ulm university universität ulm

Fakultät Ingenieurwissenschaften, Informatik und Psychologie
Institut für Datenbanken und Informationssysteme

Bachelorarbeit
im Studiengang Medieninformatik

Microservices

Eine State-of-the-Art Bestandsaufnahme und Abgrenzung zu SOA

vorgelegt von

Ferdinand Birk

Dezember 2016

Gutachter:	Prof. Reichert, Manfred
Betreuer:	Dr. Pryss, Rüdiger
Matrikelnummer:	890638
Arbeit vorgelegt am:	13.12.2016

Kurzfassung

Microservices haben in den letzten Jahren einen Hype erfahren und gelten momentan als der Stil, der es einem Unternehmen erlaubt, möglichst agil und innovativ in der Entwicklung von Software zu sein. Durch die Entkopplung von fachlich nur wenig zusammenhängenden Teilen zu eigenständigen Microservices, kann eine Unterteilung der Entwicklung in hochgradig autonome Teams erreicht werden. Zusätzlich unterstützen Cloud-Technologien und Automatisierung durch Continuous Delivery und DevOps die Teams bei der schnellen Weiterentwicklung ihres abgegrenzten Bereiches und erlaubt es diesen, selbständig für den Betrieb ihrer Microservices verantwortlich zu sein.
Der Begriff *Microservices* ist noch relativ jung und zeigt sich in unterschiedlichen Formen, weshalb oft unklar ist, was er eigentlich genau bedeutet. Daraus ergeben sich auch Schwierigkeiten in der Abgrenzung zu anderen Ansätzen wie SOA.
Der Ansatz, Software durch Services zu modularisieren, ist nicht neu und wird schon bei SOA angewendet, um die Integration verschiedener Systeme zu erleichtern und diese schneller an veränderte Geschäftsprozesse anzupassen. Obwohl Microservices aus SOA heraus entstanden sind, fokussiert der Stil mehr auf die Flexibilität und Geschwindigkeit bei der Entwicklung von modernen Online-Softwaresystemen und setzt Service-Orientierung anders um als es bei SOA der Fall ist.

Diese Arbeit erfasst das Themengebiet unterteilt in Stil-Ebene, Architektur-Ebene und Service-Ebene. Sie versucht so, dabei zu helfen, mehr Verständnis über den Begriff *Microservices* zu erlangen. Dabei werden angrenzende Konzepte sowie Technologien erklärt und anhand von Charakteristiken beschrieben, was den Microservices-Stil ausmacht. Über veröffentlichte Berichte und Konferenzvorträge von Unternehmen, die nach eigenen Angaben Microservices umsetzen, werden typische Gründe und Motivationen für den Microservice-Stil herausgearbeitet. Zuletzt wird SOA kurz definiert und anhand dem aktuellen Verständnis eingeführt. Mit den erlangten Kenntnissen über Microservices und deren Motivation, wird ein Vergleich mit SOA auf den verschiedenen Ebenen unternommen.

Eigenständigkeitserklärung

Hiermit versichere ich, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Hilfsmittel verwendet habe. Sinngemäße Übernahmen aus anderen Werken sind als solche kenntlich gemacht und mit genauer Quellenangabe (auch aus elektronischen Medien) versehen.

Ulm, den 23. Januar 2017 Ferdinand Birk

Inhaltsverzeichnis

Separation of Concerns, even if not perfectly possible, is yet the only available technique for effective ordering of one's thoughts that I know of.

-Edsger W. Dijkstra

1 Einleitung

Nahezu jedes Unternehmen setzt heute Informationstechnologie (IT) in irgendeiner Form ein. Im Zuge der Digitalisierung ist die IT aber immer weiter in den Mittelpunkt der Geschäftsabläufe gerückt und muss heute sehr hohen Anforderungen an Effektivität und Flexibilität genügen. Fachabteilungen fordern schnelle Änderungen von einer IT-Organisation, die sich an der fachlichen Organisation orientiert (Capgemini, 2014). Speziell die Softwaresysteme sind so schnelleren Änderungen ausgesetzt und müssen unterschiedlichste Anforderungen berücksichtigen.

Der Lebenszyklus einer Software beginnt typischerweise mit der Aufnahme von Anforderungen und der Festlegung ihrer Architektur. Nach der Entwicklung geht die Software in Produktion und wird fortan in der Evolutionsstufe von Entwicklerteams gewartet, weiterentwickelt und optimiert. Die ursprüngliche gewählte Architektur ist meist stabil und relativ schwer änderbar. Funktionsänderungen sind nur solange möglich, wie das notwendige Know-How der Mitarbeiter oder eine relativ überschaubare Struktur vorhanden sind. Je nach Softwaresystem kann eine Software so mehrere Jahre in dieser Evolutionsstufe verbleiben. Zwangsläufig werden irgendwann die Entwickler wechseln und damit Schritt für Schritt Know-How verloren gehen. Zusätzlich werden sich durch Änderungen mit der Zeit sehr hohe interne Abhängigkeiten einschleichen und die Software so in die Servicestufe übergehen, wo sie nur noch minimal verändert werden kann und als Legacy-System ihren Dienst vollbringt, bis sie nicht mehr relevant ist. In dieser Stufe sind nur noch kleine Änderungen möglich, welche zusätzlich mit hohen Kosten und Risiken für neue Fehler verbunden sind.

Sarkar et al. zeigten in einer Fallstudie, dass ein über mehrere Jahre gewachsener Software-Monolith nur mit sehr hohem Aufwand umstrukturiert und wieder wartbar gemacht werden kann (Sarkar et al., 2009). Ein Unternehmen, für das solche Software einen geschäftskritischen Teil darstellt, ist unflexibel und kann dadurch schlecht auf veränderte Marktbedingungen reagieren und hat Probleme, die Software bei eventuellem Wachstum stabil zu halten. Der ganze Monolith muss irgendwann ausgemustert werden und wird eventuell neu entwickelt (Bennett & Rajlich, 2000).

Die geforderte Flexibilität ist mit klassischer Softwareentwicklung und starren Architekturen nicht mehr zu erreichen. In den vergangenen Jahren wurde daher nach Möglichkeiten gesucht, Software immer weiter zu entkoppeln. Mit Service orientierten Architekturen(SOA) wurden den komplexen und monolithischen Legacy-Systemen Adapter aufgesetzt und dadurch erreicht, dass deren Funktionen über viele kleine Web-Services genutzt werden können. Über Service-Orchestrierung wurde es möglich, einzelne Aktionen von Geschäftsprozessen auf Web-Services abzubilden. Die IT sollte bei Prozessänderungen somit durch eine neue Orchestrierung von Web-Services ganz einfach angepasst werden können, wobei eventuell nur kleine Web-Services neu implementiert werden mussten.
Durch falsche Interpretationen von SOA, bei denen hauptsächlich auf Middleware in Form eines Enterprise Service Bus (ESB) fokussiert wurde, gelten viele Service-orientierte-Architekturen als fehlgeleitet oder gescheitert (Woolf, 2007). Die Komplexität des vorhandenen Systems wurde dabei nicht reduziert, sondern in ein teures Softwareprodukt verpackt und gleichzeitig wurde vergessen, organisatorische Veränderungen vorzunehmen und fachliche Prozessstrukturen zu berücksichtigen.

> „I don't think ESB is Enterprise Service Bus, I think it is erroneous Spaghetti Box. Because you take all of that complexity that was in your previous Enterprise Architecture and you drop it into someone else's proprietary framework."
>
> — Dr. Jim Webber, QCon Conference London 2008

Mit dem Aufkommen der Cloud und mobilen Anwendungen sind die Anforderungen an Flexibilität, Geschwindigkeit und Zuverlässigkeit gestiegen, gleichzeitig ergeben sich aber auch neue Möglichkeiten für die Unternehmens-IT.
Unternehmen müssen immer schneller auf veränderte Nutzungsgewohnheiten reagieren, was die Entwickler unter enormen Zeitdruck setzt. Gleichzeitig müssen Fehler im Betrieb vermieden werden, um Nutzer nicht an Konkurrenten zu verlieren. Speziell Online-Unternehmen, beispielsweise mit „*Software as a Service* "-Angeboten, müssen dynamische Lastaufkommen bewältigen und dabei trotzdem geringe Antwortzeiten gegenüber den Nutzern erreichen. Herkömmliche SOA's sind diesen Anforderungen nicht gewachsen und erfordern ein Umdenken von zentralisierten Architekturen hin zu kleinen dezentralisierten Anwendungen, die einzeln skalierbar sind und modular hinzugefügt werden können. Moderne Dienste in der Cloud bieten dafür passende Infrastrukturen.

Das Konzept der Microservices versucht diesen neuen Anforderungen gerecht zu werden, indem es agile Entwicklungsmethoden und Konzepte wie Continuous Delivery oder DevOps unterstützt, fachliche Organisationen fördert, auf die Eigenheiten der Cloud abgestimmt ist und gleichzeitig sehr flexibel in der Skalierung von Entwicklung und Betrieb ist.

1.1 Problemstellung

Microservices ist ein neuer Begriff, der vielmehr ein Architekturkonzept beschreibt, das für viele Softwareentwickler ungewohnt ist und gegenüber serviceorientierten Architekturen (SOA) schwer einzuordnen ist. Da dieser Begriff noch sehr jung, ist gibt es in der Fachwelt und in der begrenzten Literatur wenig Einigkeit über das Themengebiet und eine Abgrenzung zu anderen Konzepten wie SOA.

1.2 Zielsetzung

Diese Arbeit soll dazu beitragen mehr Verständnis über das Themengebiet Microservices und Microservice-Architekturen, sowie seine angrenzenden Techniken und Konzepte zu erlangen. Des Weiteren soll eine Einordnung der sich ähnelnden Begrifflichkeiten *„serviceorientierte Architektur“* und *„Microservices“* herausgearbeitet werden.

1.3 Struktur der Arbeit

Zu Beginn der Arbeit wird der Begriff *„Microservices“* definiert und eingeführt. Im Folgenden werden zugehörige Eigenschaften behandelt und Konzepte eingeführt, die für eine erfolgreiche Microservices-Umsetzung Relevanz haben.
Anschließend wird anhand von Charakteristiken auf Schwierigkeiten und deren Lösungsansätze eingegangen.
Das vierte Kapitel erörtert unterschiedliche Motivationen für vorhandene praktische Umsetzungen.
Im fünften Kapitel wird eine Einordnung von SOA und Microservices unternommen.

2 Grundlagen

Das Thema Microservices erweckt seit 2014 immer mehr Interesse (vgl. Abbildung 2.1), nachdem wenige Jahre zuvor mehrere populäre Unternehmen aus der „Internet-Economoy“, allen voran Netflix und Amazon, Konzepte ihrer internen Architektur auf Konferenzen veröffentlichten und Experten aus Thoughtworks-Kreisen begannen, den Begriff Microservices zu prägen. Ein weiterer Grund liegt darin, dass moderne Cloudinfrastrukturen, die das Konzept von Microservices wesentlich vereinfachen, inzwischen als etabliert und zuverlässig gelten.

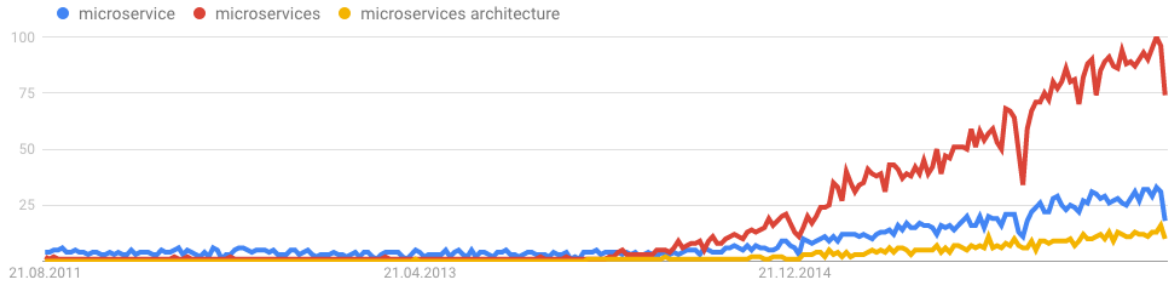

Abbildung 2.1: Google Trendverlauf von Keywords im Zusammenhang mit Microservices (*Google Trends*, 2016).

In diesem Kapitel werden die grundlegenden Begriffe rund um Microservices genauer definiert und erklärt.

2.1 Beispiel Tutinder

Als durchgehendes Beispiel soll die Tutinder-Anwendung aus dem Projektmodul *Mobile Application Development* dienen.
Die Anwendung ermöglicht es Studenten, sich in Vorlesungskurse einzuschreiben und auf praktische Art und Weise Partner für Tutorien und Abgaben zu finden. Eingeschriebene Benutzer können andere eingeschriebene Benutzer anhand Profilbild, Persönlichkeitstags, Studiengang und Beschreibung positiv oder negativ bewerten. Die Benutzer bemerken von den gegenseitigen Bewertungen nichts. Nur für den Fall, dass sich zwei Benutzer gegenseitig positiv bewerten wird ein *Match* erzeugt, was sie in die Lage versetzt, gemeinsam eine Abgabegruppe zu gründen. Innerhalb der Gruppe können sie sich über eine Chatfunktion unterhalten. Zusätzlich bietet die Anwendung die Möglichkeit, über das Abscannen eines QR-Codes einen anderen Benutzer als

Freund zu akzeptieren. Freunde haben die Möglichkeit, direkt eine Gruppe zu gründen, sofern sie beide in dem gleichen Kurs eingeschrieben sind.

2.2 Microservices Begriffsklärung

Der Begriff *Microservices* wird mehrdeutig verwendet und beschreibt im Grunde eine Architektur für Softwaresysteme und die einzelnen Komponenten dieser Architektur zugleich. Mit Microservices eng verbunden, sind auch veränderte Organisationen für die Softwareentwicklung, Methodiken, neue Herausforderungen und Probleme, die dadurch gelöst werden sollen. Eberhard Wolf sieht unter Microservices auch ein Modularisierungskonzept, das dazu dient, große Softwaresysteme aufzuteilen und die Organisation und Entwicklungsprozesse beeinflusst (Wolff, 2016). Daraus ergeben sich drei Ableitungen für die Definition von Microservices:

1. **Stil/Konzept:** Ein Konzept für Modularisierung von Software, dass Wert auf eine bestimmte Organisation und Vorgehensweise legt.
2. **Komponente:** Der einzelne Microservice, der eigenständig existiert und seine fachlich orientierte Aufgabe erledigt.
3. **Architektur:** Microservice-Architektur entsteht als großes Bild über alle einzelnen Microservices hinweg.

2.3 Ein Microservice

Sam Newman beschreibt einen Microservice als *„eigenständig ausführbare Softwarekomponente, die innerhalb eines Anwendungssystems mit anderen Softwarekomponenten kollaboriert“*. Microservices sollen eine Funktion innerhalb einer fachlich abgegrenzten Geschäftsdomäne übernehmen und damit eine hohe Kohäsion im Sinne des *Single Responsibility Principle*[1] von Robert C. Martin erreichen (Newman, 2015).

James Lewis bezeichnet mit einem Microservice eine kleine Anwendung, die unabhängig deployt, skaliert und getestet werden kann. Diese Anwendung und ihr Code soll einfach zu verstehen sein und genau eine Aufgabe übernehmen. Wie Sam Newman erfordert auch er eine Anlehnung an das *Single Responsibility Principle* (Thones, 2015).

Bob Familiar sieht in einem Microservice eine in sich abgeschlossene Einheit, die lose mit anderen Einheiten gekoppelt ist, aber unabhängig entworfen, entwickelt, getestet und betrieben wird. Die Einheit erfüllt eine genau spezifizierte Aufgabe. Der Service muss fehlertolerant sein und im Fehlerfall schnell in einen funktionierenden Zustand zurückkehren, sowie elastisch skalierbar sein (Familiar, 2015).

[1] Es besagt, dass eine Komponente nur genau eine Verantwortlichkeit abdecken soll und es somit nur einen Grund gibt, dass sich die Komponente ändert. Nämlich genau dann, wenn sich die eine Verantwortlichkeit ändert.

Dragoni et al. beschreiben einen Microservice als *„minimal independent process interacting via messages“*. Minimal soll in diesem Zusammenhang bedeuten, dass der Service nur Funktionalitäten implementiert, die seinem zugewiesenen Zweck dienen. Als Beispiel für „minimal“ wird ein Taschenrechner angeführt, der genau dann minimal ist, wenn er nur die benötigten Rechenoperationen implementiert. Beispielsweise gehört das Plotten von Funktionen nicht zu seiner zugewiesenen Aufgabe und sollte nicht im Taschenrechner-Microservice implementiert sein (Dragoni et al., 2016).

Die Charakteristik, die sich für einen Microservice ableiten lässt, ist in Abbildung 2.2 dargestellt.

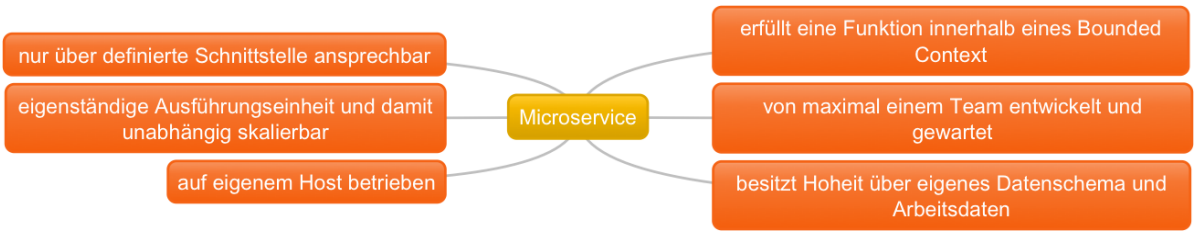

Abbildung 2.2: Charakteristik eines Microservice

Ein Microservice kann als eigenständiges Programm gesehen werden, das seinen eigenen Lebenszyklus innerhalb des Systems hat. Jeder Service soll eigenständig betrieben werden können, unabhängig davon, welche anderen Services in dem System noch vorhanden sind und auf welcher anderen Technologie diese basieren. So können einzelne Services einfach horizontal skalieren, indem mehrere Instanzen davon gestartet werden. Dafür wird meist eine Zustandslosigkeit gefordert.
Jeder Microservice erfüllt eine konkrete Aufgabe innerhalb eines Anwendungssystems, wobei die Aufgaben so zueinander abgegrenzt sind, dass sie bei Änderungen der Aufgabenspezifikation möglichst wenig gegenseitige Auswirkungen aufeinander haben.
Oft wird in diesem Zusammenhang dem Microservice die Hoheit über das Schema der eigenen Daten gegeben. Somit ist eine unabhängige Evolution des Service gegeben, da jeder Service seine eigene Datenbank mit beliebiger Technologie nutzen kann. Ein Microservice besitzt eine tolerante und zugleich robuste Schnittstelle, durch die seine Logik und die Daten gekapselt werden. So kann jeder Microservice unabhängig von anderen getestet werden. Ein erfolgreicher Test zeichnet sich durch eine fehlerfreie interne Funktionalität und Implementierung der Schnittstellenspezifikation aus.
Das Starten eines Service sollte nur wenige Augenblicke dauern, um eine schnelle Entwicklung und elastische Skalierung zu unterstützen.

2.3.1 Größe eines Microservice

Das Präfix *Micro* suggeriert, dass es sich um kleine Services handeln soll. Eine genaue Definition der Größe eines solchen Service gibt es nicht. Je nach Größe und Anzahl der Microservices im System, sind unterschiedliche Aspekte zu berücksichtigen.
Es gilt der Grundsatz, dass ein Service maximal so groß sein sollte, dass jedes Teammitglied seine Funktionsweise und den Code vollumfänglich verstehen kann (Lews, 2012). Zusätzlich sollte das verantwortliche Team in der Lage sein, diesen innerhalb von wenigen Wochen

komplett neu zu entwickeln, um zum Beispiel Vorteile von anderen Sprachen und von neuen Technologien ausnutzen zu können.

Wählt man Services zu groß und packt zu viel Funktion in einen einzelnen Microservice, kann er nicht mehr von einem Team gewartet werden. Das sollte unbedingt vermieden werden, da Verantwortlichkeiten zwischen Teams dann nicht mehr klar definiert sind. Gleichzeitig wird es für die Teammitglieder schwieriger, seine volle Funktionalität zu verstehen. Ein weiterer Nachteil ist die längere Einarbeitung von neuen Team-Mitgliedern, für die es mit zunehmender Größe der Services schwieriger wird, schnell produktiv arbeiten zu können.

Sind die Services hingegen zu klein und werden in Folge dessen zu viele Services betrieben, ergibt sich ein erheblicher Mehraufwand, um die Konsistenz zwischen den Daten der einzelnen Services zu erhalten. Zusätzlich wird der Kommunikationsaufwand über das Netzwerk höher, je verteilter die Funktionalität des Gesamtsystems ist. Letztlich limitiert auch die vorhandene Infrastruktur, sodass ab einer gewissen Anzahl von Microservices ein hoher Grad an Automatisierung gegeben sein muss (Wolff, 2016).

2.4 Software Architektur

Die Software Architektur legt die innere und äußere Struktur eines Software-Systems fest. Dazu gehören die Aufteilung der Software in Module, die Art der Benutzerschnittstelle, die Verteilung der Arbeitsdaten und die technologische Plattform, auf der dieses System betrieben werden kann oder soll. Software Architektur beschreibt hierbei die genauen Beziehungen zwischen den Teilen und soll es ermöglichen, die Fähigkeit der Software abzuleiten und hinsichtlich der Änderbarkeit zu bewerten (Hohmann, 2003).

Sie lässt sich in zwei Kategorien unterteilen:

- **Makro-Architektur:** Spezifiziert das globale System inklusive der Abgrenzung einzelner Komponenten und macht notwendige Vorgaben, wie diese miteinander interagieren.
- **Mikro-Architektur:** Spezifiziert die interne Charakteristik der Komponenten.

