler Magazin Spezial*, 35-39.

Flohre, T. (01. 09. 2015). *Wer Microservices richtig macht, braucht keine Workflow Engine und kein BPMN*. Abgerufen am 15. 03. 2017 von codecentric: https://blog.codecentric.de/2015/09/wer-microservices-richtig-macht-braucht-keine-workflow-engine-und-kein-bpmn/

Fowler, M. (29. 06. 2004). *StranglerApplication*. Abgerufen am 22. 11. 2016 von martinfowler.com: https://www.martinfowler.com/bliki/StranglerApplication.html

Fowler, M. (01. 12. 2008). *HumaneRegistry*. Abgerufen am 12. 03. 2016 von martinfowler.com: http://martinfowler.com/bliki/HumaneRegistry.html

Fowler, M. (13. 08. 2014a). *Microservices and the First Law of Distributed Objects*. Abgerufen am 02. 10. 2016 von martinfowler.com: http://martinfowler.com/articles/distributed-objects-microservices.html

Fowler, M. (20. 10. 2014b). *SacrificialArchitecture*. Von martinfowler.com: https://martinfowler.com/bliki/SacrificialArchitecture.html abgerufen

Fowler, M. (28. 08. 2014c). *MicroservicePrerequisites*. Abgerufen am 15. 06. 2017 von martinfowler.com: https://martinfowler.com/bliki/MicroservicePrerequisites.html

Fowler, M. (23. 06. 2015). *MonolithFirst*. Abgerufen am 26. 04. 2017 von martinfowler.com: https://martinfowler.com/bliki/MonolithFirst.html

Fowler, M., & Lewis, J. (24. 03. 2014). *Microservices - a definition of this new architectural term*. Abgerufen am 02. 10. 2016 von martinfowler.com: http://www.martinfowler.com/articles/microservices.html

Fowler, S. (2017). *Production-Ready Microservices* (1. Ausg.). California: O'Reilly Media, Inc.

Franz, T. (2016). Wie schneide ich richtig? *Entwickler Spezial - Microservice die andere Art der Modularisierung*, 32-34.

Franz, T. (01. 2017). Microservices erzeugen lediglich andere Schulden: Wieso konzeptuelle Modelle eine noch gewichtigere Rolle in Microservice-Architekturen spielen. *OBJEKTspektrum*, 62-66.

Freudl-Gierke, T. (03. 11. 2014). *Micro Services in der Praxis: Nie wieder Monolithen!* Abgerufen am 22. 01. 2017 von JAXenter: https://jaxenter.de/micro-services-in-der-praxis-nie-wieder-monolithen-391

Friedrichsen, U. (31. 05. 2016). *Resilient Software Design – Robuste Software entwickeln*. Abgerufen am 21. 10. 2016 von Informatik Aktuell: https://www.informatik-aktuell.de/entwicklung/methoden/resilient-software-design-robuste-software-entwickeln.html

Ghadir, P. (01. 2016). *Wie kommt man zu Self-Contained Systems?* Abgerufen am 28. 04. 2017 von SIGS DATACOM: https://www.sigs-datacom.de/uploads/tx_dmjournals/ghadir_JS_01_16_6l5t.pdf

Gilbert, S., & Lynch, N. (2002). *Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-Tolerant Web Services.* Abgerufen am 09. 01. 2017 von glassbeam: http://www.glassbeam.com/sites/all/themes/glassbeam/images/blog/10.1.1.67.6951.pdf

Goldsmith, K. (12. 03. 2015). *Microservices at Spotify*. Abgerufen am 23. 01. 2017 von Kevin Goldsmith: http://www.kevingoldsmith.com/talks/

Gray, J. (30. 06. 2006). *A Conversation with Werner Vogels*. Abgerufen am 12. 03. 2017 von acmqueue: https://queue.acm.org/detail.cfm?id=1142065

Hassan, S., & Bahsoon, R. (01. 09. 2016). *Microservices and Their Design Trade-offs: A Self-Adaptive Roadmap.* Abgerufen am 13. 11. 2016 von IEEE Explore Digital Library: http://ieeexplore.ieee.org/document/7557535/

Heath, M. (09. 03. 2015). *A Journey into Microservices: Dealing with Complexity*. Abgerufen am 23. 01. 2017 von The Hailo Tech Blog: https://sudo.hailoapp.com/services/2015/03/09/journey-into-a-microservice-world-part-3/

Indrasiri, K. (09. 02. 2016). *Microservices in Practice: From Architecture to Deployment*. Abgerufen am 29. 11. 2016 von DZone: https://dzone.com/articles/microservices-in-practice-1

innoQ. (2015). *Self-Contained Systems - Assembling Software from independent Systems*. Abgerufen am 19. 10. 2016 von SCS: http://scs-architecture.org

ISO/IEC 9126-1:2001. (2001). *Information technology - Software product quality - Part 1: Quality model*. Abgerufen am 20. 09. 2016 von https://www.cse.unsw.edu.au/~cs3710/PMmaterials/Resources/9126-1%20Standard.pdf.