2.4.1 Monolithische Architektur

Mit dem Begriff *„monolith“* wurde schon 1982 Softwarecode bezeichnet, der aufgrund interner Abhängigkeiten schwer wartbar war. Beim Ausbessern von Fehlern an einer Stelle, haben sich neue Fehler an anderer Stelle eingeschlichen (Warner, 1982).

Dragoni et al. geben eine allgemeinere Definition einer monolithischen Software, kurz Monolith genannt:

> A monolithic software application is a software application composed of modules that are not independent from the application to which they belong.

Wir schließen daraus, dass ein Monolith intern sehr wohl modularisiert sein kann. Einige Hochsprachen bringen sogar Konzepte mit, um Software zu modularisieren, entscheidend

ist aber, dass die einzelnen Module nicht unabhängig sind. Unabhängigkeit kann sehr weit gefasst sein. Das hat den Vorteil, auch Architekturen, die auf eigentlich unabhängigen Services basieren als monolithisch anzusehen, wenn diese aufgrund von schlechtem Design nicht mehr unabhängig sind und die Charakteristik eines Deployment-Monolithen aufweisen.

Aus der Microservice-Perspektive ist ein *Deployment-Monolith* eine Software, welche als einzelne große Anwendung existiert, die als Ganzes kompiliert und deployt wird. Es handelt sich um eine einzelne logische Ausführungseinheit. Änderungen an einem Teil der Ausführungseinheit haben zur Folge, dass das gesamte System neu kompiliert und deployt werden muss (Wolff, 2016).

Zusammengefasst werden folgende Probleme einem Monolithen zugeordnet (Dragoni et al., 2016):

- **Schwer Wartbar:** Trotz Modularisierung schleichen sich mit zunehmender Größe Abhängigkeiten ein, weil sie oft eine sehr große und komplexe Codebasis besitzen. Änderungen führen zu ungewollten Auswirkungen und Fehler sind oft schwer zu finden.
- **Dependency Hell:** Bibliotheken, die von verschiedenen Modulen genutzt werden, führen oft zu Inkonsistenzen, wenn neuere Versionen benötigt werden.
- **Monolithisches Deployment:** Änderungen in einzelnen Modulen haben zur Konsequenz, dass die ganze Anwendung neu deployt werden muss.
- **Technologie-Kompromisse:** Alle Module müssen auf der gewählten Technologie basieren, obwohl diese für einzelne Module sub-optimal sein kann oder einige Entwickler auf eine andere Plattform spezialisiert sind.
- **Schlechte Skalierbarkeit:** Horizontale Skalierung kann nur über Duplizieren der gesamten Anwendung erfolgen, obwohl vielleicht nur einzelne Module für die Last verantwortlich sind.
- **Einheitliche Deployment Konfiguration:** Alle Module werden auf eine einheitlich konfigurierte Infrastruktur deployt. Unterschiedliche Anforderungen an zum Beispiel CPU und RAM müssen als Kompromiss gelöst werden.

Tutinder könnte beispielsweise eine in Java entwickelte Web-Anwendung sein, die als .war-Datei in einen Anwendungsserver wie zum Beispiel Glassfish oder Tomcat geladen wird und dort als ganzheitliche Anwendung (vgl. Abbildung 2.3) ausgeführt wird. Alle benötigten Module wurden gemeinsam kompiliert und in ein .war gepackt. Bei Änderungen in einem Modul muss die gesamte Anwendung inklusive aller anderen Module neu deployt werden.

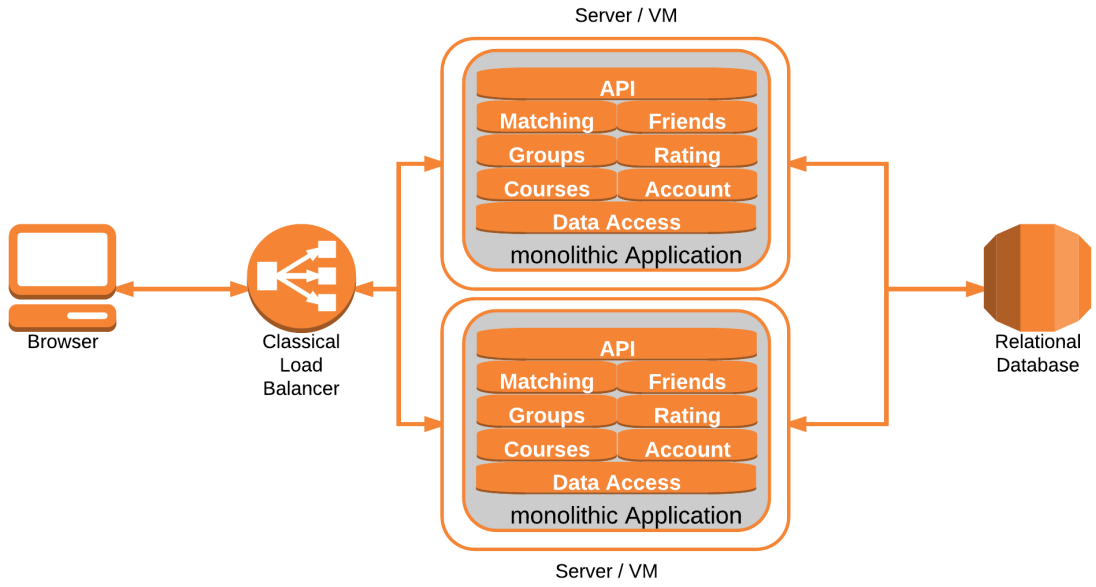

Abbildung 2.3: Typische monolithische Architektur: Die Anwendung wird komplett mit allen Modulen innerhalb eines Anwendungsservers ausgeführt und greift meistens auf eine große gemeinsame relationale Datenbank zu. Horizontale Skalierung erfolgt, indem die gesamte Anwendung dupliziert wird.

Ein weiteres Beispiel aus der Praxis ist der Amazon-Webshop, welcher noch vor Jahren eine monolithische Anwendung war. Die kompilierte Binärdatei war zwischen 300 und 500 Megabyte groß und beinhaltete alle benötigten Komponenten. Es dauerte bis zu 10 Stunden, um eine neue Version zu kompilieren (Novotny, 2015). Software in dieser Form erzeugt für ein Unternehmen sehr viel organisatorischer Mehraufwand, beispielsweise in Form von koordinierten Deployments und ist für ein schnell wachsendes Unternehmen nicht flexibel genug. Unter anderem, weil agile Methoden wie *Continuous-Delivery* (vgl. Abschnitt 2.7.4) nicht effektiv umsetzbar sind.

2.4.2 Microservice-Architektur

Dragoni et al. definieren eine Microservice-Architektur sehr einfach und präzise:

> A microservice architecture is a distributed application where all its modules are microservices.

Die Art der Modularisierung einer Microservice-Architektur (MSA) findet auf der Makro-Ebene demnach ausschließlich über Microservices statt. Zu bemerken ist, dass die Microservice-Architektur auf einen Anwendungskontext begrenzt ist und daher als Software-Architektur gilt. SOA kann im Gegensatz eher als Enterprise-IT Architektur gesehen werden, die mehrere Systeme vereint (vgl. Abschnitt 5.1).

Um eine Microservice-Architektur zu erhalten, muss auf Makro-Ebene einmalig gemeinsam von allen Verantwortlichen die Entscheidung für das Umsetzen des entsprechenden Stils (vgl. Abschnitt 2.5) beschlossen werden. Die Mikro-Architektur ist durch die Definition des Microservices festgelegt.

Abbildung 2.4 stellt eine einfache MSA am Beispiel der Tutinder-Anwendung dar.

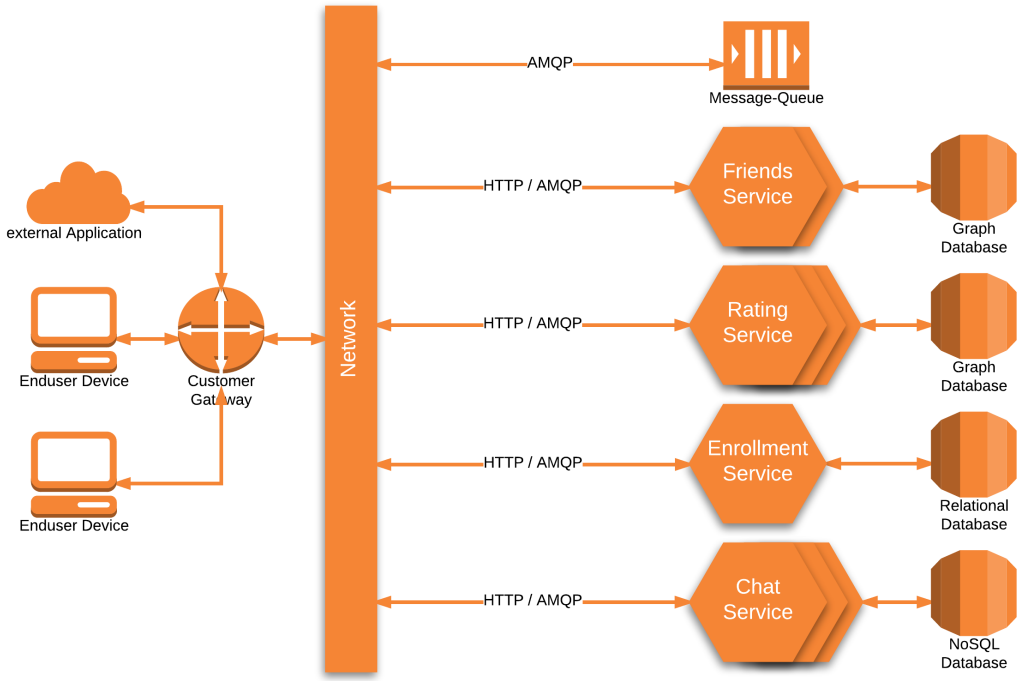

Abbildung 2.4: Microservice-Architektur der Tutinder-Anwendung (ohne Vollständigkeit): Die Anwendung ist in unterschiedliche Microservices aufgeteilt, die jeweils unterschiedlich stark skaliert sind. Jeder Microservice besitzt seine eigene Datenbank. Die Services kommunizieren untereinander mittels HTTP-Technologie oder Messages. In Anlehnung an (Stine, 2014).

Wir verwenden die Definition von Dragoni et al. und fordern zusätzlich, dass die Kommunikation über einen leichtgewichtigen Mechanismus stattfindet, wobei leichtgewichtig im Sinne der Charakteristik *smart endpoints and dump pipes* (vgl. Abschnitt 3.4) zu sehen ist.

Der *microservice architectural style* wird oft als Synonym für die Microservice-Architektur verwendet. Dabei würden Aspekte einfließen, welche über den Kontext einer Software-Architektur hinausgehen. Diese Arbeit unterscheidet deshalb zwischen Microservice-Architektur und Microservice-Stil.

2.5 Microservice-Stil

Wenn Experten von Microservices reden, meinen sie meist weniger den konkreten Service oder die Microservice-Architektur. *Microservices* oder *microservice architectural style* sind vielmehr stellvertretend für ein weitgefasstes Konzept, welches die Organisation, die angewendeten Techniken, eine Entwicklungskultur und vor allem die Art der Modularisierung von Systemen über Komponenten in Form von Microservices beschreibt. Aus diesem Stil haben sich auch die Definitionen für einen Microservice und der MSA abgeleitet.

Viele Merkmale und Ideen des Microservice-Stils sind nicht neu und stammen aus der UNIX Welt, wo viele eigenständige Programme im Betriebssystem sehr spezifische Aufgaben erledigen und beliebig kombiniert werden können. Richard Raymond beschreibt den UNIX-Ansatz in „The Art of Unix Programming“ folgendermaßen (Raymond, 2008):

> „The only way to write complex software that won't fall on its face is to hold its global complexity down — to build it out of simple parts connected by well-defined interfaces, so that most problems are local and you can have some hope of upgrading a part without breaking the whole.“

Damit beschreibt Raymond zugleich auch den Kernpunkt des Microservice-Stils. Die Modularisierung von Software soll über kleine Teile mit starken Schnittstellen stattfinden. Die Definition für den Microservice-Stil von Fowler und Lewis spezifiziert den Stil näher. Sie ist zugleich auch Grundlage für die meisten Definitionen des Microservice an sich (Fowler & Lewis, 2014):

> „In short, the microservice architectural style is an approach to developing a single application as a suite of small services, each running in its own process and communicating with lightweight mechanisms, often an HTTP resource API. These services are built around business capabilities and independently deployable by fully automated deployment machinery. There is a bare minimum of centralized management of these services, which may be written in different programming languages and use different data storage technologies.“

Das Konzept schreibt kein Programmierparadigma vor, stellt aber Anforderungen an die Organisation und Infrastruktur der Entwicklung.
Die Forderung nach automatischen Deployments zielt auf ein konsequentes Anwenden von Continuous Delivery und DevOps ab. Nur durch viel Automatisierung lässt sich der zusätzliche Aufwand von Microservices gegenüber Monolithen rechtfertigen und bewältigen. Unter anderem prägen auch neue Container-Technologien und die immer einfachere Nutzung von Cloud-Angeboten den Microservice-Stil wesentlich.
Ebenfalls prägend für die organisatorischen Aspekte des Microservice Stils sind Erkenntnisse aus *Conways Law*. Erst durch strukturelle Veränderungen der Entwicklungsabteilungen wird eine Entwicklung unabhängiger Microservices ermöglicht. Werden eine korrekte fachliche Trennung von Services und Teams bei der Umsetzung von Microservices vergessen, wird man eine eng gekoppelte Microservice-Architektur erhalten, die einem Deployment-Monolithen (vgl. Abschnitt 2.4.1) gleicht, da die Unabhängigkeit der Services nicht gegeben ist.
Somit ist die Vorgehensweise bei der Modularisierung und die Abgrenzung der fachlichen Geschäftsfelder vermutlich der wichtigste und zugleich schwierigste Aspekt für einen sauberen Microservices-Stil. Wie diese fachliche Einteilung korrekt vorzunehmen ist, wird von der Definition nicht abgedeckt und ist von Fall zu Fall unterschiedlich. Fachliche Geschäftsfelder haben

keine harten Grenzen und können mal größer oder kleiner ausfallen. Grundsätzlich wird empfohlen mit einer eher groben Unterteilung zu starten. Sind die unabhängigen Eigenschaften des Stils irgendwann nicht mehr gegeben, muss eine weitere Unterteilung stattfinden. Als Anhaltspunkt für die Modularisierung kann *Domain Driven Design* (vgl. Abschnitt 2.7.3) dienen. Die daraus resultierenden *Bounded Contexts* gelten als Abgrenzung für entsprechende Teams und deren Services mit ihren eigenen Datenschemata.

Kapitel 3 wird die genauen Charakteristiken des Konzeptes näher erläutern und Beispiele von existierenden Umsetzungen aufzeigen. Kapitel 4 wird sich mit der Analyse von praktischen Umsetzungen beschäftigen und Motivationen für den Stil herausarbeiten.

2.6 Kommunikation

Anders als in einer monolithischen Anwendung können zwischen Microservices einfache Funktions- oder Methodenaufrufe nicht innerhalb des Prozesses getätigt werden. Aufrufe müssen in der Regel über das Netzwerk erfolgen.
Generell gibt es dazu zwei Möglichkeiten (Newman, 2015):

Request / Response Hierbei wird ein Microservice einen Aufruf an einen anderen Microservice senden und eine Antwort erwarten. Das hat den Vorteil, dass der Aufrufer direkt Feedback erhält, ob sein Aufruf erfolgreich war oder ob Fehler aufgetreten sind. Dieses Konzept beruht auf Anweisungen und erlaubt es dem aufrufenden Service, zu einem gewissen Grad den Prozessablauf zu steuern und zu kontrollieren. So könnte beispielsweise bei einer Benutzerregistrierung in dem Tutinder-System der Registrierung Service den Account Service beauftragen, interne Datenstrukturen für diesen Benutzer anzulegen und gleichzeitig einen Email Service anweisen eine Willkommensnachricht zu versenden. Der Registrierung Service erhält direkte Rückmeldung ob die erwünschten Prozessschritte durchgeführt wurden.

Event basiert (Publish / Subscribe) Bei dieser Möglichkeit wird ein Microservice bei bestimmten Ereignissen eine Nachricht eines bestimmten Typs versenden. Weitere Microservices können verschiedene Nachrichtentypen abonnieren und entsprechende Aktionen ausführen. Beispielsweise würde ein Enrollment Service bei einer neuen Einschreibung eines Nutzers in einen Kurs eine Nachricht *„Neue Einschreibung von Nutzer* x *in Kurs y “* versenden, worauf andere Services dann intern reagieren können. Zum Beispiel könnte der Rating-Service die Datenstruktur für diesen Nutzer anlegen und der Friend-Service den Freunden dieses Nutzers anzeigen, dass der Nutzer nun auch in diesem Kurs ist. Was die einzelnen Services mit den Nachrichten anfangen, bekommt der emittierende Microservice nicht mit und weiß nicht einmal, welche anderen Services auf diese Nachricht reagieren.

2.6.1 Representational State Transfer (REST)

REST ist eine sehr verbreitete Art, um Schnittstellen für Microservices zu entwerfen. Dieses Konzept nutzt allgemein anerkannte Techniken des World-Wide-Web wie HTTP oder Hypermedia und baut auf die in HTTP enthaltenen Request-Methoden und Response-Codes. Die drei Hauptaspekte dieses Konzeptes sind Ressourcen, Repräsentationen und Zustände.

Tabelle 2.1: Abbildung von Ressourcen auf Entitäten in einer REST Umgebung

Zeitpunkt	Ressource	Entität
1	software/latest	Software mit der Version 1.6.2
1	software/1.6.2	Software mit der Version 1.6.2
2	software/1.6.2	Software mit der Version 1.6.2
2	software/1.7.2	Software mit der Version 1.7.2
2	software/latest	Software mit der Version 1.7.2
2	software/1.8.2	noch nicht vorhanden

2.6.1.1 Ressourcen

Jede Information, der man einen Namen geben kann, ist potentiell eine Ressource. Physische Objekte wie zum Beispiel *„eine Person“* oder aber abstrakte Informationen wie *„das heutige oder gestrige Wetter“* können Ressourcen sein. Selbst eine Transaktion zwischen zwei Konten sollte nicht als Aktion beziehungsweise Verb repräsentiert werden, sondern vielmehr als Eigenständige Ressource *„Transaktion“* die einen Vorgang zwischen zwei Konten beschreibt. Nach Roy Fielding's Definition ist eine Ressource nicht direkt eine Entität, sondern vielmehr eine Abbildung auf eine Menge von Entitäten. Wobei zu einem bestimmten Zeitpunkt genau eine konkrete Entität aus dieser Menge referenziert wird. Es ist sogar möglich, Ressourcen zu definieren, die auf eine leere Menge von Entitäten zeigt (Fielding, Roy Thomas, 2000). In Tabelle 2.1 sieht man beispielsweise die Repräsentation einer Software mit verschiedenen Versionen in einem Software-Versionierungssystem. Die Ressource */latest* zeigt an zwei verschiedenen Zeitpunkten auf zwei verschiedene Entitäten aus der Menge aller Entitäten und die Ressource */1.8.2* ist zum Zeitpunkt *2* noch gar nicht vorhanden.

2.6.1.2 Repräsentationen

Jede Ressource kann verschiedene Repräsentationen haben. Beispielsweise kann sie mit klassischem Text ohne Auszeichnungen dargestellt sein, sie kann in einem sehr mächtigen Format wie XML oder in einem leichtgewichtigen Format wie JSON repräsentiert sein. Eine andere Ressource verweist möglicherweise auf eine Multimedia-Entität und wird daher in einem Mutlimedia konformen Dateiformat repräsentiert. Dank der Nutzung von HTTP kann über Content-Negotiation Mechanismen eine Vereinbarung der passenden Repräsentation zwischen Client und Server stattfinden und so eine noch größere Entkopplung dieser erreicht werden (Fielding & Reschke, 2014).

2.6.1.3 Zustände

Generell soll die Interaktion zweier Teilnehmer über eine REST-Schnittstelle zustandslos sein. Der Server soll sich keinen Anwendungszustand des Clients merken müssen, so dass jede Anfrage eines Clients als erster Kontakt zwischen Client und Server angesehen werden kann. Der Zustand der Ressource hingegen wird auf dem Server verwaltet (Feng, Shen & Fan, 2009).

Es gibt keinen Standard in dem REST definiert ist, vielmehr gilt es, Prinzipien aus den HTTP-Standards und Best-Practices einzuhalten. So sollte eine REST-Schnittstelle für die entspre-

chenden HTTP-Operationen jeweils ein definiertes Verhalten aufweisen. Die Tabelle 2.2 listet das gewünschte Verhalten der 5 wichtigsten einzelnen HTTP-Operationen beim Aufruf einer Ressource auf.

Tabelle 2.2: Verwendung von HTTP-Verben in REST

Verb	Verhalten	Sonstiges
GET	Gibt die aktuell referenzierte Entität hinter der Ressource zurück	Kann mit *If-Headern* zu *conditional-GET* erweitert werden
PUT	Ersetzt die referenzierte Entität hinter der Ressource durch den mitgelieferten Inhalt	Existiert die spezifizierte Ressource noch nicht, so wird sie erstellt.
PATCH	Ersetzt nur Teile einer Ressource durch den mitgelieferten Inhalt	Wird verwendet, um die Datenmenge bei großen häufig geänderten Ressourcen gering zu halten
POST	Erstellt eine neue Ressource; der mitgelieferte Inhalt repräsentiert die zu referenzierende Entität	Pfad zur neuen Ressource in der Antwort enthalten
DELETE	Löscht die Ressource und die dahinter referenzierte Entität	

2.6.2 Messaging

Messaging basiert auf dem Verschicken von Nachrichten und wird in der Regel asynchron eingesetzt. Theoretisch kann mit Messaging auch das Request-Response Schema umgesetzt werden, indem eine Nachricht verschickt wird und auf die Antwortnachricht gewartet wird. Die Charakteristik von Messaging-Systemen passt aber wesentlich besser zu ereignisbasierter Kommunikation. In einem hochgradig verteilten System wie einer Microservice Architektur, welche mit Latenzen und der Fehleranfälligkeit von Netzwerken zu kämpfen hat, können Messaging-Systeme ihre Stärke ausspielen (Wolff, 2016).

- Nachrichten überleben den Ausfall des Netzwerkes, da sie in den Systemen zwischengespeichert werden. Sie werden zugestellt, sobald die Empfänger wieder erreichbar sind.
- Bei Bearbeitungsfehlern an Empfängern wird die Bearbeitung erneut versucht (Anzahl der Versuche konfigurierbar) bevor eine Fehlermeldung zum Emittenten gesendet wird.
- Durch den asynchronen Ansatz können Messaging-Systeme besser mit Netzwerklatenzen umgehen.
- Die Systeme ermöglichen es, Sender und Empfänger komplett zu entkoppeln. Es ist keine Kenntnis voneinander notwendig. So ist es unter anderem möglich, dass mit einer Nachricht mehrere Empfänger erreicht werden.
- Einige Messaging-Systeme erlauben Transaktionen in verteilten Systemen und unterstützen dadurch Entwickler bei der Erhaltung der Konsistenz.

Obwohl Messaging-Systeme perfekt auf die Anforderungen einer Microservice Architektur passen, muss darauf geachtet werden, welche Funktionen dem System übertragen werden. Grundsätzlich ist ein Messaging-System in Form eines leichtgewichtigen Message-Brokers zu empfehlen, der sich auf die Zustellung von Nachrichten und gegebenenfalls um Transaktionen kümmert. Werden zu viele Funktionen auf das zentrale System übertragen, würde der dezentralisierte Ansatz von Microservices durchbrochen werden und das Prinzip der „Dumb Pipes“ (vgl. Abschnitt 3.4) verletzt werden.
Darüber hinaus erhöhen Messageing-Systeme die Komplexität der Architektur und benötigen zusätzliche Infrastruktur und Know-How. Entwickler müssen mit den Eigenheiten asynchroner Kommunikation vertraut gemacht werden und ohne ein gutes Monitoring-System wird es schwierig, ein Fehlverhalten des Software-Systems zu beheben (Newman, 2015). Zuletzt sollte man sich darüber im Klaren sein, dass obwohl Message-Broker auf Robustheit ausgelegt sind, solche Systeme einen „Single Point of Failure“ darstellen. Ein Ausfall eines solchen Systems hat zur Folge, dass sämtliche Kommunikation zwischen den Services zusammenbricht (Wolff, 2016).

Ereignisbasierte-Messages ermöglichen bei richtiger Anwendung eine sehr starke Entkopplung zwischen den Microservices, da sie als Events nur Fakten transportieren und nicht mittels Kommandos Anweisungen an andere Services verteilen. Jeder Service kann auf andere Events reagieren, wie er es für sinnvoll erachtet. Ist ein Service fehlerhaft und versendet keine Events, so hat dies keinerlei Auswirkung auf andere Services.

2.6.3 Remote Procedure Calls (RPC)

RPC ist eine Technik, um Funktionen über Prozessgrenzen hinweg aufzurufen. Prinzipiell ist sie daher auch geeignet, um mehrere Microservices miteinander kommunizieren zu lassen. Einen interessanten Ansatz verfolgt das Apache Thrift Projekt, welches ursprünglich von Facebook vorgestellt wurde (Slee, Agarwal & Kwiatkowski, 2007).
Thrift ist eine Library, die darauf ausgelegt ist, eine effiziente und zuverlässige Kommunikation zwischen Backend-Services zu gewährleisten. Es nutzt eine sprachunabhängige Initialisierungsdatei, in der Datentypen und Schnittstellen spezifiziert werden. Mittels Code-Generatoren werden aus den Definitionen der für eine RPC-Kommunikation nötige Code in den unterstützten Sprachen erzeugt (Apache Software Foundation, 2016).
Die Library unterstützt einfache Datentypen und Structs, welche auch für das Exception-Handling genutzt werden. Zusätzlich kann man den Datenfeldern manuell Identifier zuweisen oder automatisch generieren lassen, welche später für die Versionierung der Schnittstelle genutzt werden.
Der große Vorteil von Thrift gegenüber den etablierten strukturbasierten Protokollen ist die Datenübertragung von Binärdaten. Damit entfällt ein aufwändiges Parsen von zum Beispiel XML und ermöglicht eine effizientere Kommunikation (Sumaray & Makki, 2012).

Natürlich ist es auch möglich, das im SOA-Umfeld verbreitete SOAP als Kommunikationsprotokoll einzusetzen. Wegen des komplexen, Webservice spezifischen, Protokollstacks, den SOAP mit WSDL und WS*- Technologien mitbringt, kann es als schwergewichtig angesehen werden und wird daher im Rahmen von Microservices sehr selten eingesetzt.
Außerdem ist SOAP mit sehr viel Parsen von verschiedensten XML Dateien verbunden. Es müssen Schema, WSDL und die Nachricht bei jedem Aufruf geparst werden. Das hat zwar

den Vorteil, dass alle Aufrufe streng validiert werden können, hat aber gegenüber anderen Protokollen einen erheblichen Performance Nachteil.
Ein weiterer Nachteil von SOAP zeigt sich, wenn man seine Microservices direkt der Öffentlichkeit zur Verfügung stellen möchte. Für öffentliche APIs spielt SOAP heute praktisch keine Rolle mehr, das zeigt sich auch dadurch, dass in den letzten Jahren immer mehr namhafte Internetdienste ihre Unterstützung für SOAP-APIs beendet haben (Oracle, 2016),(Amazon Web Services, 2016b),(Atlassian, 2016) und (Tholomé, 2009).

2.7 Grundbegriffe

Um Microservices und ihre Intention zu verstehen, sind einige Begriffe und Konzepte notwendig, welche im Folgenden beschrieben werden.

2.7.1 CAP-Theorem

Die von Eric Brewer aufgestellte These besagt, dass sich Konsistenz (C), Verfügbarkeit (A) und Partitionstoleranz (P) in einem verteilten System nicht gleichzeitig erfüllen lassen. Sie gilt inzwischen als belegt (Gilbert & Lynch, 2002).

Konsistenz meint in diesem Zusammenhang, dass alle Knoten des Systems mit den aktuellsten Daten arbeiten können und bei Anfragen keine auf alten Daten basierende Antwort zurückliefern.

Verfügbarkeit meint, dass jede Anfrage eines Nutzers an das System irgendwann sinnvoll beantwortet wird. Aus praktischer Sicht ist eine sinnvolle Verfügbarkeit oft schon nicht mehr gegeben, falls die Antwortzeiten mehrere Sekunden betragen.

Partitionstoleranz bezieht sich auf die Verteilung des Systems über mehrere Netzknoten. Das System soll auch funktionieren, wenn Netzwerkpartitionen auftreten. Das heißt, bei Kommunikationsfehlern innerhalb des System, so dass sich Knoten in mindestens zwei Gruppen einteilen lassen, die nicht fehlerfrei miteinander kommunizieren können. Beispielsweise eine unterbrochene Verbindung zwischen zwei Rechenzentren, die gemeinsam ein System betreiben.

Da hinlänglich bekannt ist, dass Kommunikationsnetzwerke nicht zuverlässig, sind wird ein System meist die Anforderung der Partitionstoleranz erfüllen müssen, um zuverlässig zu funktionieren.
Daraus lässt sich schließen, dass zwischen Konsistenz und Verfügbarkeit des Systems abgewägt werden muss. Dabei ist zu beachten, dass die Abwägung zwischen Konsistenz und Verfügbarkeit für einzelne Teile des Systems unterschiedlich gewählt werden kann. Es ergibt sich damit ein Kompromiss für das Gesamtsystem.
So gibt es einige Funktionen, die strenge Konsistenz verlangen und Anfragen mit Fehlern beantworten, anstatt falsche Informationen zurückzuliefern. Für andere Aufgaben kann es aber sinnvoll sein, die strenge Konsistenz aufzugeben und stattdessen weniger aktuelle oder nicht vollständige Daten zu liefern, um den Nutzer nicht zu verärgern.

2.7.1.1 CAP-Theorem und Microservices

Da wir es bei Microservice-Architekturen mit verteilten Softwaresystemen zu tun haben, ist die Betrachtung des CAP-Theorems in diesem Kontext unumgänglich. Das klassische relationale Datenbanksystem aus einer monolithischen 3-Tier-Architektur ist ein zentraler Punkt, an dem die Datenkonsistenz über Transaktionen in Form von *Two-Phase-Commit* sichergestellt werden kann.
In einer Microservice-Architektur wird jedem Service eine eigene Datenbank mit eigener Datenhoheit zugesprochen und somit nicht nur Geschäftslogik, sondern auch Geschäftsdaten verteilt. Operationen auf verteilten Geschäftsdaten von verschiedenen Microservices erfordern ein Abwägen zwischen Konsistenz der Daten oder Verfügbarkeit des Systems.

2.7.2 Conway's Law

Der Begriff wurde von Frederick P. Brooks, Jr. geprägt. Er hat eine Veröffentlichung von Melvin E. Conway zitiert und sie als Conway's Law bezeichnet (Brooks, 1995). Das Gesetz lautet (Conway, 1968):

> „...organizations which design systems (in the broad sense used here) are constrained to produce designs which are copies of the communication structures of these organizations."

Obwohl dieses Gesetz für Systeme unterschiedlicher Arten gilt, schließen wir für Softwaresysteme daraus, dass sich die Kommunikationsstruktur der entwickelnden Organisation im Systemdesign und somit in der Software-Architektur wiederfindet.

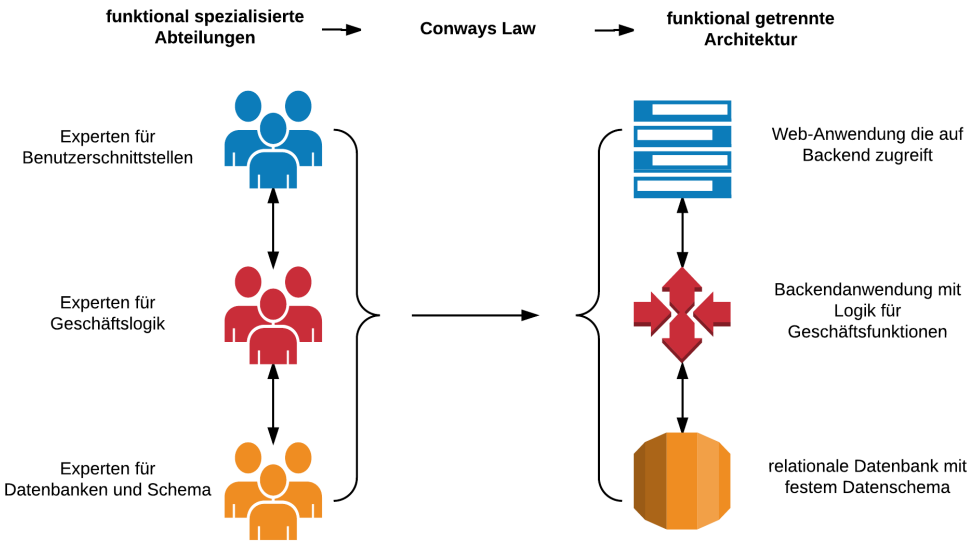

Abbildung 2.5: Funktional aufgeteilte Teams führen zu Systemen, die nach funktionalen Schichten unterteilt sind. In Anlehnung an (Fowler & Lewis, 2014).

Abbildung 2.5 zeigt eine, nach funktionalen Gruppen aufgeteilte, Organisation der Entwicklungsabteilung. Jedes Team besteht nur aus Spezialisten für ihren Bereich. Würde man diesen Teams die gemeinsame Aufgabe übertragen, ein neues System zu entwickeln, erhält man nach Conway's Law mit hoher Wahrscheinlichkeit ein Softwaresystem in der klassischen 3-Tier-Architektur. Das Datenbankteam würde sich um die einzelnen Datenschemata und relationalen Zusammenhänge kümmern. Die Entwickler für die Geschäftslogik würden die Logik des Systems beispielsweise als eine Anwendung in einer objektorientierten Hochsprache entwickeln, wobei der Grad der Modularisierung der Anwendung von der Kommunikation innerhalb des Teams abhängig sein würde. Das Team für die Benutzerschnittstellen würde für ein geeignetes Layout und eine passende Darstellung sorgen. Zwischen Datenbank, Logik und UI wird es spezifizierte Schnittstellen geben, da die Kommunikation zwischen den Teams naturgemäß nicht so ausgeprägt ist wie innerhalb eines Teams.

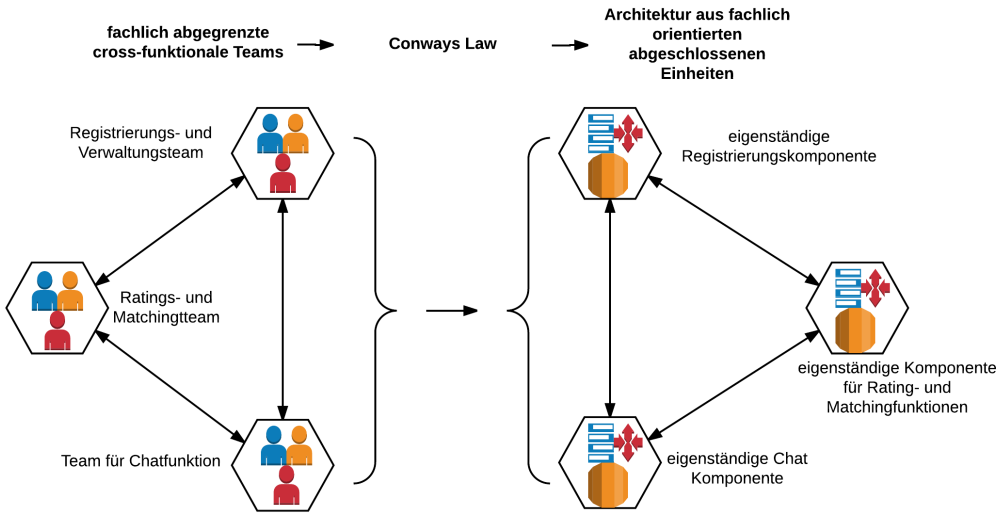

Abbildung 2.6: Mehrere gemischte Teams führen zu einem System aus mehreren gekapselten Einheiten (Fowler & Lewis, 2014).

Abbildung 2.6 zeigt eine andere Struktur in der es mehrere Teams gibt, die jeweils einzelne Spezialisten aus den funktionalen Bereichen enthalten. Idealerweise sind diese Teams für eigene fachliche Bereiche zuständig (vgl. Abschnitt 2.7.3.1). Da nun jedem Team die volle Kompetenz für das Entwickeln einer kompletten Anwendung bereitsteht, kann sich das Team eigenständig um ihren Bereich kümmern und die Software mit den erforderlichen Eigenschaften versehen. Die Kommunikation innerhalb des Teams kann auf kürzeren Wegen geschehen als über Abteilungsgrenzen hinweg. Änderungen, die alle Funktionalitäten betreffen, sind so einfacher durchzuführen. Die globale Struktur des Systems besteht nun aus gekapselten Anwendungen, da die Kommunikation zwischen den fachlichen Teams in der Regel nicht informell stattfindet, sondern sich auf Funktionen an der Schnittstelle beschränken sollte, die ein Team von einem anderen benötigt.

2.7.2.1 Conway's Law und Microservices

Microservices sollen einzelne eigenständige Anwendungen sein und sich unabhängig von anderen weiterentwickeln können. Mit einer klassischen Unternehmensorganisation nach funktionalen Bereichen wird man Schwierigkeiten bei der Umsetzung der einzelnen Services haben.
Im Zusammenhang mit Microservices ist Conway's Law eher invers zu betrachten. Die Auswirkungen von Conway's Law können genutzt werden, um die gewünschte Microservices-Architektur mehr oder weniger automatisch zu erhalten, indem die funktionale Unternehmensorganisation zugunsten von kleinen und multifunktionalen Teams aufgegeben wird.

2.7.3 Domain Driven Design (DDD)

Domain Driven Design ist eine Sammlung von Patterns, um Software für komplexe Anwendungsdomänen strukturiert zu modellieren und umzusetzen. Generell setzt Domain Driven Design stark darauf, die fachlichen Zusammenhänge der Domäne als Kernpunkte für die Struktur einer Software zu betrachten. Wesentlich im DDD ist das strategische Design, wobei die Gesamtkomplexität einer Anwendungsdomäne in verschiedene Fachdomänen unterteilt wird, welche einen *Bounded Context* bilden.

2.7.3.1 Bounded Context

Das Ziel ist es, für jede Fachdomäne eine allumfassende Sprache zu entwickeln, die für Fachexperten und Softwareentwickler gleichermaßen verständlich ist und als Basis für Kontext-lokale Probleme der Software dienen soll. Daraus lässt sich ein Domänenmodell ableiten, welches ausschließlich für den abgegrenzten Kontext gültig ist. Es beinhaltet eine Repräsentation der konkreten fachlichen Muster. Die wichtigsten 3 Muster sind:

- **Entity:** Objekt mit einer Identität, zum Beispiel ein Benutzer.
- **Value Object:** Wertobjekt ohne eigene Identität und nur gültig in Verbindung mit einem Entity, zum Beispiel die Anzahl der Kurse, in die ein Benutzer eingeschrieben ist.
- **Aggregate:** Zusammengesetztes Domänenobjekt, zum Beispiel eine Gruppe mit mehreren Benutzern.

Abbildung 2.7 veranschaulicht anhand der Tutinder-Anwendung unterschiedliche Domänenmodelle des Benutzers in verschiedenen Bounded Contexts.
Für den *Courses and Enrollment*-Kontext sind beispielsweise Namen, Profilbild, Logindaten oder Persönlichkeitsbeschreibungen uninteressant, weil sie nicht verwendet werden. Einzig der Studiengang und die Kurse, in die ein Benutzer schon eingeschrieben ist, sind interessant, um entscheiden zu können, ob eine neue Einschreibung valide ist oder nicht.
Für *Registration und User Management* sind Kurse nicht relevant, dafür aber Kontaktinformationen wie die E-Mail für eine eventuelle Passwortrücksetzung.
Im Kontext von *Rating und Matching* werden wiederum andere Attribute benötigt, die es ermöglichen einen hohen Übereinstimmungsfaktor zu berechnen und potentiell gut passende Partner zuerst vorzuschlagen. Hierbei sind Persönlichkeitsbeschreibungen und bisher getätigte Bewertungen über andere Benutzer hilfreich und daher relevant.

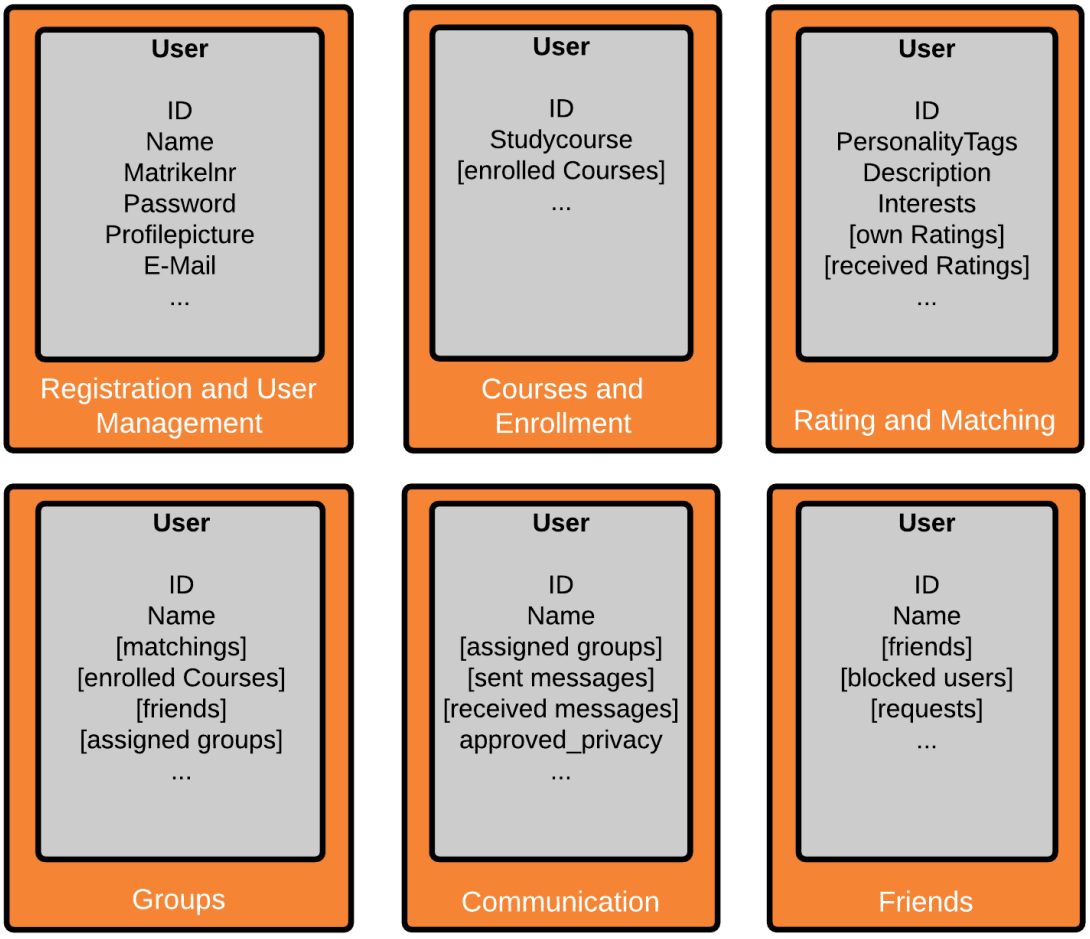

Abbildung 2.7: Beispielhafte Bounded Contexts der Tutinderanwendung mit unterschiedlichen Ansichten auf das User-Entity

Indem man jedem Kontext sein eigenes Modell überlässt, kann dieser sich selbständig weiterentwickeln, ohne dass Änderungen an dem Datenmodell durch andere Kontexte blockiert werden. Möchte das Team für den Bereich *Communication* in Zukunft Statistiken über die Antwortzeit eines Benutzers erheben, kann das Team sein eigenes Modell um ein Attribut erweitern ohne, dass andere Teams diesem Modell ebenfalls zustimmen müssen. Genauso können nicht länger benutzte Attribute von ehemaligen Features problemlos entfernt werden, was für ein übersichtlicheres Datenmodell sorgen kann.