JAXenter. (22. 09. 2016). *Infografik: Software-Architektur-Trends 2016*. Abgerufen am 12. 03. 2017 von JAXenter: https://jaxenter.de/infografik-software-architektur-trends-2016-47014

Kansy, R., Lubkowitz, M., & Schäfer, M. (2016). Microservice-Entwicklung erfordert Kompromisse. *Entwickler Spezial*, 26-31.

Karcher, C. (22. 03. 2016). *ClusterHQ - The Container Data People*. Abgerufen am 08. 12. 2016 von Tech Talk: Modeling Microservices at Spotify with Petter Måhlén: https://clusterhq.com/2016/03/22/microservices-spotify-petter-mahlen/

Kosewski, L., Tatelman, J., Reynolds, J., & Rosenthal, C. (01. 10. 2015). *Flux: A New Approach to System Intuition*. Abgerufen am 06. 12. 2016 von The Netflix Tech Blog: http://techblog.netflix.com/2015/10/flux-new-approach-to-system-intuition.html

Krause, L. (2015). *Microservices Patterns and Applications - Define fine-grained services by applying patterns.* Amazon Digital Services LLC.

Larsson, M. (25. 03. 2015). *An operations model for Microservices*. Abgerufen am 20. 11. 2016 von callista-enterprise: http://callistaenterprise.se/blogg/teknik/2015/03/25/an-operations-model-for-microservices/

Linders, B. (14. 12. 2015). *Microservices at Spotify*. Abgerufen am 07. 12. 2016 von infoQ: https://www.infoq.com/news/2015/12/microservices-spotify

Lowe, S. A. (15. 10. 2015). *An introduction to event storming: The easy way to achieve domain-driven design*. Abgerufen am 25. 10. 2016 von TeachBeacon: http://techbeacon.com/introduction-event-storming-easy-way-achieve-domain-driven-design

Müller, F. (2015). *TWINTIP*. Abgerufen am 06. 12. 2016 von GitHub: https://github.com/fmueller/twintip-crawler

Müller, J. (2016). Microservices bei Eurospace. *Entwickler Spezial*, 92-94.

Martraire, C. (22. 06. 2016). *Decentralized Architecture: Issues in the Field*. Abgerufen am 12. 03. 2017 von DZone: https://dzone.com/articles/decentralized-architecture

Mauro, T. (19. 02. 2015). *Adopting Microservices at Netflix: Lessons for Architectural Design*. Abgerufen am 04. 12. 2016 von NGINX: https://www.nginx.com/blog/microservices-at-netflix-architectural-best-practices/

Messinger, L. (17. 02. 2013). *Better explaining the CAP Theorem*. Abgerufen am 09. 02. 2017 von DZone: https://dzone.com/articles/better-explaining-cap-theorem

Millet, S., & Tune, N. (2015). *Patterns, Principles, and Practices of Domain-Driven Design* (1. Ausg.). Indianapolis: John Wiley & Sons, Inc.

Montesi, F., & Weber, J. (21. 09. 2016). *Circuit Breakers, Discovery, and API Gateways in Microservices.* Abgerufen am 12. 04. 2017 von Cornell University Library: https://arxiv.org/abs/1609.05830

Nadareishvili, I., Mitra , R., McLarty , M., & Amundsen , M. (2016). *Microservice Architecture: Aligning Principles, Practices, and Culture.* Sebastopol: O'Reilly Media, Inc. .

Newman, S. (2015). *Microservices - Konzeption und Design* (1. Ausg.). Wachtendonk: mitp Verlags GmbH & Co. KG.

Parnas, D. L. (02. 1971). *Information distribution a spects of design methodology*. Abgerufen am 20. 07. 2017 von Research Showcase @ CMU: http://repository.cmu.edu/cgi/viewcontent.cgi?article=2828&context=compsci

Pelka, C., & Plöd, M. (2016). Microservices a la Netflix. *Entwickler Spezial*, 45-48.

Plöd, M. (2016). Microservices lieben Domain Driven Design. *Entwickler Spezial*, 22-25.