Als gegensätzliche Alternative gibt es das Enterprise Integration Pattern *„Canonical Data Model (CDM)“*, bei welchem ein großes Datenmodell für jede Entität entworfen wird, welches dann über die Fachdomänen hinweg gültig und an den Schnittstellen der Domänen bindend ist (Wolff, 2016).

2.7.3.2 Domain Driven Design und Microservices

Ein CDM widerspricht dem Prinzip der unabhängigen Entwicklung von Microservices. Die Teams müssten sich in einem langwierigen Prozess gemeinsam auf ein global gültiges Domänenmodell einigen. Anschließend müsste jede Änderung am Model mit allen anderen Nutzern dieses Models abgesprochen und an allen Schnittstellen integriert werden.
Ein weiteres Problem sind mehrdeutige Wörter. Bestimmte Begriffe wie zum Beispiel „Bank“ haben in verschiedenen Fachdomänen unterschiedliche Bedeutungen. Im Kontext von finanziellen Angelegenheiten gehen die Fachdomänen vom Kreditinstitut aus. In anderen Domänen könnte damit aber auch das Gebäude des selbigen Institutes gemeint sein oder das gleichnamige Möbelstück.

Eine Abgrenzung anhand von Bounded Contexts ist ein guter Anhaltspunkt für die Abgrenzung von Microservices. In der Regel sind gut geschnittene Bounded Contexts gleichbedeutend mit einem Microservice, wobei es den Kontext-Teams prinzipiell überlassen ist, ihren Kontext in mehrere Microservices zu unterteilen, falls sie der Meinung sind, dass ein Service zu groß ist oder seine Funktionalität keine Kohäsion beherbergt. In diesem Fall gilt das Domänenmodell dieses Bounded Context für beide Microservices, was kein Problem ist, solange diese beiden Microservices einzig von diesem Team betreut werden.
Zu vermeiden ist jedoch ein Microservice, der Verantwortlichkeiten über mehrere Bounded Contexts hinweg abdeckt und somit von zwei Teams verwaltet werden würde. Das würde durch unklare Verantwortlichkeiten und ein inkonsistentes Domänenmodell früher oder später zu Missverständnissen oder aufgeblähten Datenmodellen führen.

2.7.4 Continuous Delivery

Continuous Delivery (CD) ist ein Ansatz in der Softwareentwicklung, um Änderungen an einer Software zuverlässig und reproduzierbar in Produktion zu bringen (O'Reilly, Molesky & Humble, 2014). Jez Humble et al. haben hierfür Anforderungen erarbeitet und beschrieben, die für erfolgreiches CD erfüllt sein müssen.

Sicherheit CD beruht im Kern auf einer Deployment Pipeline von verschiedenen Tests. Diese sind größtenteils automatisiert und werden durch verschiedenen manuellen Tests ergänzt. Das Durchlaufen dieser Pipeline kann als Validierung der Software gesehen werden und in Kombination mit den richtigen Log-Daten als Prüfprotokoll verwendet werden. Kernpunkt für die Sicherheit ist die Abdeckung und Qualität der Tests. Jeder Test arbeitet nach dem Prinzip, die Fehlerfreiheit der getesteten Software zu widerlegen. Schafft dies kein Test und es bestehen trotzdem Zweifel an der Produktionsfähigkeit der Software, müssen die Tests erweitert werden.

Schnelligkeit Hauptmesskriterium für erfolgreiches Continuous Delivery ist die Geschwindigkeit mit der Änderungen in Produktion gelangen. So zielt CD darauf ab, mit jeder neuen Änderung mehr über die Qualität und die Nutzer der Software zu lernen. Für die verantwortlichen Teams ist ein schnelles Feedback zu getätigten Änderungen enorm wichtig. Zusätzlich versucht CD, langwierige Integrationstests durch Einbeziehen von Continuous Integration (vgl.

Abschnitt 2.7.4.1) zu vermeiden.
Die Zeit, die es dauert, die Deployment Pipeline zu durchlaufen, soll durch ständige Verbesserung immer weiter verkürzt werden. So ist es wünschenswert, dass das Beheben eines Fehlers durch Ändern weniger Codezeilen innerhalb von ein paar Minuten als „reif für die Produktion“ validiert werden kann. Hierfür ist eine immer stärkere Automatisierung erforderlich. Mit DevOps (vgl. Abschnitt 2.7.5) eng verbunden, ist dabei auch das automatisierte Bereitstellen von Testumgebungen für Akzeptanz- und Kapazitätstests.

Nachhaltigkeit Hinter CD verbirgt sich zu Beginn sehr viel zusätzlicher Aufwand, der durch die Automatisierung des Deployments und das Schreiben der Tests entsteht. Zusätzlich benötigt man dafür auch eine Infrastruktur und Software-Architektur, die einen hohen Grad an Automatisierung unterstützt. Das langfristige Ziel ist aber, es wirtschaftlich sinnvoll zu machen, in kleinen Schritten zu denken und selbst kleine Änderungen direkt in den Produktionsbetrieb zu bringen. CD ist daher als nachhaltige Investition in mehr Agilität zu sehen.

Team Das Team hat die Verantwortlichkeit, zu jedem Zeitpunkt eine aktuelle und funktionierende Version im Hauptbranch zu haben, die guten Gewissens in den Produktionsbetrieb gebracht werden kann. Zeigt sich bei einer Version innerhalb der Pipeline ein Problem, muss der verantwortliche Entwickler dieses innerhalb von wenigen Minuten lösen oder seine Änderungen von dem Hauptbranch zurücknehmen.
Die Definition, wann ein Entwickler seine Arbeit als fertig ansieht verlagert sich von dem Zeitpunkt, wo eine Änderung fehlerfrei in den Hauptbranch integriert wurde hin zu dem Zeitpunkt, ab dem die Änderung fehlerfrei durch alle Tests gelaufen ist und in Produktion gebracht werden kann.

2.7.4.1 Continuous Integration

Continuous Integration (CI) ist eine Praxis, die sich mit dem Integrieren von Änderungen an der Codebasis einer Software beschäftigt. Sie versucht, Probleme beim Integrieren von verschiedenen Änderungen an einer Software zu minimieren, Fehlersuche zu erleichtern und die Reaktionsgeschwindigkeit von Änderungen zu erhöhen.

Angenommen es arbeiten mehrere Entwickler an verschiedenen Features einer Software. Die Features werden unabhängig voneinander fertig entwickelt, um sie dann nach mehreren Wochen Arbeit miteinander in den Hauptbranch des Softwareversionierungssystems zu integrieren. Was bei wenigen Entwicklern und überschaubaren Änderungen mit einer exakten Planung noch gut funktionieren kann, wird bei hunderten von Entwicklern mit hoher Wahrscheinlichkeit zu Problemen führen, da die verschiedenen Features unvorhergesehene Abhängigkeiten aufweisen. Im besten Fall zeigen sich diese Probleme durch ein Fehlschlagen beim Kompilieren einer aktuellen Version der Software. Im schlechtesten Fall erfahren die Entwickler aber erst Wochen später von Problemen, da die Abhängigkeiten erst in manuellen Tests oder im Produktionsbetrieb auftreten. Fehlerbehebung an einer großen Menge von Code wird dann durch die zeitliche Distanz von Entwicklung und Auftreten des Fehlers zusätzlich erschwert.

Continuous Integration fordert daher, dass Entwickler ihre Änderungen nicht wochenlang in eigenen Branches fortentwickeln und dann das gesamte Paket zusammen integrieren. Vielmehr sollen kleinere Teilpakte in kurzen regelmäßigen Abständen in den Hauptbranch integriert werden. Das erfordert zum einen Automatisierung von Kompilierung- und Testvorgängen, zum anderen aber auch ein Umdenken bei der Art, wie Entwickler Features umsetzen. Entwickler müssen ihre Änderungen so durchführen, dass sie in der Lage sind, am besten täglich eine lauffähige Version ihres aktuellen Standes mit den Ständen der anderen Entwickler zu integrieren. Hierdurch ergibt sich eine geringere Distanz zwischen Fehlereinbringung und Erkennung, was Risiko und Kosten für die Fehlerbehebung minimiert.
Zusätzlich schafft CI Fakten. Der Entwicklungsfortschritt ist besser abgebildet, da die umgesetzten Features alle sofort zentral integriert werden und somit immer ein aktueller Stand der Entwicklung vorhanden ist.
Insgesamt ergibt sich auch eine bessere Vorhersagbarkeit, wenn die zentrale CI-Infrastruktur dieselben Plattformversionen wie eine Produktionsinfrastruktur verwendet. Alle Änderungen werden so täglich auf einer Art Referenzinfrastruktur getestet. So ist das Risiko, dass Bibliotheken im Betrieb Probleme verursachen, viel geringer, als wenn die Entwickler nur auf ihrer lokalen Entwicklungsinfrastruktur testen (Duvall, Matyas & Glover, 2007), (Fowler, 2006).

2.7.4.2 Deployment Pipeline

Im Kern besteht CD aus einer Deployment Pipeline, die nacheinander mehrere Tests ausführt und in angemessener Zeit Rückmeldung erteilt. Diese Tests sind größtenteils automatisiert, können aber auch menschliche Tests und manuelle Freigaben enthalten. So durchläuft die veränderte Software nacheinander verschiedene Phasen.

Commit-Phase Hierbei wird die Änderung der Software in ein Versionierungssystem eingecheckt und automatisch kompiliert. Anschließend werden Unit-Tests durchgeführt, um eine erste Korrektheit sicherzustellen. Wichtig für diese Phase ist auch ein guter Continuous Integration Prozess, um das Integrieren von Änderungen mehrere Entwickler zu erleichtern.

Akzeptanztests In dieser Phase wird die fachliche Korrektheit überprüft. Automatisierte Tests überprüfen, ob die fachlichen Anforderungen von der Software vollständig erfüllt werden.

Kapazitätstests Eine Software, die fachlich korrekt ist, muss noch lange nicht performant sein. Deshalb wird die Software nach einer Änderung Lasttests unterzogen, um zu überprüfen ob eine Änderung ein Bottleneck geschaffen hat. So kann es passieren, dass eine Software zwar die fachlichen Anforderungen erfüllt, aber wegen einem schlechten Programmierstiel unter Last mit großen Arbeitsdaten nicht mehr die nötige Performanz aufweist.

Explorative Tests Erst wenn die vorherigen Bedingungen erfüllt sind, wird der Zweck der eigentlichen Änderung getestet. So werden hier meist manuell die Features, welche zu der Änderung geführt haben getestet.
Ein bekanntes Beispiel ist hierbei *A/B-Testing*, wobei bis auf einen Unterschied zwei identische

Versionen der Software parallel in Produktion betrieben werden. Durch Lastverteilung kann so das Nutzerverhalten auf den unterschiedlichen Versionen getestet und analysiert werden.

Produktion Werden auch die explorativen Tests für gut erachtet, so wird diese Version der Software als aktuellstes Build in Produktion gebracht. Zur Sicherheit kann dies auch noch schrittweise über *Canary Releases* erfolgen (Sato, 2014). Hierbei werden ähnlich dem A/B-Testing die neue und alte Version parallel betrieben und schrittweise die Last auf der neuen Version erhöht. Arbeitet die neue Version zufriedenstellend und stimmen die entsprechenden Metriken, kann die alte Version abgeschaltet werden.

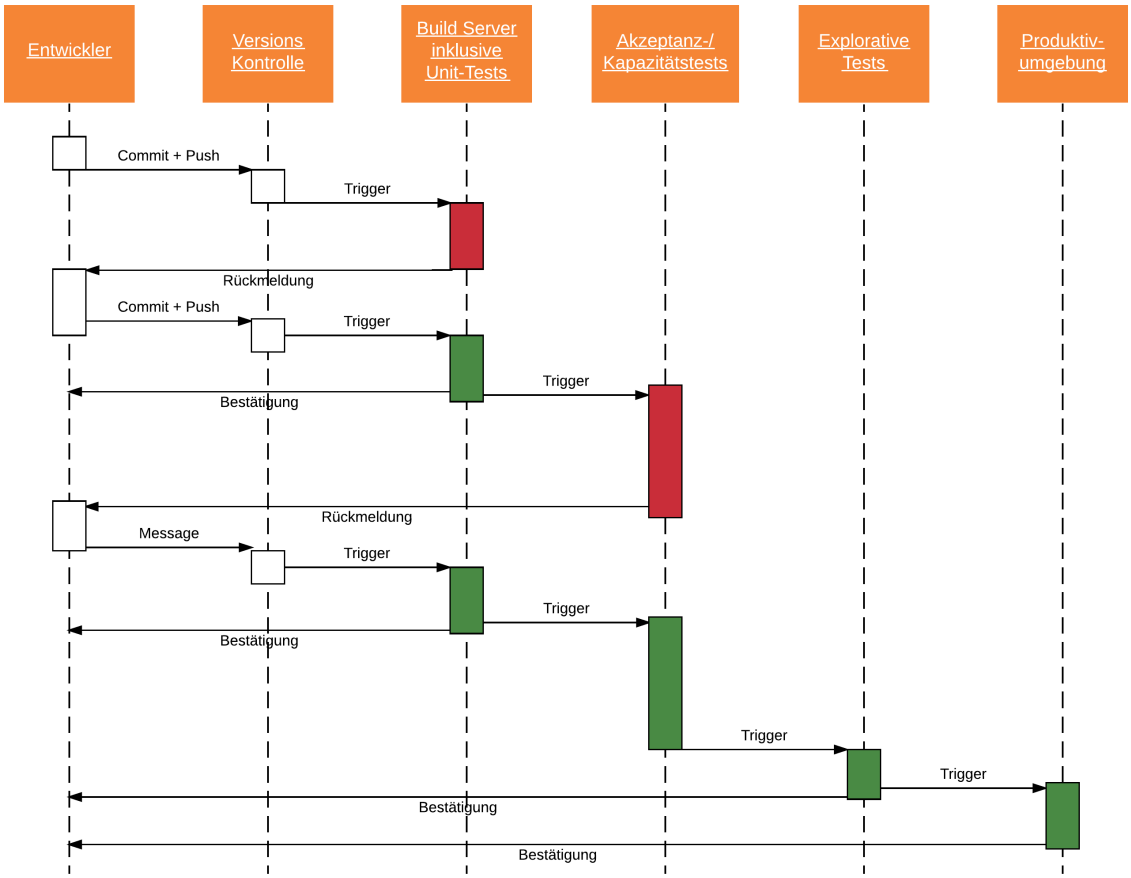

Abbildung 2.8: Diagramm der Deployment Pipeline. In Anlehnung an (O'Reilly et al., 2014).

2.7.4.3 Continuous Delivery und Microservices

Microservices und Continuous Delivery sind sehr eng verbunden (Wolff, 2016). Sie erleichtern ein Umsetzen eines Continuous Delivery Prozesses. Ein kleiner Microservice kann eine Pipeline viel schneller durchlaufen und Entwicklern innerhalb einer angemessenen Zeit Rückmeldung

über den Effekt ihrer Änderung geben.
Andererseits ist eine sinnvolle Umsetzung des Microservice-Stils auf eine funktionierende Deployment Pipeline angewiesen. Das automatisierte Durchlaufen dieser Pipeline erleichtert es, viele Microservices zuverlässig in den Produktionsbetrieb zu bringen. Manuelles Verwalten und Testen der verschiedenen Versionen von vielen kleinen Microservices wäre unnötiger Mehraufwand und würde die gewonnene Flexibilität für Softwareänderungen stark einschränken.

2.7.5 DevOps

DevOps ist ein zusammengesetzter Begriff aus „Development“ und „Operations“. Er beschreibt eine engere Form der Zusammenarbeit und Kooperation zwischen Entwicklern und Administratoren.
In erster Linie geht es darum, die Ziele beider Lager (Entwicklung und Betrieb) in Betracht zu ziehen, um damit die strategischen Geschäftsziele beim Betrieb von Softwaresystemen zu realisieren.

> DevOps is an organizational approach that stresses empathy and cross-functional collaboration within and between teams – especially development and IT operations – in software development organizations, in order to operate resilient systems and accelerate delivery of changes (Dyck, Penners & Lichter, 2015).

Der Betrieb ist verantwortlich dafür, immer komplexer werdende Softwaresysteme zuverlässig zu betreiben und daher interessiert daran, möglichst wenige Änderungen vorzunehmen. Getreu dem Motto *„Never touch a running System“*, birgt jede Veränderung des Systems neue Fehler oder ein Ausfall des Systems. Das Ergebnis sind Release-Pläne, welche Wochen im Voraus geplant werden müssen. Dieser Umstand ist für kundenorientierte und innovative Unternehmen zu behäbig und unflexibel.
Entwickler, die kundenorientiert arbeiten sollen, benötigen Nutzerfeedback, um die richtigen Features zu entwickeln und Änderungen an den richtigen Stellen anzusetzen. Das Ziel von Entwicklern ist es daher, den Zyklus von Änderung und Feedback zu beschleunigen. Fix geplante Release Termine sind für diesen Zyklus extrem limitierend.
DevOps versucht diesen Widerspruch durch Kollaboration und Vermengung der Kompetenzen beider Seiten aufzulösen. Die Anwendung von DevOps ist die konsequente Fortsetzung einer nach fachlichen Aspekten veränderten Unternehmensorganisation.

Konkret kann sich DevOps durch eine veränderte Organisation von Entwicklung und Betrieb zeigen. So kann eine verstärkte Zusammenarbeit durch das Integrieren von Operations-Spezialisten in fachliche Teams, die von Ausarbeitung der Anforderungen bis hin zum Betrieb gemeinsam für einen bestimmten Bereich verantwortlich sind. Beispielsweise werden Kompetenzen von Betriebsverantwortlichen schon in der Entwicklung miteinbezogen und so eine der Produktivumgebung ähnliche Entwicklungsumgebung geschaffen.
Gleichzeitig ist die Kommunikation zwischen Entwicklern und Betriebspersonen innerhalb eines Teams einfacher als über Abteilungsgrenzen hinweg.
Von technischer Seite zeigt sich DevOps durch eine weitreichende Automatisierung. So sind Continuous Intergration, Continuous Delivery bis hin zum vollautomatisierten Deployment zentrale Punkte für DevOps (Brunnert et al., 2015). Auch die Verwaltung der Infrastruktur lässt sich durch *Infrastructure as a Service* (IaaS) und *Platform as a Service* (PaaS) Angeboten

immer weiter automatisieren. Hierbei spielt *Infrastructre as Code* eine zentrale Rolle für die Bereitstellung von standardisierter und passender Infrastruktur für die jeweilige Software.

2.7.5.1 DevOps und Microservices

Einige Ziele von DevOps und Microservices gehen in eine ähnliche Richtung. So soll auch DevOps dazu dienen, leichtere und schnellere Änderungen in der Software zu ermöglichen, konkret soll der Zyklus von Änderung, Deployment, Feedback und erneute Änderung beschleunigt werden. Eine gute strukturierte Microservice-Architektur unterstützt dieses Ziel.
Zugleich wirkt sich DevOps positiv auf die Unabhängigkeit der fachlichen Teams aus, da jedes Team auch Kompetenzen für den Betrieb erhält und dadurch eigenmächtig über den Release von Änderungen entscheiden kann.

Genau wie der Microservice-Stil ist auch DevOps sehr stark von der Cloud geprägt. Eine immer stärkere Nutzung von IaaS, die zum großen Teil über eine API gesteuert wird, erfordert, dass sich auch Operations-Spezialisten mehr Kompetenzen aus der Software-Entwicklung aneignen. Ohne diese Kompetenzen lässt sich eine große Microservice-Architektur nicht vernünftig managen. In den großen Technologie Unternehmen, die schon Microservices einsetzen, hat sich deshalb eine rege DevOps-Community gebildet, die in Form von Open-Source praktische Deployment und Monitoring Tools entwickelt. Gleichzeitig sind Entwickler der Microservices aber auch in der Pflicht, das automatisierte Monitoring durch sinnvolles Logging zu unterstützen.

2.7.6 Skalierbarkeit

Skalierbarkeit beschreibt die Fähigkeit, durch das Hinzufügen von angemessenen Ressourcen zu einem System die Leistungsfähigkeit desselben zu erhöhen, um damit steigende Last auszugleichen oder die Geschwindigkeit zu verbessern.

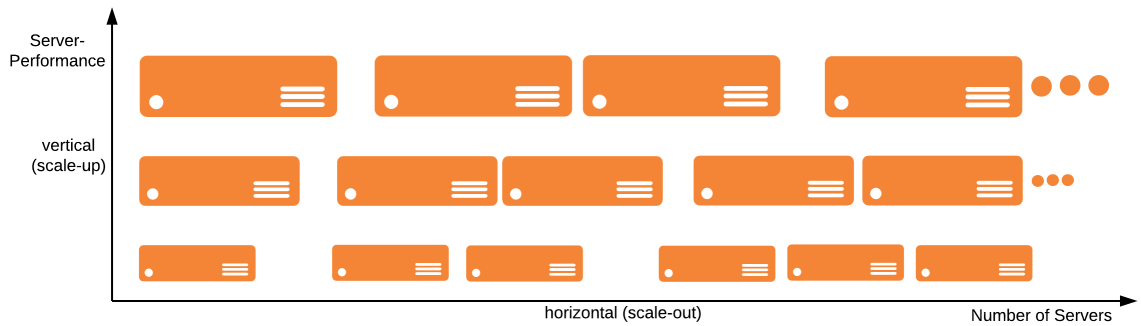

Abbildung 2.9: Horizontale und Vertikale Skalierung am Beispiel von Servern im Vergleich. Angelehnt an (Lantin, 2014).