Posch, T., Birken, K., & Gerdom, M. (2011). *Basiswissen Software Architektur* (3. Ausg.). Heidelberg: dpunkt.verlag GmbH.

Preissler, J., & Tigges, O. (2015). *Docker – perfekte Verpackung von Microservices.* Abgerufen am 04. 10. 2016 von SIGS DATACOM: http://www.sigs-datacom.de/uploads/tx_mwjournals/pdf/preissler_tigges_OTS_Architekturen_15.pdf

Röwekamp, L., & Limburg, A. (09. 02. 2016). *Der perfekte Microservice*. Abgerufen am 08. 10. 2016 von heise online: https://www.heise.de/developer/artikel/Der-perfekte-Microservice-3091905.html

Reinhard, B. (06. 2011). *Business Capability Management - Gezielte Ausrichtung der Artefakte einer Unternehmensarchitektur.* Abgerufen am 02. 12. 2016 von reinhard.one: http://www.generate-value.com/wp-content/uploads/2013/02/Business_Capabilities_Boris_Reinhard_300_new.pdf

Reussner, R., & Hasselbring, W. (2009). *Handbuch der Software Architektur* (2. Ausg.). Heidelberg: dpunkt.verlag GmbH.

Reynolds, J., & Rosenthal, C. (03. 08. 2016). *Vizceral Open Source*. Abgerufen am 05. 12. 2016 von The Netflix Techblog: http://techblog.netflix.com/2016/08/vizceral-open-source.html

Richards, M. (2016). *Microservices vs. Service-Oriented Architecture* (1. Ausg.). California: O'Reilly Media, Inc.

Richardson, C. (18. 03. 2014a). *Microservice Architecture*. Abgerufen am 06. 06. 2017 von Microservice Architecture: http://microservices.io

Richardson, C. (2014b). *Pattern: Event sourcing*. Abgerufen am 19. 10. 2016 von Microservice Architecture: http://microservices.io/patterns/data/event-sourcing.html

Richardson, C. (2014c). *Pattern: Microservices Architecture*. Abgerufen am 02. 10. 2016 von Microservice Architecture: http://microservices.io/patterns/microservices.html

Richardson, C. (2014d). *Pattern: Single Service Instance per Host*. Abgerufen am 06. 10. 2016 von Microservice Architecture: http://microservices.io/patterns/deployment/single-service-per-host.html

Richardson, C. (08. 09. 2014e). *Pattern: API Gateway*. Abgerufen am 06. 10. 2016 von Microservice Architecture: http://microservices.io/patterns/apigateway.html

Richardson, C. (2014f). *Pattern: Service instance per container*. Abgerufen am 05. 10. 2016 von Microservice Architecture: http://microservices.io/patterns/deployment/service-per-container.html

Richardson, C. (2014g). *Pattern: Event-driven architecture*. Abgerufen am 11. 01. 2017 von Microservice Architecture: http://microservices.io/patterns/data/event-driven-architecture.html

Richardson, C. (2014h). *The Scale Cube*. Abgerufen am 22. 10. 2016 von Microservice Architecture: http://microservices.io/articles/scalecube.html

Richardson, C. (2014i). *Pattern: Multiple service instances per host*. Abgerufen am 05. 10. 2016 von Microservice Architecture: http://microservices.io/patterns/deployment/multiple-services-per-host.html

Richardson, C. (2014j). *Pattern: Serverless deployment*. Abgerufen am 11. 10. 2016 von Microservice Architecture: http://microservices.io/patterns/deployment/serverless-deployment.html

Richardson, C. (12. 10. 2015). *Service Discovery in a Microservices Architecture*. Abgerufen am 06. 10. 2016 von NGINX: https://www.nginx.com/blog/service-discovery-in-a-microservices-architecture/

Richardson, C. (03. 10. 2016). *Developing Transactional Microservices Using Aggregates, Event Sourcing and CQRS - Part 1*. Abgerufen am 22. 05. 2017 von InfoQ: https://www.infoq.com/articles/microservices-aggregates-events-cqrs-part-1-richardson

Richardson, C., & Smith, F. (2016). *Microservice - From Design to Deployment.* Abgerufen am 21. 11. 2016 von NGINX: https://www.nginx.com/resources/library/designing-deploying-microservices/?utm_source=microservices-from-design-to-deployment-ebook-nginx&utm_medium=blog&utm_campaign=Microservices

Riebisch, M., & Bode, S. (2009). *Software-Evolvability*. Abgerufen am 01. 07. 2017 von Gesellschaft für Informatik: https://www.gi.de/service/informatiklexikon/detailansicht/article/software-evolvability.html

Rotem-Gal-Oz, A. (2006). *Fallacies of Distributed Computing.* Abgerufen am 13. 11. 2016 von Wisconsin - University of Wisconsin-Madison: https://pages.cs.wisc.edu/~zuyu/files/fallacies.pdf

RV, R. (06. 2016). *A capability model for microservices*. Abgerufen am 22. 11. 2016 von www.packtpub.com: https://www.packtpub.com/books/content/capability-model-microservices

Schäfer, R. (20. 09. 2016). *GOTO 2016 – From Monolith to Microservices at Zalando*. Abgerufen am 22. 01. 2017 von Zalando: https://tech.zalando.com/blog/goto-2016-from-monolith-to-microservices/

Schlosser, H. (03. 02. 2016a). *Beim Übergang zu Microservices geht es nicht nur um einen Technologiewechsel*. Abgerufen am 08. 12. 2016 von JAXenter: https://jaxenter.de/beim-uebergang-zu-microservices-geht-es-nicht-nur-um-einen-technologiewechsel-34188

Schlosser, H. (09. 08. 2016b). *Microservice-Architekturen visualisieren: Ordina präsentiert neues Dashboard für Spring Boot*. Abgerufen am 02. 12. 2016 von JAXenter.de: https://jaxenter.de/spring-boot-microservices-44715

Schwartz, A. (2016). Baukasten für flexible und robuste Microservices. *Entwickler Spezial*, 82-87.

Soika, R. (19. 12. 2016). *Prozessmanagement in einer Microservices-Architektur – passt das zusammen?* Abgerufen am 15. 03. 2017 von JAXenter: https://jaxenter.de/prozessmanagement-microservices-50505

Stachowiak, H. (1973). *Allgemeine Modelltheorie.* Wien: Springer Verlag.

Starke, G. (2014). *Effektive Software Architekturen* (6. Ausg.). München: Hanser Verlag.

Starke, G., & Hruschka, P. (2016). *Ressourcen für Softwarearchitekten*. Abgerufen am 20. 08. 2016 von arc42: http://www.arc42.de/template/struktur.html

Starke, G., Simons, M., & Zörner , S. (2017). *arc42 by Example (eBook).* Lean Publishing.

Steinacker, G. (02. 07. 2015). *Von Monolithen und Microservices*. Abgerufen am 19. 10. 2016 von Informatik Aktuell: https://www.informatik-aktuell.de/entwicklung/methoden/von-monolithen-und-microservices.html

Stropek, R. (21. 12. 2016). *Microservices ohne Cloud und DevOps? Keine Chance!* Abgerufen am 12. 03. 2017 von entwickler.de: https://entwickler.de/online/windowsdeveloper/microservices-ohne-cloud-und-devops-579746561.html

The Open Group. (2016). *Microservices Architecture – SOA and MSA*. Abgerufen am 14. 03. 2017 von The Open Group: http://www.opengroup.org/soa/source-book/msawp/p3.htm

Tilkov, S. (2011). *REST und HTTP - Einsatz der Architektur des Web für Integrationsszenarien* (2. Ausg.). Heidelberg: dpunkt.verlag GmbH.

Tilkov, S. (09. 06. 2015). *Don't start with a monolith… when your goal is a microservices architecture*. Abgerufen am 18. 07. 2017 von martinfowler.com: https://martinfowler.com/articles/dont-start-monolith.html

Toth, S. (05. 08. 2015). *Netflix-Architektur Blogserie – Teil 2: Microservices*. Abgerufen am 05. 12. 2016 von embarc - Software Consulting GmbH: http://www.embarc.de/netflix-architektur-blogserie-teil-2-microservices/

Trelle, T. (29. 08. 2011). *Grundlagen Cloud Computing: CAP-Theorem*. Abgerufen am 10. 01. 2017 von codecentric: https://blog.codecentric.de/2011/08/grundlagen-cloud-computing-cap-theorem/

Tseitlin, A. (27. 06. 2013). *The Antifragile Organization - Embracing Failure to Improve Resilience and Maximize Availability*. Abgerufen am 05. 12. 2016 von acmqueue: http://queue.acm.org/detail.cfm?id=2499552

Vernon, V. (2017). *Domain-Driven Design kompakt* (1. Ausg.). Heidelberg: dpunkt.verlag GmbH.