Grundsätzlich unterscheidet man in zwei Varianten von Skalierung (vgl. Abbildung 2.9):

Horizontale Skalierung Bezeichnet eine Vergrößerung der Ressourcen in die Breite. Der Pool von verfügbaren Ressourcen wird zahlenmäßig vergrößert. In der Abbildung werden somit mehrere Server parallel betrieben, um die Leistungsfähigkeit zu erhöhen. Im Falle von Personal könnte damit auch das Einstellen von mehreren gleichwertigen Mitarbeitern gemeint sein.

Vertikale Skalierung Anstatt mehrere Ressourcen einzusetzen, werden bei der vertikalen Skalierung vorhandene Ressourcen durch leistungsfähigere ersetzt. So wird in der Abbildung ein schwacher Server durch einen leistungsfähigeren Server ersetzt. Bezogen auf Personal könnte damit das Qualifizieren von Mitarbeitern bezeichnet werden.

Beide Skalierungsarten haben ihre Daseinsberechtigung. So kann in kleinem Rahmen eine vertikale Skalierung Sinn machen, um die Komplexität eines verteilten Systems zu vermeiden. Jedoch ist der vertikale Bereich stark limitiert und die Kosten für sehr leistungsfähige Ressourcen sind oft sehr viel höher, wie wenn die benötigte Leistung durch mehrere schwächere Ressourcen abgedeckt wird (Wolff, 2016).
Die horizontale Skalierung bietet zusätzlich den Vorteil, das System geographisch zu verteilen, was zusätzlich positive Auswirkungen auf die Reaktionszeit des Systems haben kann. Gleichzeitig kann damit aber auch die Verfügbarkeit erhöht werden, da das System beim Ausfall einer Ressource trotzdem seinen Dienst erfüllt, wenn auch mit verminderter Leistung.

Horizontale Skalierung kann sehr vielfältig ausfallen und auf mehrere Arten geschehen. Fisher et al. unterteilen sie in drei Dimensionen und stellen diese in der *AFK-Scale-Cube* (vgl. Abbildung 2.10) dar (Fisher, 2008). Wobei der Nullpunkt links unten eine monolithische Anwendung auf einer einzelnen Ressource beschreibt. Rechts oben wäre eine Anwendung, die aus mehreren eigenständigen Microservices besteht (Y-Achse), welche ihrerseits mehrfach geklont sind (X-Achse) und die Datenbanken beispielsweise nach geographischen Regionen getrennt sind (Z-Achse).

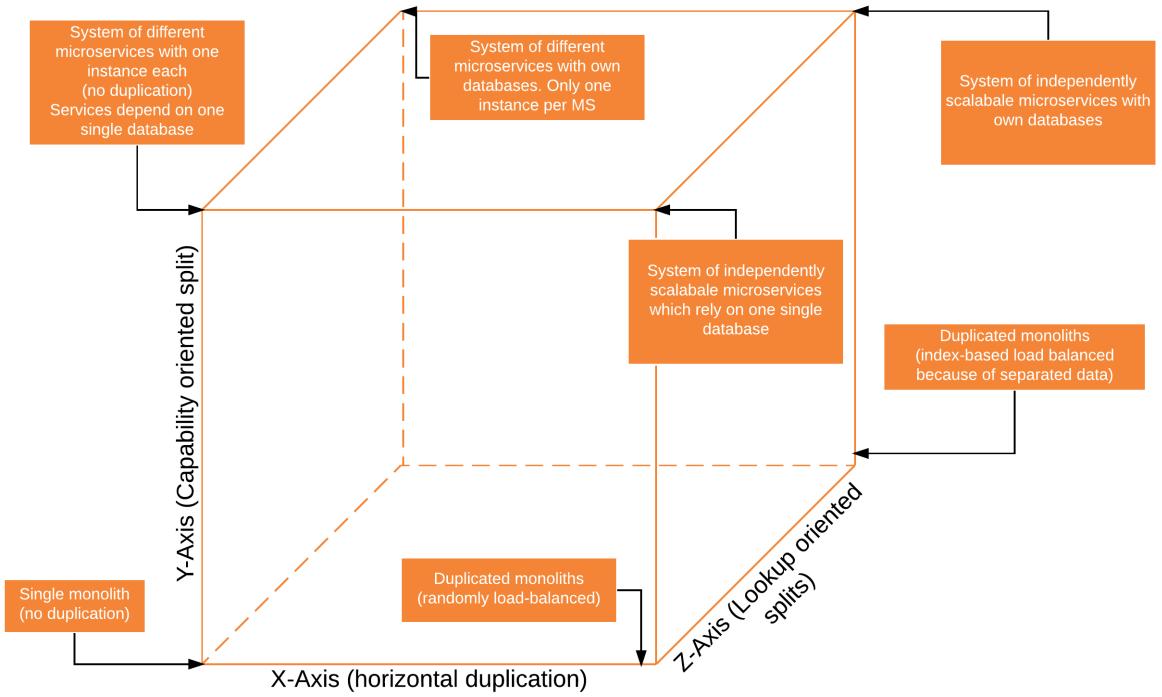

Abbildung 2.10: AFK Scale-Cube. Zeichnung angelehnt an (Fisher, 2008).

- **X-Achse (Horizontales Duplizieren):** Diese Art der Skalierung gilt als die klassische Art. Es werden mehrere identische Kopien einer Software hinter einer Lastverteilung betrieben und somit die Kapazität und Performance erhöht. Der Vorteil liegt in der Einfachheit der Skalierung, es genügt lediglich eine lastverteilende Komponente vorzuschalten und mehrere Instanzen der Software zu nutzen. Ein Verändern der Softwarearchitektur ist in der Regel nicht nötig. Die Anwendung sollte aber zustandslos sein, um sich ein komplexes Routing bei der Lastverteilung zu sparen.
- **Y-Achse (Funktionelle Aufteilung):** Bei der funktionellen Aufteilung werden Anwendungen in eigenständig ausführbare Komponenten aufgesplittet. Microservices, die anhand von Bounded Contexts aufgeteilt wurden, sind ein Beispiel für solche Komponenten. Man erreicht dadurch eine unabhängige Skalierung der Komponenten und kann im Vergleich zur X-Achsen Skalierung Rechenressourcen effizienter nutzen. Jede in Y-Achse aufgeteilte Komponente lässt sich zusätzlich einzeln in X- und Y-Achse skalieren.
- **Z-Achse (Daten Aufteilung):** Diese Art ähnelt sehr stark der X-Achsen Skalierung, da hier ebenfalls mehrere identische Softwareinstanzen hinter einem Lastverteiler betrieben werden. Hinzu kommt hierbei jedoch, dass jede Instanz nur für einen Teil der Daten zuständig ist und der Lastverteiler zum Beispiel anhand von Indexschlüsseln der angefragten Ressourcen zu den zuständigen Instanzen routet. Charakteristisch für diese Art der Skalierung ist die Abhängigkeit zu dem jeweiligen Anfragekontext. Datenunterteilung kann zum Beispiel anhand von geographischen oder kundenspezifischen Abgrenzungen erfolgen. Die Schwierigkeit liegt darin, eine sinnvolle Aufteilung der Daten zu finden.

2.7.6.1 Skalierbarkeit und Microservices

Eine effiziente und gute Skalierung wird für Unternehmen immer wichtiger. Die Cloud bietet dafür durch flexible IaaS-Angebote perfekte Voraussetzungen, um Rechenleistung innerhalb von wenigen Sekunden bereitzustellen. Wie im eigenen Rechenzentrum gilt auch in der Cloud, dass der vertikalen Skalierung Grenzen gesetzt sind und ab einer gewissen Größe ohne elastische horizontale Skalierung ein zuverlässiger Betrieb nicht profitabel ist.
Microservices bieten eine Möglichkeit, einzelne Komponenten sehr schnell und präzise zu skalieren und versetzen Unternehmen in die Lage unerwartete Lastspitzen in einzelnen Komponenten abzufangen. Das Ausrollen von monolithischen Anwendungen ist dafür aufgrund der Größe und längeren Startzeit oft nicht geeignet. Außerdem werden wertvolle Ressourcen für nicht genutzte Komponenten vorgehalten, weil immer die ganze Software dupliziert werden muss.

3 Charakteristik

Fowler und Lewis haben den Microservice-Stil anhand verschiedener Charakteristiken beschrieben. Dieses Kapitel beschreibt und wertet diese gegenüber einem monolithischen Stil (Fowler & Lewis, 2014).

3.1 Componentization via Services

Software in einzelne Komponenten aufzuteilen, welche über definierte Schnittstellen genutzt werden können, ist mit Paradigmen wie der objektorientierten Programmierung längst etabliert. Komponenten sind dort meist als einzelne Bibliotheken eingebunden und werden innerhalb desselben Prozesses aufgerufen. Diese sind dank der definierten Schnittstellen zwar auch einzeln austauschbar und veränderbar, das hat aber zur Folge, dass die gesamte Anwendung neu gebaut und deployt werden muss.
Sind Komponenten als eigenständige Microservices realisiert, muss bei internen Änderungen nur der entsprechende Microservice neu kompiliert und getestet werden. Die einzelnen Komponenten erreichen so wesentlich kürzere Build- und Testdurchlaufzeiten, da jeweils nur kleine Teile des Systems die entsprechenden Prozesse durchlaufen müssen. Ein weiterer Vorteil ist die strikte Trennung der Komponenten. Diese verhindert, dass sich ungewollte Abhängigkeiten einschleichen und kann dazu beitragen, dass die Software eine saubere Struktur behält und somit leichter zu warten ist. Entwickler können von anderen Komponenten nur nutzen, was diese explizit über ihre externe Schnittstelle zur Verfügung stellen. Die Entwickler eines Microservice können so sicher sein, dass interne Änderungen, bei denen die Schnittstellenspezifikation erhalten bleibt, keine Auswirkung auf andere Microservices hat. Bei vielen Programmiersprachen ist das ohne aufwändige Tools und Codeanalyse nicht möglich, weil Funktionen von fremden Modulen willkürlich genutzt werden können, was zu ungewollten Abhängigkeiten führt.

3.1.1 Herausforderung: Auswirkungen von Schnittstellenänderungen

Die Unabhängigkeit der Services geht verloren, sobald Schnittstellen geändert werden müssen. Der Nachteil bei Microservices ist hierbei, dass die Auswirkungen der Schnittstellenänderungen nicht direkt ersichtlich sind. Als unabhängige Deployment-Einheiten kann der Compiler keine Inkonsistenzen bei der Verwendung von Schnittstellen zwischen zwei Services feststellen. Als Konsequenz lassen sich Fehler erst in Produktion oder in aufwändigen Integration-Tests erkennen. Eine Methode, mit der Entwickler die Korrektheit ihrer Schnittstellenänderung überprüfen können, sind *Consumer-Driven Contract Tests*.

3.1.1.1 Consumer-Driven Contract Test

Schnittstellen Spezifikationen in Form von Verträgen gehören zu den Grundprinzipien in verteilten Softwaresystemen.
In Form von *Provider-Contracts* legt der Service fest, welche Operationen mit welchen Datenstrukturen und Anforderungen er anbietet. Konkrete Beispiele für Provider-Contracts sind die von Web-Services bekannten *WSDL-Verträge* oder automatische Dokumentationssysteme wie *Swagger* für REST-Schnittstellen.
Für Consumer-Driven Contract Testing wird ein Vertrag in Form eines *Consumer-Contracts* benötigt. Er beschreibt, welche Funktionalität ein Consumer von einem Service tatsächlich nutzt und welche Datenfelder dabei verwendet werden. Zusammengenommen ergeben die Verträge von allen Consumern den *Consumer-Driven Contract (CDC).* Dieser gilt als minimale funktionelle Anforderung für den Service, da darin die komplette extern genutzte Funktionalität des Service aufgeführt ist.
Beim *Consumer-Driven Contract Testing* geht es darum, seinen Service gegen diesen Consumer-Driven Contract zu testen. Jede Änderung wird drauf überprüft, ob noch alle Anforderungen der Consumer erfüllt werden.
CDC-Testing kann als eine Art Unit-Test für Service-Schnittstellen gesehen werden. Konsequenterweise sollte der CDC-Test somit in einem Continuous Integration Prozess (vgl. Abschnitt 2.7.4.1) integriert sein, da keine Version des Service in Produktion gelangen sollte, die Fehler in anderen Consumern verursachen würde.

Beispiel: Pact ist ein Werkzeug mit dem CDC-Testing realisiert werden kann. Es ist in verschiedenen Programmiersprachen verfügbar und kann daher auch in einer mehrsprachigen Umgebung verwendet werden. Der Consumer definiert seine spezifischen Anforderungen an den Producer mit Hilfe der Pact-Library in seiner Programmiersprache. Anschließend kann ein Pact-Mock-Server gestartet werden, der ein Pact-Datei erzeugt, in welcher der Vertrag mittels JSON spezifiziert ist. Diese Datei wird in einem Pact-Broker oder direkt im CI-Repository abgelegt, sodass Build-Plugins den Vertrag mittels Testaufrufen gegen den Provider prüfen können (Skurrie, 2015).

CDC-Testing ist teamübergreifend trotzdem mit Vorsicht zu behandeln und erfordert eine zuverlässige Entwicklerkultur. Werkzeuge wie Pact erlauben es, auch Anforderungen an den Provider zu stellen, welche dieser gar nicht in seinem Provider-Contract spezifiziert hat. Innerhalb eines Teams sind solche Missverständnisse schnell geklärt. teamübergreifend können daraus Kommunikationschaos und Schuldzuweisungen entstehen.

3.1.1.2 Tolerant-Reader Pattern

Das Tolerant-Reader Pattern ist ein Service-Design Pattern und kann in einer Microservice-Architektur nur in Form von Entwickler-Guidelines eingesetzt werden. Wendet ein Microservices dieses Pattern an, darf er bei eingehenden Nachrichten keine harten Annahmen über die Anzahl von Datenfeldern oder der genauen Position eines Datenfeldes machen. Er muss in der Lage sein, aus einer Nachricht die für ihn interessanten Daten zu extrahieren und andere zu ignorieren. Das erlaubt es anderen Microservices, ihre Schnittstelle abwärtskompatibel zu erweitern, indem neue

Funktionen zur bestehenden Nachrichtenstruktur hinzugefügt werden, ohne alte Datenfelder zu löschen (Daigneau, 2011).

3.1.1.3 Expand-Contract Pattern

Testen löst zwar das Problem der Erkennung von Fehlern in der Schnittstelle, aber nicht die Kopplung, die eine Schnittstelle zwischen Services erzeugt. Die Unabhängigkeit von Deployment-Einheiten kann aber durch dieses Pattern zum Teil erhalten bleiben. Die Kernidee besteht darin, während einer Migrationsphase die neue Version der Schnittstelle parallel zu der alten Schnittstelle zu betreiben und nach und nach die Consumer der alten Schnittstelle auf die neue Schnittstelle zu migrieren. Das kann prinzipiell auf mehrere Arten geschehen (Newman, 2015).

Beispiel: Semantic Versioning kann das Pattern in Form von versionierten Endpunkten realisieren. Die Versionierung erfolgt auf dreistufiger Basis, wobei Nummern nach folgendem Format inkrementiert werden (Major.Minor.Patch):

- **Major** gibt an, dass nicht-abwärtskompatible Änderungen an der Schnittstelle durchgeführt wurden.
- **Minor** gibt an, dass die Funktion abwärtskompatibel erweitert wurde.
- **Patch** gibt an, dass kleinere Fehler verbessert wurden.

Wenn das Tolerant-Reader Pattern richtig angewendet wird, können Minor und Patch für die Migration der Schnittstelle vernachlässigt werden. Muss ein Service-Endpunkt nicht abwärtskompatibel erweitert werden, kann die neue Funktion auf einem höher versionierten Endpunkt parallel betrieben werden, bis alle Consumer ihre Anfragen an den neuen Endpunkt richten. Besteht die Microservice-Architektur aus sehr vielen Services, liegt die Schwierigkeit darin, zu erkennen, ab wann alle Consumer den neuen Endpunkt nutzen. Hierbei können auch Consumer-Driven-Contracts unterstützend wirken.

Beispiel: Microservices mit unterschiedlichen Versionen zu betreiben ist ein anderer Ansatz, um das Pattern zu realisieren. Hierbei werden nicht die Endpunkte versioniert, sondern parallel eine zweite Instanz des Microservice mit der neuen Funktionalität betrieben.

3.2 Organized around Business Capabilities

Die Struktur der entwickelnden Organisation hat laut Conway's Law (vgl. Abschnitt 2.7.2) einen großen Einfluss auf die Struktur des Softwaresystems.
Die klassische Einteilung anhand funktionaler Abgrenzungen in Spezialisten für Benutzerschnittstellen, Entwickler für Geschäftslogik und Datenbankadministratoren, hat bei Änderungen oft eine teamübergreifende Koordination nötig. Sollen beispielsweise mehr Details auf der Benutzerschnittstelle angezeigt werden, so muss das Datenmodel der Datenbank angepasst oder erweitert werden. Zusätzlich muss die Geschäftslogik dafür sorgen, dass diese Details verarbeitet und im richtigen Format an Datenbank und Benutzerschnittstelle weitergegeben werden. Erst anschließend kann die Informationen auf der Benutzerschnittstelle

präsentiert werden. Alle Änderungen müssen zeitlich koordiniert erfolgen, da sie aufeinander aufbauen. Dieser Ablauf kann in starren Unternehmen oft unverhältnismäßig viel Zeit in Anspruch nehmen, weil die Kommunikation zwischen den Abteilungen gestört ist oder gar nur über einen zentralen Architekten geschehen darf.
Ein zusätzlich negativer Nebeneffekt ist die Tendenz der einzelnen Entwickler dazu, den unkomplizierten Weg zu suchen. Die funktionalen Abteilungen wie zum Beispiel Benutzerschnittstellen- und Datenbankteam werden meist versuchen, möglichst viel eigene Logik für kleinere Änderungen implementieren, um den Kommunikationsaufwand gering zu halten und so die saubere Abtrennung der Geschäftslogik aushebeln, weil in der Folge Logik in allen Schichten zu finden ist und weitere Änderungen immer schwieriger werden.
Abschließend haben die Abteilungen untereinander oft wenig Verständnis für die Anforderungen des Anderen und wenig Überblick über den fachlichen Kontext, in den ihre Änderung eingebettet ist. Die eigentlich für den Nutzer bestimmte Funktion leidet oft darunter, weil sie schlussendlich umständlich konzipiert oder missverständlich ist.

Im Kontext Microservices wird eine Aufteilung der Entwicklung um fachliche Geschäftseinheiten herum angestrebt. Jedes Team ist im Rahmen seines abgegrenzten Kontext eigenständig für Benutzerschnittstelle, Geschäftslogik und Datenbank inklusive Datenschema verantwortlich. Änderungen können so unkompliziert innerhalb eines kleinen Teams, im besten Fall persönlich unter vier bis sechs Augen, besprochen werden.
Ein zusätzlicher Vorteil ist der bleibende fachliche Bezug des Teams. Entwickler verbleiben im fachlichen Kontext und bauen über die Zeit ein immer besseres Verständnis über Anforderungen und Möglichkeiten des entsprechenden Fachbereichs auf. Das steigende kontextuelle Know-How ermöglicht es den Entwicklern, präziser an den Schwachstellen ihres Dienstes zu arbeiten. Das kann dazu beitragen, eine stetige Verbesserung der Software für den konkreten Anwendungszweck zu erreichen.

3.2.1 Herausforderung: Transfer von funktionellem Know-How

Know-How für funktionelle Bereiche kann schlecht zwischen den Teams geteilt werden, da die Spezialisten nicht mehr als funktionelles Team arbeiten, sondern auf unterschiedliche Teams verteilt sind. Viele Teams kämpfen eventuell mit ähnlichen Problemen in der Geschäftslogik. Ohne einen geschickten Wissenstransfer über Teamgrenzen hinweg werden Probleme unnötigerweise öfter als einmal gelöst.

Agile Organisation bei Spotify Ein Ansatz, diese Schwächen der fachlich orientierten Organisation auszumerzen, ist die agile Organisation bei dem Musik-Streaming Dienst Spotify. Er setzt ebenfalls auf unabhängige Teams, die aber zusätzlich in eine mehrschichtige flexible Matrixorganisation eingebettet sind. Bei Spotify gibt es vier wichtige Organisations-Entitäten.

Squads Die fachlichen Teams bei Spotify heißen *Squads*. Sie sind eigene kleine Start-Ups im Unternehmen und größtenteils autonom mit eigenem Product-Owner. Ausgestattet mit einer langfristigen Mission, sollen diese Teams selber an ihrem fachlichen Bereich arbeiten und über A/B-Testing und Metriken diesen optimieren. Der Slogan für jeden Squad lautet: *„Think it, build it, ship it, tweak it“*.

Tribes In höherer Ebene sind Squads nach thematisch passenden Bereichen zu *Tribes* zusammengefasst. Typischerweise sind alle Squads innerhalb eines Tribes in räumlicher Nähe platziert und kleiner als 100 Personen. Sie halten regelmäßige informelle Treffen ab, in denen jeder Squad vorstellt, an was er gerade arbeitet und wie er bestimmte Probleme gelöst hat.

Chapters and Guiles Die funktionell auseinander gerissenen Spezialisten vereinigt Spotify in *Chaptern* und *Guilden*.
Chapter existieren innerhalb eines Tribes und vereinigt diejenigen Mitarbeiter der verschiedenen Squads, welche ähnliche Kompetenzen und Aufgabengebiete haben oder in ähnlichen Problembereichen arbeiten. So gibt es beispielsweise ein Test-Chapter, Web-Entwicklungs-Chapter und ein Backend-Chapter. Sie treffen sich regelmäßig und diskutieren Herausforderungen und neue Technologien.
Guilden erweitern Chapter über Tribes hinaus. Sie stellen vielmehr Interessengruppierungen dar. Jeder der sich für bestimmte Themen interessiert, kann diesen Guilden beitreten und Problemlösungen teilen oder über Technologien diskutieren (Kniberg & Ivarsson, 2012).

3.3 Products not Projects

Bei der klassischen funktionalen Aufteilung der Entwicklung ist es üblich, projektbasiert vorzugehen. Abbildung 3.1 zeigt die Unterteilung der einzelnen Projektschritte und unternimmt eine Zuordnung in die zuständigen Abteilungen. Für Änderungen oder neue Features wird ein Projektteam aus den funktionalen Abteilungen zusammengewürfelt, welches dann für Analyse, Planung und Umsetzung der Änderung verantwortlich ist und seine fertige Änderung an die Qualitätssicherung übergibt. Ist die Softwarequalität zufriedenstellend, wird das Projekt als abgeschlossen angesehen. Das gewonnene Know-How wird im besten Fall in Abschlussberichten versteckt zu den Akten gelegt und anschließend das Projektteam aufgelöst, weil die neue Version erfolgreich an den Betrieb übergeben wurde. Der Betrieb ist anschließend dafür verantwortlich, dass die Software problemlos im Betrieb läuft.

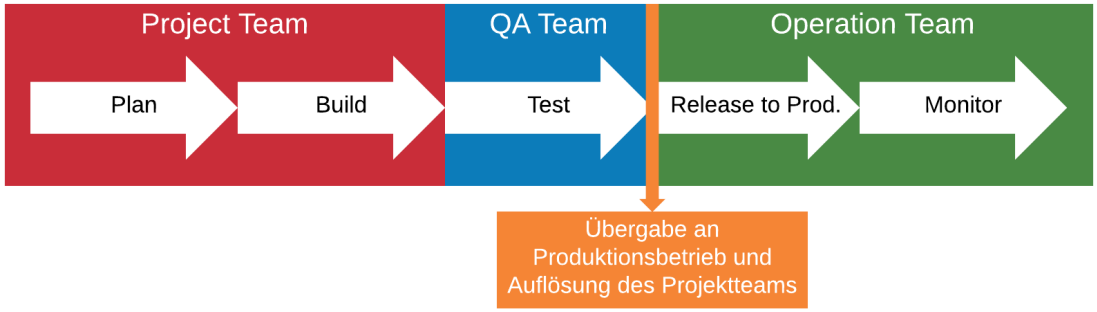

Abbildung 3.1: Typischer Ablauf von projektbasiertem Vorgehen in der Softwareentwicklung.

Genau dieses Szenario soll in Rahmen von Microservices vermieden werden, weshalb ein Microservice eher als ein zu pflegendes Produkt gesehen werden muss. Ist der Geschäftsbereich

durch gute fachliche Kontextgrenzen sauber eingeteilt, sollen die meisten Änderungen und Features nur innerhalb eines Teams von statten gehen. Das erlaubt es, Änderungen und neue Features nicht als Projekt, sondern als kontinuierliche Verbesserungen an dem Produkt Microservice zu sehen. Jedes Produkt wird von einem stabilen Team betreut. Erweiterungen, Wartung und Betrieb wird in der Regel von dem gleichen Personenkreis in Eigenverantwortung durchgeführt. Diese Praxis stammt aus dem DevOps-Bereich (vgl. Abschnitt 2.7.5) und kann positive Auswirkungen auf die Beziehung von Entwicklern zu ihrer Software haben. Entwickler kommen durch die Aufgaben beim Betrieb täglich in Kontakt mit ihrem Code und können schneller reagieren, falls doch Fehler auftreten. Da die Entwickler im Bewusstsein haben, dass sie für den reibungslosen Betrieb der Software verantwortlich sind, kann hierdurch auch die Codequalität verbessert werden.

Der Kernpunkt von *Products not Projects* ist, dass ein Microservice kein einmaliges Projekt ist, das irgendwann abgeschlossen ist und dann jahrelang vom Betrieb verfügbar gehalten wird. Vielmehr ist ein Microservice ständigem Wandel des betreuenden Teams unterzogen und soll idealerweise durch Anwendung von Continuous-Delivery (vgl. Abschnitt 2.7.4) täglich verbessert und erweitert werden.

3.3.1 Beispiel Amazon

Eine radikale Umsetzung dieses Prinzipes kommt von Amazon. Dort gilt die Order *„You built it, you run it!“* und macht die Entwickler rund um die Uhr direkt für ihr eigenes Produkt verantwortlich. Bei Problemen im Betrieb gibt es keine verantwortliche Operations-Abteilung, sondern die verantwortlichen Entwickler werden direkt per Pager alarmiert.

Ein typisches Team bei Amazon ist zwischen 6-10 Leute stark und wird an verschiedenen Geschäftsmetriken gemessen. Das Team an sich bleibt in seinen weiteren Entscheidungen relativ autonom und versucht die Geschäftsmetriken seines konkreten Service-Produkts zu maximieren. Ganz wichtig ist dabei der *Customer-Feedback-Loop* in Abbildung 3.2.

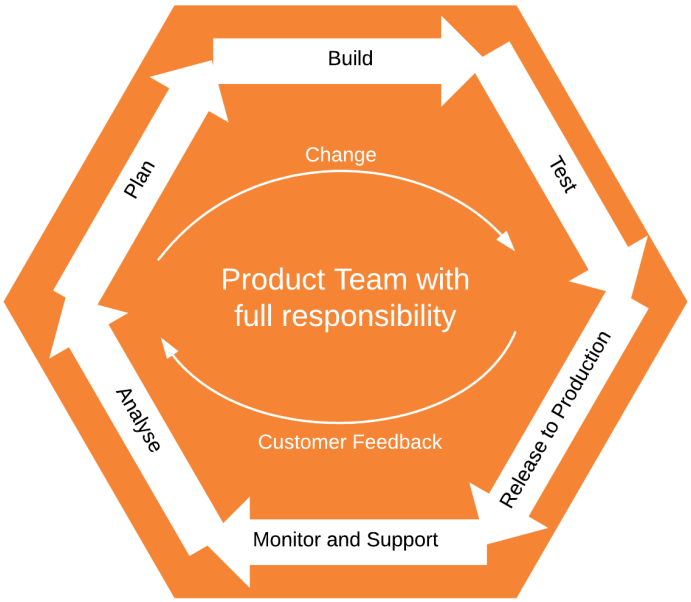

Abbildung 3.2: Typischer Zyklus von produktorientierten Teams in der Softwareentwicklung.

Im Gegensatz zu Abbildung 3.1 sind die einzelnen Schritte nicht auf unterschiedliche Einheiten verteilt und es gibt

auch keine Übergabe an den Betrieb. Die Entwickler stellen die Qualität ihrer Software selber sicher und entscheiden, welche Versionen in Produktion betrieben werden. Ganz entscheidend ist, dass jedes Team selber für das Monitoring der Anwendung zuständig ist und so ein immer besseres Gefühl für Schwachstellen und Kernkomponenten ihres Produktes enthält. Durch veränderte Werte der Metriken im Monitoring erhalten die Entwickler sehr schnell Rückmeldung, welche Auswirkungen ihre Änderungen hatten und können angemessen reagieren.

Für Amazon bedeutet das absolute Kundenorientierung, da das Team erster Ansprechpartner für unternehmensinterne und externe Kunden ist und anschließend selbständig reagieren kann, ohne die Angelegenheit über mehrere Instanzen an die verantwortlichen Personen zu delegieren.

3.4 Smart Endpoints and Dumb Pipes

Diese Charakteristik zielt auf die Kommunikationstechnologie der Services ab. In diesem Zusammenhang wird oft von schwergewichtigen und leichtgewichtigen Methoden für Service-Kommunikation geredet, wobei sehr schwer zu definieren ist was genau schwer- und was leichtgewichtig sein soll.

Schwergewichtig Fowler und Lewis gehen hierbei konkret auf den Enterprise Service Bus (ESB) ein, welcher im Zusammenhang mit Microservices als verpönt gilt. Schwergewichtig ist an diesem Ansatz, Intelligenz und Validierung in der Kommunikationstechnologie zu verstecken. Es können Regeln für Geschäftslogik, Datentransformation und eine Orchestrierung von einzelnen Services an zentraler Stelle definiert werden. Dadurch besitzt ein ESB komplexe Mechanismen mit denen die Funktion des Gesamtsystems entscheidend beeinflusst werden kann und widersprechen so den abgegrenzten fachlichen Verantwortlichkeiten zwischen den Teams, welche jeweils für eine konkrete Teilfunktion des Systems verantwortlich sind.

Leichtgewichtig Demnach können leichtgewichtige Technologien als Gegenpart gesehen werden. Die eingesetzten Technologien sollen möglichst dezentralisiert arbeiten, sodass jede Verantwortlichkeit in den Teams verbleiben kann. Konsequenterweise wird Logik wie Datentransformation, Security-Enforcement und Orchestrierung nicht in die Kommunikationstechnologie verpackt, sondern muss von jedem Team eigenverantwortlich in seinem Service implementiert werden. Dieser Ansatz orientiert sich am Erfolg des Internets, welches sich als dezentrales System mit relativ einfachen Kommunikationsprotokollen aber intelligenten Netzknoten als flexibel und robust erwiesen hat.

Im Microservices Stil wird auf jegliche Dezentralisierung wert gelegt, um den Teams ein möglichst unabhängiges Arbeiten zu ermöglichen. Dabei wird auf leichtgewichtige Technologien gesetzt und auch in Kauf genommen, dass dadurch ein höherer Aufwand für die Schnittstellenentwicklung und das Management von Sicherheitsfunktionen betrieben werden muss. Dazu sind intelligente Schnittstellen notwendig. Diese sollen nach außen so effektiv wie möglich nutzbar

sein, aber gleichzeitig soll jede Anfrage möglichst skeptisch behandelt werden. Jedes Team muss selbst unternehmensinterne Microservices von anderen Teams als potentiell unberechenbar einstufen.
Grundsätzlich orientieren sich Microservices an der Webtechnologie, die sich als Basis des Internets bewährt hat. Cacheing, HTTP-Operationen mit semantischer Bedeutung und Fehlercodes sind somit automatisch gegeben. Im Falle von asynchroner Kommunikation sind ganz einfache Messageing-Systeme ohne große Sonderfunktionalität die präferierte Wahl.

3.4.1 Herausforderung Autorisierung

Dezentralisierung der Services hat den Nachteil, dass die Autorisierung und Authentifizierung mit höherem Aufwand verbunden ist. Jeder Service muss einzeln in der Lage sein, zu beurteilen, ob eine Anfrage berechtigt ist oder ob diese abgelehnt werden muss.
In einem auf Komponenten basierten Monolith wird dieses Problem über ein Session- oder Requestobjekt gelöst, bei welchem zu Beginn der Anfrage Authentifizierung und Autorisierung durchgeführt wird und die entsprechenden Informationen mittels diesem Objekt an alle Komponenten weitergereicht wird.
In zentralisierten SOA-Systemen kann ein ESB als Sicherheits-Manager fungieren und die entsprechenden Vorgänge erzwingen.

Für Microservices gibt es verschiedene Ansätze, um Authentifizierung und Autorisierung für Anfragen durchzuführen.

OAuth2.0 und OpenID Connect Eine Möglichkeit, jeden Microservice in die Lage zu versetzen, selber zu entscheiden, ob Anfragen berechtigt sind, besteht in der Delegierung von Autorisierung und Authentifizierung.
Für die Integration verschiedenster Webdienste und APIs hat sich dabei OAuth2.0 und OpenId durchgesetzt. Microservices stellen im weitesten Sinn auch unterschiedliche Webdienste dar, die in irgendeiner Form integriert werden, weshalb diese Technik auch in diesem Kontext verwendet werden kann. Sie beruht darauf, die Autorisierung und Authentifizierung an einen Identity-Provider Service (IdP) zu delegieren und versetzt jeden Microservice über Tokens in die Lage, diese zu verifizieren.

OAuth2.0 kennt hier vier Kern-Rollen für den Protokollablauf bei der Autorisierung von Clients (Hardt, 2012):

- **Client:** Der Client ist die anfragende Entität. Das kann beispielsweise eine Web-Anwendung oder eine mobile App sein, die Informationen von einem Microservices-Backend abrufen möchte. Je nach interner Architektur könnte aber auch ein Microservice die Rolle des Clients einnehmen.
- **Resource-Owner:** Der Owner ist die Entität, die in der Lage ist, den Zugriff zu erlauben. Im Falle einer menschlichen Person ist diese mit dem eigentlichen Benutzer gleichzusetzen.
- **Authorization-Server:** Ist die Instanz, welche die eigentliche Authentifizierung des Resource-Owners durchführt und ein gültiges Token an den Client versendet, falls der Owner den Zugriff autorisiert. Kann im Zusammenhang mit Microservices ein eigener IdP-Service sein oder von einem externen Dienstleister bereitgestellt werden.

- **Resource-Server:** Ist die angefragte Entität, welche letztendlich die geschützten Ressourcen verwaltetet und an den Client versenden kann, beispielsweise ein Microservice.

Um den einzelnen Resource-Servern nicht nur Referenzen auf gültige Autorisierungen bereitzustellen, gibt es mit *OpenId Connect* einen Standard für Identity-Provider. Damit lassen sich ganze Identitäten in signierte Tokens kodieren, welche dann den Resource-Servern zur Verfügung stehen.
Dabei werden die Informationen der Identität in JSON Web Tokens (JWT) verpackt. Die Informationen sind dabei encodiert und mit einem private Key des IdP-Service signiert. Über einen public Key, den jeder Ressource-Server besitzt, kann dieser dann die Echtheit überprüfen.

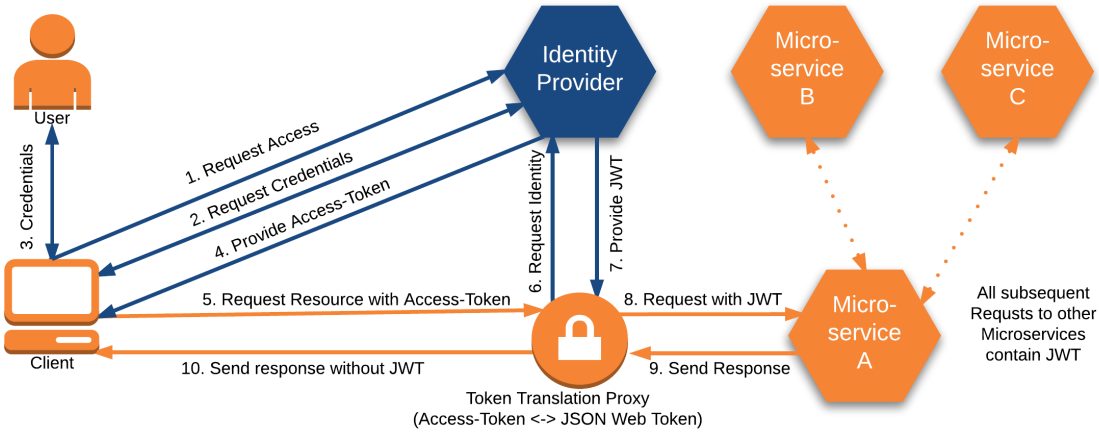

Abbildung 3.3: Beispielablauf einer Autorisierung von Requests einer Client-Anwendung an ein Microservice-System.

Abbildung 3.3 zeigt einen vereinfachten Ablauf der Autorisierung von Clientrequests an ein Microservice System mit OAuth2.0 und JWT.

1. Die Clientanwendung fragt nach Zugang zum System.
2. Der IdP fragt nach Credentials einer gültigen Identität, um ein entsprechendes Access-Token auszustellen.
3. Der User stellt korrekte Credentials bereit, welche die Clientanwendung dem IdP übergibt.
4. Der IdP stellt der Clientanwendung ein Access-Token für den Zugriff auf das System aus. Umfang der Rechte, Identität und Länge des Zugangs verbleibt im IdP versteckt und gelangt vorerst nicht nach außen.
5. Die Clientanwendung stellt eigentlichen Request an das Microservice-System.
6. Der Token-Translation Proxy nimmt Access-Token und überprüft es auf Gültigkeit und fragt gleichzeitig Informationen über die Identität und Umfang des Zugriffsrechts ab.
7. Der IdP stellt dem Proxy die Informationen in Form eines signierten JWTs bereit.
8. Der Proxy leitet eigentlichen Request inklusive JWT an den Microservice weiter, welcher so alle Informationen über die anfragende Identität enthält und das JWT ebenfalls mit einem public-Key auf Gültigkeit überprüfen kann.

9. Der Microservice sendet dieses JWT bei allen nachfolgenden Anfragen an andere Microservices im System, von denen er zusätzliche Informationen einfordert, mit und generiert schließlich die Response an den Client.
10. Proxy entfernt das JWT aus dem Header und leitet Response an die externe Clientanwendung weiter.

Der Token Translation Proxy kann in Form eines Gateways zum Microservice-System realisiert werden und stellt sicher, dass die Identitäten nur innerhalb des Backend-Systems aus Microservices benutzt werden und nicht in den öffentlichen Internetverkehr gelangen.
Der Identity-Provider ist ein eigener Microservice, welcher die Benutzeridentitäten abspeichert und Rechte und Rollen verwalten kann. Hierbei können auch externe IdP wie Google, Facebook oder Twitter verwendet werden. Es ist zu beachten, dass die Anfragen 1-4 nur nötig sind, wenn die Clientanwendung kein Access-Token besitzt oder dieses abgelaufen ist. Die Anfragen 5-6 können durch Caching am Proxy ebenfalls eingespart werden.

3.5 Decentralized Governance

Ein großes Problem monolithischer Software besteht darin, dass zu Beginn der Entwicklung die Entscheidung für die verwendete Technologie fällt und anschließend zementiert ist. Alle Komponenten sind an diese Technologie gebunden, obwohl für einige Teilaufgaben eine andere Plattform die bessere Wahl wäre. Die Wartung muss auch nach Jahren an einem möglicherweise völlig veralteten Technologiestack vorgenommen werden, an dem neue Programmierkonzepte und flexiblere Programmierung nicht möglich ist. Der verwendete Technologiestack wird nach aufwändigen Machbarkeitsanalysen von zentraler Stelle vorgegeben und die Entwickler sind dann in der Pflicht, sich damit abzugeben.

Microservices haben durch ihre Eigenständigkeit die Möglichkeit, auf unterschiedlichen Plattformen und Technologien zu basieren. Oft wird eine dezentrale Führung favorisiert, wobei den jeweiligen Teams viel Autonomie zugesprochen wird. Sie haben eine relativ freie Wahl in ihrem Technologiestack und müssen lediglich die Anforderungen ihrer Schnittstelle erfüllen. So können Teams die beste Technologie für ihre spezifischen Probleme nutzen.
Zusätzlich sind Teams in der Lage, selbständig neue und moderne Technologie auszuprobieren und die Services, für die sie verantwortlich sind, innerhalb weniger Tage portieren.

3.5.1 Beispiel Zalando: Radical Agility

Bei Zalando wurde das Top-Down Management zugunsten von selbstbestimmten Teams aufgegeben. Vorherige Führungskräfte fungieren jetzt als Mentoren und kümmern sich um den Aufbau neuer Teams und Produkte. Wenige Product Owner geben die grobe Richtung für Teams vor, stellen aber keinesfalls konkrete Zielvorgaben. Die drei Grundpfeiler von Radical Agility sind (Apple, 2015b):

- **Zweck:** Jegliche Entwicklung muss irgendeinen Zweck verfolgen. Jedes Team benötigt einen tieferen Sinn beziehungsweise eine Mission.

- **Eigenständigkeit:** Zu viel Bestimmung hindert Innovation. Teams müssen die Möglichkeit haben, selbst Erfahrungen zu machen um Innovationen zu kreieren. Sie bestimmen selbst, wann und wie ein Feature entwickelt wird.
- **Verbessern:** Jeder Mitarbeiter hat die Möglichkeit, sich in seine präferierte Richtung weiterzuentwickeln. Mitarbeiter dürfen ab und an Teams wechseln und können so mit verschiedenen Technologien experimentieren.

3.6 Decentralized Data Management

Datenbank und Datenschema sind oft kritische Punkte für die Flexibilität in der Software-Entwicklung. Klassische Systeme nutzen historisch bedingt große relationale Datenbanksysteme, die als zentrale Punkte im System viele andere Anwendungen integrieren. Prinzipiell sind auch mit Microservices Zugriffe auf geteilte Datenbanken möglich und Konzepte wie SOA haben in der Vergangenheit verteilte Anwendungsentwicklung mit zentralen Datenbanken praktiziert.
Der große Nachteil dabei ist die schlechte horizontale Skalierbarkeit von einzelnen großen relationalen Datenbanken. Viel gravierender ist aber, dass das zentrale Datenschema für eine horizontale Skalierung der Entwicklung anhand fachlicher Teams sehr hinderlich ist. Bei jeder Änderung eines Datenfeldes muss überprüft werden, welche anderen Microservices von der Änderung betroffen sind und anschließend eine Kompromisslösung gefunden werden.

Eine effektive Umsetzung von verteilter Entwicklung in fachlichen Teams verlangt eine Aufteilung der Datenhoheit. Jedes Team soll innerhalb seines Bounded Context (vgl. Abschnitt 2.7.3.1) sein eigenes Datenmodell entwickeln und Zugriffe darauf über die Schnittstellen der Microservices kapseln. So können bei Änderungen unabhängig von anderen Teams Datenfelder verändert werden, was im Optimalfall zu einem sauberen Datenschema mit möglichst wenigen veralteten Attributen führt.
Gleichzeitig sind die Teams frei in der Wahl der Datenbanktechnologie. In der Tutinderanwendung kann es beispielsweise sinnvoll sein, für die Freundschaften und das Rating statt einer herkömmlichen relationalen Datenbank eine auf Graphen basierte Datenbanktechnologie zu wählen, welche besser für die Abbildung von Beziehungen geeignet ist. Für den Chat sind vielleicht sehr einfache aber schnelle No-SQL Datenbanken von Vorteil.
Durch die Unabhängigkeit der Daten zwischen den Microservices können die einzelnen Datenbanksysteme besser skaliert werden. Zusätzlich ergibt sich dadurch ein robusteres System (vgl. Abschnitt 2.7.6). Ist beispielsweise bei großer Last auf dem *Rating-Service* dessen Datenbank überlastet, so können die Gruppen weiterhin problemlos chatten, da der Chat-Service und seine Datenbank davon nicht betroffen sind.