Vitz, M. (04. 2016a). *Consumer-Driven Contracts – Testen von Schnittstellen innerhalb einer Microservices- Architektur.* Abgerufen am 01. 12. 2016 von SIGS DATACOM: http://www.sigs-datacom.de/uploads/tx_dmjournals/vitz_JS_04_16_TaTZ.pdf

Vitz, M. (06. 2016b). *Microservice-Infrastruktur.* Abgerufen am 10. 03. 2017 von SIGS DATACOM: https://www.sigs-datacom.de/uploads/tx_dmjournals/vitz_JS_06_16_5Xuz.pdf

Vogel, O., Arnold, I., Chughtai, A., Ihler, E., Mehlig, U., Neumann, T., . . . Zdun, U. (2005). *Software Architektur: Grundlagen - Konzepte - Praxis.* München: Elsevier GmbH.

Watson, C., Emmons, S., & Gregg, B. (18. 02. 2015). *A Microscope on Microservices*. Abgerufen am 05. 12. 2016 von The Netflix Techblog: http://techblog.netflix.com/2015/02/a-microscope-on-microservices.html

Wehrens, O. (15. 09. 2016). *How not to lose your mind with too many microservices - BedCon 2016*. Abgerufen am 06. 12. 2016 von Speaker Deck: https://speakerdeck.com/owehrens/how-not-to-lose-your-mind-with-too-many-microservices-bedcon-2016

Wehrens, O., & Gentsch, L. (2016). Microservices bei der E-Post. *Entwickler Spezial*, 95-97.

Weilkiens, T. (2008). *Systems Engineering mit SysML/UML* (2. Ausg.). Heidelberg: dpunkt.verlag GmbH.

Weilkiens, T., Weiss, C., Grass, A., & Duggen, K. (2015). *Basiswissen Geschäftsprozessmanagement* (2. Ausg.). Heidelberg: dpunkt.verlag GmbH.

Westheide, D. (06. 03. 2016). *Why RESTful communication between microservices can be perfectly fine*. Abgerufen am 20. 07. 2017 von InnoQ: https://www.innoq.com/de/blog/why-restful-communication-between-microservices-can-be-perfectly-fine/

Wolff, E. (09. 09. 2014). *Microservices im Zusammenspiel mit Continuous Delivery*. Abgerufen am 06. 10. 2016 von heise online: https://www.heise.de/developer/artikel/Microservices-im-Zusammenspiel-mit-Continuous-Delivery-Teil-1-die-Theorie-2376386.html

Wolff, E. (2016a). *Continuous Delivery - Der pragmatische Einstieg* (2. Ausg.). dpunkt.verlag GmbH.

Wolff, E. (2016b). *Microservices - Grundlagen flexibler Softwarearchitekturen* (1. Ausg.). Heidelberg: dpunkt.verlag GmbH.

Wolff, E. (Nov/Dez 2016c). Microservices - der aktuelle Stand: Was hinter dem Hype steckt und wie es weitergeht. *OBJEKTspektrum - Moderne Architekturen der neue Look der IT*, 40-44.

Wolff, E. (27. 01. 2017). *Meine Datenbank gehört mir! - Continuous Architecture*. Abgerufen am 20. 07. 2017 von heise online: https://www.heise.de/developer/artikel/Meine-Datenbank-gehoert-mir-3608372.html

Wolff, E., Schwartz, A., & Heusingfeld, A. (2016). *Microservices - Der Hype im Realitätscheck.* Frankfurt: entwickler.press.

Wootton, B. (08. 04. 2014). *Microservices - Not A Free Lunch!* Abgerufen am 12. 03. 2017 von High Scalability: http://highscalability.com/blog/2014/4/8/microservices-not-a-free-lunch.html

Zörner, S. (2012). *Softwarearchitekturen dokumentieren und kommunizieren.* München: Hanser Verlag.

Zörner, S. (12. 02. 2015). *Gut das ist? Die umgekehrte Architekturbewertung eines Internet-Giganten.* Abgerufen am 05. 12. 2016 von microxchg: http://microxchg.io/2015/slides/01_02_microXchg_2015_architekturbewertung_szoerner.pdf

Zörner, S. (2016). Bring your own Architecture. *Entwickler Spezial - Microservices die andere Art der Modularisierung*, 16-19.

Zuther, B. (11. 05. 2015). *Microservice-Deployment ganz einfach mit Docker Compose*. Abgerufen am 20. 05. 2017 von codecentric: https://blog.codecentric.de/2015/05/microservice-deployment-ganz-einfach-mit-docker-compose/

Zuther, B. (2016). Canary-Releases mit der Very Awesome Microservices Platform. *Java aktuell*(03), 25-27.