3.6.1 Herausforderung: Daten-Konsistenz

ACID - Atomic, Consistent, Isolated, Durable Große relationale Datenbanksysteme sind zwar schlecht im Zusammenhang mit flexibler Softwareentwicklung mit verschiedenen Teams, haben aber den großen Vorteil, dass sie die Möglichkeiten bieten, Änderungen in Form von Transaktionen durchzuführen und erlauben es, auf streng konsistenten Daten zu arbeiten. Durch die Aufteilung der Daten in mehrere Datenbanken geht diese Möglichkeit verloren.

Die von relationalen Datenbanksystemen gewohnte Konsistenzart ist *ACID*:

- **Atomic:** Alles oder nichts, entweder werden alle Änderungen angenommen oder der Zustand wird nicht verändert.
- **Consistent:** Eine Transaktion kann in keinem inkonsistenten Zustand enden.
- **Isolated:** Durch Sperren wird sichergestellt, dass sich mehrere Transaktionen nicht gegenseitig beeinflussen.
- **Durable:** Daten, die eine abgeschlossene Transaktion hinterlässt, sind auch dauerhaft persistiert.

In Umgebungen, in denen die Daten verteilt sind, müssten verteilte Transaktionen eingesetzt werden. Diese Transaktionen beruhen auf einem zentralen Manager wie beispielsweise einem Message-Broker. Verteilte Transaktionen ermöglichen ACID über einen *Two-Phase-Commit* Verfahren. Hierbei werden alle Änderungen vom Manager koordiniert angefragt und nur durchgeführt, wenn alle Teilnehmer bestätigen. In der Theorie sind diese Transaktionen zuverlässig und können für kleinere Systeme problemlos eingesetzt werden.
Sobald die Systeme größer werden und die Zahl der Anfragen steigt, stellt sich ACID grundsätzlich als Flaschenhals für eine effektive Skalierung heraus. Gesperrte Ressourcen führen zu Konkurrenzsituationen, in denen immer eine Transaktion blockiert. Zusätzlich sind Transaktionen im verteilten System sehr anfällig, da bei Verbindungsproblemen zu einem Teilnehmer der Transaktion alle anderen Aktionen abgebrochen werden.

Für Systeme, die Wert auf eine hohe Verfügbarkeit legen, sind verteilte Transaktionen und deren ACID-Konsistenz wegen des CAP-Theorems grundsätzlich ungeeignet, da mit Forderung nach Konsistenz die Verfügbarkeit leidet. Gerade bei hoch skalierten verteilten Systemen wird es aber immer irgendwo Verbindungsprobleme geben und Netzwerkanfragen fehlschlagen, was zu ständig blockierten Ressourcen und abgebrochenen Transaktionen führt.

BASE - Basically Available, Soft State, Eventual Consistency BASE ist ein anderer Ansatz der Konsistenz. Im Gegensatz zu ACID ist es optimistisch ausgerichtet und erlaubt innerhalb des Systems einen zeitlich begrenzten inkonsistenten Zustand. Für die meisten Anwendungen ist nachträgliche Konsistenz unkritisch. Typischerweise sind auch gängige Geschäftsabläufe nicht konsistent und erfordern gelegentlich Kompensationen.
Nachträgliche Konsistenz ist besonders im Zusammenhang mit ereignisbasierter Kommunikation interessant. Anders als bei Request-Response basierter Kommunikation gibt es keine Informationen und Möglichkeiten, den Prozessablauf innerhalb eines Services zu kontrollieren. Beispielsweise sind in der Tutinderanwendung mehrere Bounded Contexts auf den Namen eines Users angewiesen. Ändert der Benutzer seinen Namen im User-Management Service, befindet sich das System in einem inkonsistenten Zustand. Es wird nötig, bei jeder Änderung des Namensattributes ein globales Ereignis zu veröffentlichen, auf das andere Services reagieren können, wenn sie das Namens-Attribut verwenden. Das System befindet sich schließlich nur solange im inkonsistenten Zustand, bis alle Services auf das Ereignis reagiert haben und den Namen entsprechend in ihren replizierten Daten geändert haben.

Ist Daten-Konsistenz in einem bestimmten Bereich zwingend erforderlich, beispielsweise bei Finanzangelegenheiten, dann kann strenge Konsistenz auf diesen Bereich beschränkt erreicht werden, indem der entsprechende Bounded Context größer gewählt wird. Entsprechend würden

dann zwei Microservices innerhalb dieses Bounded Context zusammen eine relationale Datenbank nutzen, in welcher strenge Konsistenz mittels Transaktionen erreicht werden kann.

3.7 Infrastructure Automation

Da Microservices per Definition kleine eigenständige Deployment-Einheiten sind, steigt dadurch natürlich der Aufwand beim Schaffen einer Test- und Produktionsinfrastruktur. Nicht selten haben große Unternehmen über hundert einzelne Services im Einsatz, die oft tausendfach skaliert werden.
Ein manuelles Einrichten der einzelnen Umgebungen wäre sehr aufwändig und würde viel zu lange dauern, sodass eine flexible Skalierbarkeit nicht gegeben ist.

In diesem Zusammenhang setzen Microservices sehr stark auf Continuous-Delivery (vgl. Abschnitt 2.7.4). Das automatisierte Durchlaufen der einzelnen Stufen der Delivery Pipeline ist nur möglich, wenn die Infrastruktur flexibel und automatisiert ist, was auch eine starke Umsetzung von DevOps (vgl. Abschnitt 2.7.5) erfordert.
Techniken wie Virtualisierung oder das Packen der Software in Containern mit definierter Konfiguration sind unerlässlich, um Microservices effizient entwickeln und betreiben zu können.

Eine Infrastruktur mit gut funktionierendem Provisioning macht ein Softwaresystem extrem flexibel in der Wartung. Da Microservices in der Regel um ein Vielfaches schneller deployt werden können als monolithische Systeme, ist das System in der Lage, bei steigender Last sehr schnell neue Instanzen hinzuzufügen.
Von Vorteil für ein hoch verfügbares System ist auch die Möglichkeit eine neue Version schrittweise über Blue/Green Deployment einzuführen und so eine unterbrechungsfreie Wartung zu ermöglichen.

Der Betrieb von Microservices in einer vollautomatisierten Umgebung ist effizienter, einfacher und risikoärmer. Jedoch ist die Hürde zu dieser Umgebung hoch. Das Prinzip von Continuous Delivery klingt einfach, eine Umsetzung kann sehr schwierig und komplex werden. Speziell wenn verschiedene Software unterschiedliche Konfigurationen der Infrastruktur benötigen.

3.7.1 Beispiel Netflix: Spinnaker

Ein Tool, das dabei helfen kann einen Deploymentprozess zu erschaffen, der auf viele Microservices und Cloudplattformen abgestimmt ist, heißt Spinnaker. Es stammt aus dem Open-Source-Software Bereich von Netflix und kooperiert mit den meisten großen Anbietern von IaaS und PaaS, darunter AWS, CloudFoundry, Google und zukünftig auch Microsoft Azure. Spinnaker bezeichnet sich selber als *Continuous Delivery Plattform für die Cloud* und schafft es, Artefakte wie beispielsweise Packages als Input zu nehmen und ein fertig deploytes Image als Output zu hinterlassen.
Spinnaker unterteilt hierzu in 3 Deployment Einheiten:

- **server-group:** Ist im Prinzip ein Microservice mit einer konkreten Version. Enthält die Basis-Konfiguration der jeweiligen Hosts, sowie Security und Load-Balancing Konfigurationen.
- **cluster:** Logische Gruppierung mehrere Server-Groups. Zum Beispiel mehrere Microservices mit unterschiedlichen Versionen.
- **application:** Logische Gruppierung der Cluster zu der endgültigen Anwendung.

Spinnaker setzt nach Continuous Integration an, wo mit Hilfe von Build-Servern wie *Jenkins* ein fertig kompiliertes Software-Package vorliegt. Alle Vorgänge werden als Spinnaker-Workflow angelegt und bestehen aus *Stages*, wobei verschiedene Stages zu *Pipelines* kombiniert werden, welche wiederum hierarchisch als Stage dienen können.

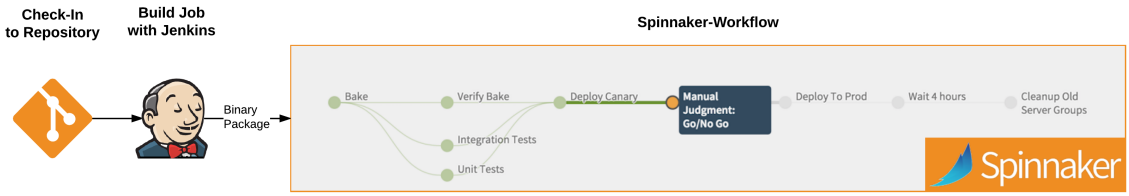

Abbildung 3.4: Beispielvorgang eines Deployments mit Jenkins und Spinnaker.

Beim beispielhaften Deployment-Vorgang in Abbildung 3.4 wird Spinnaker im ersten *Bake* Stage aus dem Binären-Paket ein fixiertes Image mit den eingestellten Konfigurationen basteln. Je nach Wunsch kann man daraus beispielsweise ein Amazon-Machine-Image oder einen Docker-Container erhalten. Anschließend werden verschiedene Tests durchgeführt und anschließend ein Canary-Release durchgeführt. Das heißt, die neue Version wird parallel zur alten Version unter Teil-Last betrieben. Nach der manuellen Entscheidung, dass die neue Version einwandfrei läuft werden 100% der Anfragen auf die neue Version geleitet und die alte Version für eine gewisse Zeit in Reserve gehalten, bevor sie aufgeräumt wird.

Weitere Formen von Stages sind:

- **Destroy Server Group:** Entfernt eine vorhandene Server-Gruppe.
- **Disable Server Group:** Deaktiviert eine laufende Server-Gruppe.
- **Enable Server Group:** Aktiviert eine vorhandene Server-Gruppe.
- **Find Image:** Sucht nach früher erstellten Images für ein weiteres Deployment.
- **Jenkins:** Führt beliebige weitere Jenkins-Prozesse aus.
- **Manual Judgment:** Wartet bis eine manuelle Benutzerentscheidung getroffen wird.
- **Pipeline:** Führt eine vorher aus Stages definierte Pipeline aus, erlaubt komponieren verschiedener Pipelines zu einem Workflow.
- **Resize Server Group:** Verändert die voreingestellte Größe der Server-Gruppe.
- **Script:** Kann beliebige eigene Skripte ausführen und erlaubt somit volle Freiheit.

3.8 Design for Failure

Eine Microservice-Architektur ist naturgemäß eine verteilte Architektur und unterliegt so den besonderen Bedingungen beim Betrieb einer solchen. Mit einer Wahrscheinlichkeit von fast 100% wird es irgendwann zu Kommunikationsfehlern kommen, auf die das System angemessen reagieren muss. Im Gegensatz zu Funktionsaufrufen innerhalb eines Prozesses kann bei Microservices jeder Aufruf über das Netzwerk fehlschlagen und muss daher für jeden spezifischen Fall einzeln behandelt werden.
Die Kommunikation der Microservices muss so gestaltet sein, dass ein Service sinnvoll weiterarbeiten kann, wenn ein anderer aufgerufener Service gerade nicht zur Verfügung steht. Andernfalls könnte ein ausfallender Service eine Kette auslösen und so das gesamte System lahmlegen. Das kann zum Beispiel durch das Annehmen von Defaultwerten oder Verwenden von gecachten Einträgen passieren. In der Tutinder-Anwendung könnte der Rating-Service für die Berechnung der Reihenfolge von angezeigten Kursteilnehmern Detailinformationen von einem Service abrufen. Ist dieser nicht verfügbar wäre es schlecht, wenn dadurch der Rating-Service seine Funktionalität komplett einstellt und dem Nutzer gar kein Rating ermöglicht. Vielmehr sollte im Sinne von *Graceful-Degradation* eine Berechnung ohne Detailinformationen erfolgen.

3.8.1 Beispiel Netflix

Wenn es um das stabile Betreiben von verteilten Microservice-Systemen geht, erfreuen sich viele an den Open-Source Werkzeugen von Netflix.

Hystrix-Turbine Ganz bekannt ist dabei das Framework *Hystrix* und der Aggregator *Turbine*. Mit Hystrix lassen sich zwei Patterns im Zusammenhang mit Stabilität sehr einfach umsetzen:

- **Timeout-Pattern:** Hystrix kapselt alle Aufrufe eines Microservice an andere Systeme in Form von Commands, die in einem eigenen Thread ausgeführt werden. So lassen sich eigene Zeitgrenzen für externe Aufrufe festlegen, nach denen das System entscheidet, dass ein Aufruf fehlgeschlagen ist. Ohne Timeouts kann ein einziger Microservice dafür sorgen, dass nach und nach alle anderen Systeme ausfallen, weil diese wartend blockieren, aber niemals mehr eine Antwort erhalten.
- **Circuit-Breaker-Pattern:** Dieses Pattern ist wichtig, um eine Überlastung anderer Services zu vermeiden beziehungsweise ihnen Luft zu verschaffen, sich nach Überlast wieder zu erholen.

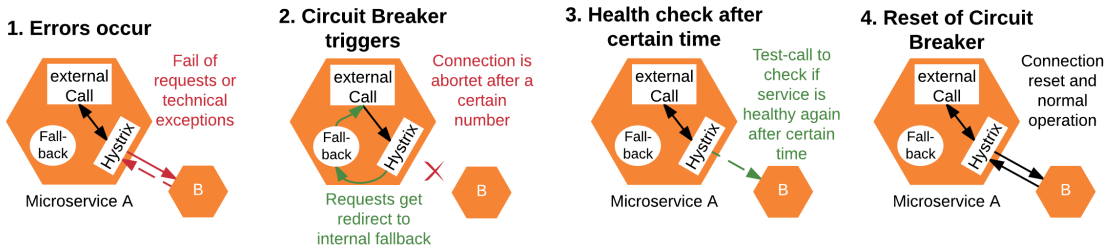

Abbildung 3.5: Vier Schritte des Circuit-Breaker Patterns. In Anlehnung an (Newman, 2015).

Abbildung 3.5 zeigt den typischen Ablauf bei der Anwendung dieses Patterns. Ein *Microservice A* benötigt von *Microservice B* Informationen und macht dazu externe Aufrufe. Ist B überlastet oder hat interne Probleme, wird Hystrix den Aufruf nach einem Timeout oder einer Exception erneut versuchen, die Anzahl der Versuche kann konfiguriert werden. Ab einer bestimmten Anzahl wird Hystrix zum Schutz von B alle weiteren Aufrufe von A an B blockieren und auf interne Fallback-Methoden umleiten. Erst nach einer definierten Zeit wird A einen Test-Aufruf an B senden und anhand von Metriken, zum Beispiel der Latenz, seine Gesundheit verifizieren und anschließend wieder in den Normalbetrieb übergehen.
Der Kernpunkt besteht darin, dass andere Services bei Problemen sofort den eigenen Anteil der Last am Gegenüber reduzieren und ihm somit die Chance geben, eventuelle Warteschlangen abzuarbeiten oder sich wieder in einen funktionierenden Zustand zu begeben. Würden alle Services einfach weiterhin Anfragen auf den überlasteten Service schicken, hätte dieser keine Chance, sich zu regenerieren und zusätzlich wird unnötige Last im Netzwerk erzeugt. Gleichzeitig wären alle weiteren Anfragen von A an B auch verzögert, da immer erst Timeouts abgewartet werden müssten. So kann A, im Bewusstsein über den Zustand von B, ein langwieriges Fehlschlagen gleichartige Anfragen von Nutzern verhindern, indem es direkt einen passenden Fallback anwendet, was zum Beispiel gecachte Daten sein könnten. Sinnvollerweise implementiert ein Großteil der betriebenen Microservices dieses Pattern.

Die Informationen über offene und geschlossene Circuit-Breaker lassen sich auch sehr gut als Metrik für das Health-Monitoring der gesamten Microservice-Anwendung nehmen. Netflix bietet dafür direkt eine Anwendung namens *Hystrix Dashboard*, diese nimmt die Log-Informationen, die das Hystrix System als Json-Strom herausschreibt, und visualisiert diese.
Weil es kaum sinnvoll ist für eine Vielzahl von Microservices jeweils ein eigenes Dashboard zu betreiben und einzeln zu überwachen, gibt es zusätzlich noch einen Aggregator für diese Log-Informationen. *Turbine* sammelt die Hystrix-Logs von allen angeschlossenen Microservices und stellt diese übersichtlich im Dashboard dar (vgl. Abbildung 3.6) (Wolff, 2016).

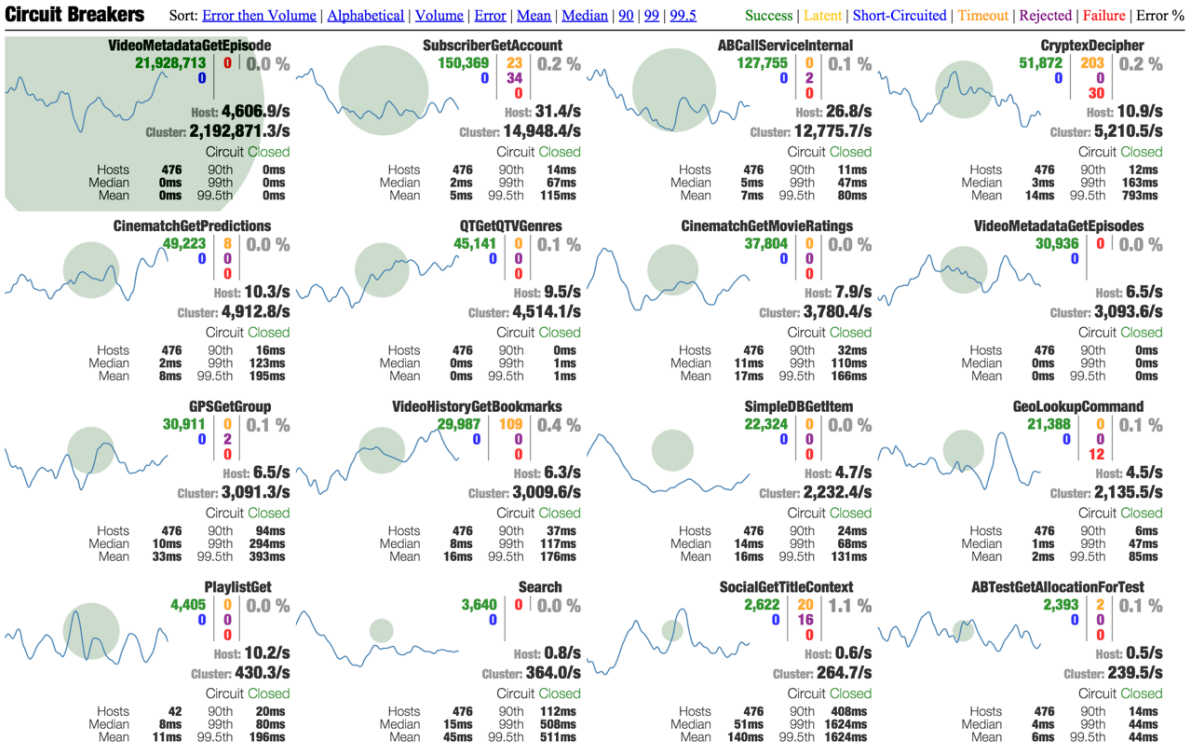

Abbildung 3.6: Screenshot des Hystrix-Dashboard. Übersichtsdarstellung mit den von Turbine aggregierten Informationen am Beispiel einiger Netflix-Services (Jacobs, 2016).

3.9 Evolutionary Design

Eine monolithische Architektur fokussiert typischerweise auf lange Sicht, Architekten und Entwickler versuchen deshalb Anforderungen für viele Jahre voraus abzuschätzen. Das liegt vor allem daran, dass Änderungen über mehrere Abteilungen hinweg koordiniert werden müssen und oft gemeinsam in festen Release-Zyklen in Produktion gebracht werden. Selbst kleine Änderungen oder Fehlerkorrekturen werden so wochenlang blockiert, bis die gesamte Software in einer neuen Version freigegeben wird.

Microservices verfolgen einen unabhängigeren Ansatz, in dem die Planung auf kürzere Zeiträume beschränkt ist. Oft werden in diesem Zusammenhang konkrete Missionen als visionäres Ziel an Teams ausgegeben. Die konkrete Umsetzung ergibt sich aber durch viele kleine Schritte, welche wenig Planung erfordern. So können aufwändige Planänderungen durch geänderte Anforderungen oft vermieden werden.

Der evolutionäre Part an Microservices wird auch durch die *„Try it out“*-Auffassung unterstrichen. Neue Ideen werden ohne große Planung einfach ausprobiert und eventuell als neuer eigener Service realisiert. Ist das Ergebnis nicht zufriedenstellend, hat das *„Mini-Produkt“* vielleicht wenige Wochen in Anspruch genommen. Mit einer monolithischen Architektur sind diese Versuche nicht so flexibel möglich, wodurch die Evolution der Software gehemmt ist. Neue Ideen werden

oft nicht umgesetzt, weil sie nicht mehr zu der in Stein gegossenen Struktur passen oder werden in ein um Monate späteres Release verschoben, weil eine komplexe Planung erforderlich ist.

4

Motivation praktischer Umsetzungen

Microservices sind ein Stil, welcher vielmehr aus der Praxis kommt als aus wissenschaftlichen Analysen. Unternehmen wie Amazon und Netflix gelten als Vorreiter in diesem Kontext und viele Ideen, die heute als zentrale Punkte gelten, sind aus Problemlösungen solcher Unternehmen entstanden.
Dieses Kapitel versucht, die konkrete Motivation für den Einsatz eines Microservice-Stils zu erarbeiten. Anhand von veröffentlichten Informationen dieser Unternehmen wird beschrieben, welche Probleme Unternehmen mit früheren Architekturen hatten und wohin die Umsetzung von einzelnen Aspekten des Microservice-Stils geführt haben.

4.1 Amazon

Speziell Amazon hat mit seiner heutigen Plattform *Amazon Web Services* eine sehr flexible Architektur geschaffen. Amazon hat sich selber vom Online-Shop für Bücher zu einer Plattform für Onlinehandel und inzwischen auch zu einem Cloud-Anbieter gewandelt. Amazon selbst hat seine eigene Organisation und Softwarearchitektur auf dem Weg zu dieser Plattform komplett umgestellt, um technologisch mit dem schnellen Wachstum mithalten zu können.

4.1.1 Grundlage

Die ehemalige Anwendung des damaligen Webshops bestand aus einer monolithischen Anwendung auf einem Webserver und mehreren großen Datenbanken.

4.1.2 Ziele

Amazon verfolgt das Prinzip, *„das Unternehmen mit der weltweit besten Kundenorientierung zu sein“* und gleichzeitig die Interaktion mit den Kunden kontinuierlich zu verbessern. Dafür ist es nötig, dass die Anwendungen flexibel fortentwickelt werden können. Viel wichtiger ist aber, dass die Entwickler direkt für die Interaktion mit den Kunden verantwortlich sind und keine weitere Instanz, die sich um den Betrieb kümmert, dazwischenliegt.

4.1.3 Probleme

Viele komplexe Softwarekomponenten waren innerhalb einer Anwendung vereint, die sich nicht mehr entfalten konnte. Einzelne Teile konnten nicht skaliert werden, weil sie über geteilte Res-

sourcen Abhängigkeiten enthielten. Da die Softwarekomponenten nicht voneinander isoliert waren, gab es unter den Entwicklern auch keine klaren Zuständigkeiten. Durch die Größe der Anwendung dauerten Builds zu lange und mussten über viele Teams hinweg koordiniert werden.

> „The many things that you would like to see happening in a good software environment couldn't be done anymore."
>
> —Werner Vogels, CTO Amazon

Gleichzeitig musste sich Amazon mit mehreren großen Datenbanken auseinandersetzen, die von den Komponenten als geteilte Ressourcen genutzt wurden.
Für Amazon war es hinderlich, dass viele Ressourcen über viele Teams und sogar Prozessabläufe hinweg verteilt genutzt wurden. Die große Erkenntnis bei Amazon war, dass Skalierung das größte Problem ist, dem das Unternehmen ausgesetzt ist.

4.1.4 Lösung

Amazon begann, alles voneinander zu isolieren was keinen fachlichen Zusammenhang hatte. Die bisherige Anwendung wurde in kleine Ausführungseinheiten *„Services"* zerteilt. Daten und Logik, die zusammengehören, wurden zusammengefasst, so dass jeder Service seine eigene Datenbank hatte. Ein direkter Zugriff auf die Datenbank eines anderen Service war ausdrücklich verboten, stattdessen sollte jeder Service eine Schnittstelle zum Zugriff auf die Daten bereitstellen. Werner Vogels beschreibt den vollzogenen Vorgang folgendermaßen (Vogels, 2011):

> „We take this App-Server that we had, look for each piece of unique business logic, take that business logic, look at what's the data that actually that business logic operates on, bring those things together, slap an API around it and that's what we call a Service. No direct database access allowed anymore. "

Amazon verfolgt heute eine strenge DevOps-Philosophie. Das Unternehmen ist in Teams aufgeteilt, die voll verantwortlich für ihre Services sind. Neue Ideen, Weiterentwicklung, Fehlerbehebung und Betrieb obliegen der Verantwortung des Teams nach dem Prinzip *„You build it, you run it "*. Amazon sieht darin den großen Vorteil, dass Entwickler durch den Betrieb ihres Service direkt in Kontakt mit den Kunden ihrer Schnittstelle kommen und so ein besseres Gefühl für die Qualität ihres Service erhalten (O'Hanlon, 2006). Die gesamte Infrastruktur ist automatisiert und in Form von *Infrastructure as Code* direkt für die Entwickler verfügbar. Das Deployment ist durch Continuous Delivery und einer automatisierten Deployment Pipeline beschleunigt worden, sodass das Unternehmen auf insgesamt 50 Millionen erfolgreiche Deployments von Softwareänderungen im Jahr kommt.

4.2 Netflix Service Architecture

Netflix ist ein Video-On-Demand Service der Serien und Filme über Online-Streaming anbietet. Der Dienst gilt als Pionier in Themen rund um Microservices und dem Betreiben von Anwendungen in der Cloud.
Netflix begann schon 2009 mit dem Umbauen seiner monolithischen Architektur, als der Begriff einer Microservice-Architektur so noch nicht existierte, weshalb das Unternehmen es als *fine-*

grained SOA bezeichnete. Der Fokus lag dabei auf dem Entwickeln einer passenden Architektur, damit Vorteile der Cloud ausgenutzt und Risiken minimiert werden konnten.
Der Schritt in die Cloud war für Netflix ein logischer Schluss zugunsten der Geschwindigkeit bei Produktentwicklung und Betriebssicherheit.

4.2.1 Grundlage

Die ursprüngliche Architektur von Netflix bestand aus mehreren Bladeservern, auf denen horizontal skaliert eine monolithische Java-Anwendung betrieben wurde. Diese Anwendung griff hauptsächlich auf ein großes Oracle-Datenbanksystem auf einem hochperformanten Rechnersystem zu. Nach einer großen Störung der eigenen vertikal skalierten Datenbank musste Netflix die Anfälligkeit seiner Anwendung reduzieren und überdachte seine Architektur.

4.2.2 Probleme

Die großen relationalen Datenbanken oder die monolithische Backend Anwendung stellten sich als Single-point-of-Failure dar, da sie in keinem anderen Rechenzentrum repliziert waren. Zusätzlich hatte man mit über 100 Entwicklern schon Probleme beim Integrieren von Codeänderungen in die monolithische Anwendung und erwartete gleichzeitig einen Anstieg auf ungefähr 2000 Entwickler in den nächsten Jahren, weshalb die monolithische Anwendung sowieso aufgeteilt werden musste. Zuletzt war in der bisherigen Architektur sehr viel Business Logik zwischen den Webservern und Datenbanken über PL/SQL aufgeteilt, was zu sehr viel Komplexität im Umgang mit der Wartung führte.

4.2.3 Lösung

Netflix wagte den Schritt in die Cloud und benötigte dafür eine neue Architektur. Nach und nach wurden alle Netflix Anwendungen auf ein verteiltes Cloudsystem umgebaut, wobei dessen Teile zuverlässig und einfach horizontal skaliert werden können. Die neue Architektur sollte folgende übergeordnete Ziele erreichen (Amazon Web Services, 2014):

- **Verfügbarkeit:** Wird erreicht durch Redundanz der Anwendung über mehrere AWS-Zonen mit jeweils mehreren Instanzen. Bei einem Ausfall eines Teils der AWS-Cloud kann der Traffic auf andere Zonen umgeleitet werden. Zusätzlich nutzt Netflix eigene Tools wie *Hystrix*, die einen Circuit-Breaker implementieren. Dadurch soll verhindert werden, dass Fehler oder Ausfälle eines Microservices nicht die gesamte Anwendung durch kaskadierte Wartezeiten kollabieren lässt. Netflix setzt hierbei sehr stark auf *Graceful Degradation*[1], wobei zum Beispiel statt nutzerspezifischen Vorschlägen nur randomisierte Vorschläge erscheinen, der Rest des Systems aber fehlerfrei funktioniert.
- **Skalierbarkeit:** Wird erreicht durch das Entkoppeln der monolithischen Anwendung und der Aufsplittung der relationalen Datenbank. Netflix nutzt ein Cluster aus Memcache und Cassandra Systemen, die beliebig skaliert werden können. Da Netflix einen Dienst betreibt, der sehr starken Lastschwankungen ausgesetzt sein kann, ist zusätzlich eine hohe

[1] Bezeichnet die Art wie ein System mit Fehlersituationen umgeht. Fehler sollen nicht zum Komplettausfall eines Systems führen, sondern nur die fehlerhafte Funktion so weit wie möglich kompensiert oder abgeschaltet wird.

Elastizität notwendig. IaaS-Plattformen sind durch die schnelle Bereitstellung schon sehr stark darauf ausgelegt, trotzdem hält Netflix einige Instanzen als Reserve vor. Diese führen während Ruhezeiten Aufgaben wie Pre-Processing von Daten für Vorschläge durch und können sehr schnell recyclt werden.

- **Performanz:** Wird über strenge *Service-Level-Agreements* für jede Ebene geregelt. Nicht die durchschnittliche Performanz wird gemessen, sondern die Werte in den kritischen Phasen sind entscheidend. Alle Komponenten im System müssen diese Werte erreichen, da sich schlechte Performance auf andere Komponenten kaskadiert.
- **Agilität:** Wird hauptsächlich durch das Aufsplitten der monolithischen Anwendung in Microservices erreicht. Teams können ihre Änderungen nun unabhängig von anderen Durchführen. Bisher hatten Teams eigenständig Komponenten entwickelt, welche in die monolithische Codebasis integriert werden mussten und sich an einen strikten Release-Plan hielten, der im besten Fall Releases alle zwei Wochen vorsah. Jetzt kann jedes Team seinen eigenen Plan aufstellen oder direkt Continuous Delivery anwenden, um noch schneller Feedback über Änderungen zu bekommen.
  Zusätzlich erlaubt das Nutzen von Infrastructure as a Service (IaaS) den Aufbau einer Weltweiten Infrastruktur, obwohl alle Entwickler in Kalifornien sitzen. Es können Testläufe und Experimente in verschiedenen Regionen durchgeführt werden, ohne dort eigene Rechenzentren aufzubauen.
- **Sicherheit:** Der Schritt in die Cloud ermöglicht Netflix einen neuen Fokus in Bezug auf die Sicherheit. Die Standardaufgaben wie beispielsweise Sicherheitsupdates für die Infrastruktur, Management der Zugangsidentitäten für Rechenzentren oder Backups werden vom Cloud-Provider übernommen. Netflix kann sich so den wichtigen Themen widmen. Beispielsweise der Verschlüsselung der Daten und der Kommunikation zwischen den Microservices.
- **Skalierung:** Jede einzelne Komponente soll ohne Architektur-bedingtes Limit unendlich skalierbar sein. Hierzu müssen alle Komponenten parallel hinter einem Load-Balancer betrieben werden können.

4.2.4 Lessons learned

Für Netflix war die Migration von eigenen Rechenzentren hin zu einem Betrieb auf IaaS ein langwieriger Lernprozess, aus dem sich drei Kernpunkte schließen lassen (TiE SiliconValley, 2014).

- **Cloud-Native Architecture:** Für einen reibungslosen und effektiven Betrieb in der Cloud muss die Architektur auf deren Eigenschaften abgestimmt sein. Sie muss Vorteile wie Elastizität in der Skalierung und Flexibilität in der Bereitstellung voll ausschöpfen und gleichzeitig Tolerant gegenüber Fehlern sein. Bei Netflix hat sich über die Jahre eine Microservice-Architektur als passend herausgestellt.
- **Cloud-Native Operations:** Der Betrieb muss der Cloud angepasst werden. DevOps unterstützt das Erstellen von Tools für den automatisierten Betrieb der Cloud-Instanzen und führt neue Konzepte in Logging und Monitoring von virtuellen Hosts ein.

- **Cloud-Native Organization:** Eine veränderte Kultur mit mehr Freiheiten und klaren Missionen kann den Betrieb in der Cloud unterstützen. Budget für Ressourcen muss nicht zentral verwaltet werden, sondern kann direkt den verantwortlichen Teams überlassen werden, welche sich selbständig Instanzen für ihre Services erstellen, um damit schnelle Releasezyklen zu erreichen.

4.3 Zalando

Zalando ist ein 2008 gegründeter deutscher Onlineversandhändler, der seit Gründung ein sehr starkes Wachstum zu verzeichnen hat.

4.3.1 Grundlage

Angefangen hatte der Händler mit einem Shop basierend auf der Standardlösung von Magento und hatte wegen starkem Wachstum bald Probleme, das große Lastaufkommen zu bewältigen. Da jegliche Optimierungsversuche scheiterten, wurde eine Eigenlösung in Java und PostgreSQL geschaffen. Der Fokus lag wegen den vorherigen Problemen vor allem auf Effizienz, Geschwindigkeit und Stabilität der Anwendung. Dafür wurde ein defensiver Ansatz gewählt und den Webentwicklern Zugriff auf die Datenbank nur über Stored-Procedures gewährt.

4.3.2 Probleme

Nach einigen Jahren weiteren Wachstums, in denen viele neue Features eingeführt wurden, war die Wartung immer schwieriger geworden. Die hauptsächlichen negativen Effekte waren:

- **Produktivität:** Immer mehr Entwickler arbeiteten an derselben Codebasis, was immer mehr Koordination zur Folge hatte und letztendlich dazu führte, dass das Hinzufügen von Entwicklern keinen positiven Effekt auf die Entwicklungszeit hatte. Die Entwicklung war nicht mehr angemessen skalierbar.
- **Innovation:** Mit steigender Größe der Codebasis reduzierte sich die Innovation. Das lag vor allem daran, dass sich mit zunehmender Größe die Fehlerdichte erhöhte und Nebeneffekte von Änderungen nicht mehr überschaut werden können. So musste der ganze Veröffentlichungsprozess sehr strikt gehalten werden, was die Freiheit für Innovation einschränkte.
- **Personal:** Die Eigenentwicklung basierte mit Spring und JSP auf einem bewährten Technologiestack, der mit der Zeit aber veraltet und wenig ansprechend für neue Entwickler ist. Das Unternehmen hat so Schwierigkeiten, die besten und vor allem junge Entwickler einzustellen, die zu der modernen Firmenkultur passen. Zusätzlich dauerte es sehr lange, neue Entwickler in die große Codebasis einzuarbeiten, da diese einen sehr großen Umfang verstehen müssen.

4.3.3 Ziele

Das Unternehmen sieht sich selbst vielmehr als Technologiekonzern. Es soll eine Transformation, weg vom reinen Onlineshop und hin zu einer ganzen Online-Plattform für viele andere Händler, stattfinden. Dafür hat Zalando die in Abschnitt 3.5.1 behandelte Unternehmenskultur namens *Radical Agility* eingeführt (Apple, 2015b).

4.3.4 Lösung

Wichtig für die Umsetzung der freiheitlichen Kultur war eine passende Software-Architektur. Zalando setzt hierbei auf einen API-First Ansatz, der konkret ausschließlich auf REST aufbaut. Die Logik wird in Form von Microservices implementiert, wobei jeder Microservice von genau einem Team verwaltet wird. Teams sind zwischen 8 und 12 Mitarbeiter stark und können auch mehrere Services verwalten. Ganz wichtig ist, dass alle Services mit einem Bewusstsein für Software as a Service entwickelt werden und daher *Multitenant*[2] sind.

Die Infrastruktur ist sehr cloudorientiert und setzt auf der AWS-Plattform auf. Zusätzlich gibt es einen sehr starken Drang, schon bestehende SaaS Angebote zu nutzen, um eigenen Entwicklungsaufwand zu reduzieren. So werden beispielsweise Bewerbungssysteme oder Monitoring von externen Expertensystemen übernommen, welche meist ebenfalls über eine REST-Schnittstelle angesprochen werden (Apple, 2015a).

4.4 Soundcloud's Weg zu Microservices

Soundcloud bezeichnet sich selbst als *social sound Plattform*, auf der Nutzer Sounds hochladen und veröffentlichen können. Andere Nutzer können diese abrufen, kommentieren und mit Freunden teilen. Die Plattform bietet Zugang über Mobile Anwendungen, eine Webanwendung und eine öffentliche HTTP-Schnittstelle. Obwohl das Unternehmen Millionen Nutzer bedient, gilt es immer noch als Start-Up und hat mit einer monolithischen Anwendung begonnen.

4.4.1 Grundlage

Phil Calçado, ehemaliger Director of Engineering der Kernanwendung beschreibt die Hintergründe über den Architekturwandel der Soundcloud Plattform aus Produktivitätssicht.
Die Entwicklung bei Soundcloud war in Backend- und Webteams getrennt, die hauptsächlich über Tickets und Chat kommunizierten. Der eigentliche Entwicklungsprozess liest sich ganz einfach:

> Eine neue Idee wurde von jemandem kurz schriftlich und in Form von Mockups niedergeschrieben, worauf Designer und Entwickler ihre Aufgaben erledigen und nach kurzem Testen wird die Funktion in Produktion gebracht.

[2] Unterstützt mehrere unabhängige Kunden. Jeder Kunde besitzt seine eigenen Daten, Konfigurationen und Zugang. Kundendaten sind voneinander abgeschottet. Beispielsweise sollen mehrere Teams einen Deployment-Microservice unabhängig voneinander nutzen können

4.4.2 Probleme

In Wirklichkeit benötigten Features aber über einen Monat, um in Produktion zu gelangen, was für das junge Unternehmen nicht schnell genug war. Es gab zwei Kernprobleme:

- Features wurden zwischen Webteam und Backend-Team hin- und hergeschoben, wobei zusätzliche Wartezeiten für ein Feature entstand, weil sich niemand verantwortlich fühlte und kein Team das Ticket bearbeitete.
- Das Backend-Team entfremdete sich von dem eigentlichen Produkt und der Nutzererfahrung, da sie mehrheitlich auf Anforderungen des Webteams reagieren müssen und selbst keine Entscheidungskraft mehr besaßen.

4.4.3 Ziele

Die Ziele und Anforderungen an den Umbau der Architektur waren zusammengefasst:

- Features sollen isoliert entwickelt werden können, ohne dabei andere Komponenten anfassen zu müssen.
- Mit hoher Wahrscheinlichkeit sollten Änderungen an einer Komponente keine Fehler oder Verhaltensänderungen in nicht betroffenen Teilen des Systems verursachen.
- Features sollten isoliert von anderen Features deployt werden können.
- Sollte ein Deployment eines Features einen Fehler im Betrieb verursachen, darf nur das geänderte Feature betroffen sein.

4.4.4 Erster Lösungsansatz

Als Lösung dafür, wurden Feature-Teams bestehend aus Designern, Produktmanagern, Frontend-Entwicklern und Backend-Entwicklern eingeführt, die sich um das Umsetzen neuer Features von Idee bis Produktionsbetrieb kümmern.

Das Abwandern vieler Entwickler aus dem Backend-Team in die Feature-Teams erschwerte die Kommunikation und Abstimmung bei der Entwicklung an der Software aufgrund der jetzt räumlichen Trennung. Informelle Kommunikationskanäle, die vorher bestanden, gab es nicht mehr, weshalb es zu Missverständnissen bei der Codeintegration kam.
Soundcloud nutzte eine vereinfachte Form von Continuous Integration, wobei alle Änderungen im Hauptbranch integriert wurden. Es mussten verstärkte Code-Reviews in Form von doppelt gecheckten Pull-Requests eingeführt werden, bevor eine Codeänderung im Hauptbranch integriert wurde, um die Fehlerquote niedrig zu halten. Diese Reviews zögerten den Entwicklungsprozess um Tage hinaus, da die monolithische Kernanwendung so komplex geworden war. Große zu überprüfende Pakete wurden von Entwicklern gemieden und die Features verzögerten sich zusätzlich.

4.4.5 Zwischenanalyse

Die Analyse warf grundsätzliche Fragen auf:

- 1. Warum sind Pull-Requests überhaupt nötig?
  A: Die Erfahrung zeigt, dass Menschen Fehler machen, diese unerkannt in Produktion bringen und somit die ganze Plattform ausfällt.
- 2. Warum machen Menschen Fehler?
  A: Eine große Codebasis ist komplex, alles und jede Auswirkung zu überblicken ist schwer. Kein Entwickler kennt die gesamte Anwendung und Experten für ein Modul sind nicht immer verfügbar.
- 3. Warum ist die Codebasis so komplex?
  A: Weil Soundcloud aus einer kleinen Webseite zu einer großen Plattform mit vielen verschiedenen Features, Clientanwendungen, synchronen und asynchronen Arbeitsabläufen geworden ist. Die Codebasis ist ein Spiegelbild dieser komplexen Plattform.
- 4. Warum ist eine einzelne Codebasis für all diese Komponenten nötig?
  A: Aus Effizienzgründen. Der Monolith hatte an für sich ein gutes Deployment mit viel Werkzeugunterstützung und daher eine robuste Architektur gegenüber Spitzenlast und DDOS-Attacken, da er noch relativ einfach horizontal zu skalieren war. Würde man das System aufsplitten, müssten diese Merkmale für jedes System extra erreicht werden, wobei der wirtschaftliche Verbundeffekt nicht mehr zum Tragen käme.
- 5. Warum können wir keine Skaleneffekte von vielen, kleinen Systemen erreichen?
  A: Diese Frage war schwierig zu beantworten und führte dazu, dass die eigene Plattform analysiert wurde.

Die Plattform bestand vereinfacht gesehen aus den in Abbildung 4.1 abgebildeten Komponenten ohne klare Zuständigkeiten, aber mit jeweils eigenen Evolutionszyklen. Das *Subscriptions*-Modul wird nur wenig geändert, im Gegensatz zum *Notification*-Modul oder anderen Modulen, die direkt mit der Kernfunktionalität der Plattform wachsen und täglich Änderungen erfahren. Soundcloud machte auch verschiedene Anforderungen an die Zuverlässigkeit aus. Fällt das *Notification*-Modul für eine Stunde aus, ist das nicht so schlimm. Ein Ausfall für wenige Minuten bei einem *Soundwiedergabe*-Modul ist aber sehr schädlich für das Image bei Kunden.
Die Skaleneffekte waren prinzipiell möglich, es gab jedoch Meinungen, die es favorisierten, die Verbundeffekte des Monolithen zu behalten und die Prozesse um ihn herum zu optimieren. Ein Ansatz war es, die impliziten Komponenten stärker zu trennen und damit explizit abzugrenzen, aber trotzdem in dem Monolithen zu belassen und über einen Load-Balancer die Anfragen funktional nach der Zuständigkeit der Komponenten zu verteilen. Diese Infrastruktur hatte seine eigenen Risiken und eine gewisse Komplexität. Die Umsetzung als iterativer Prozess hätte sich als schwierig erwiesen und der Monolith hätte sowieso überarbeitet werden müssen.

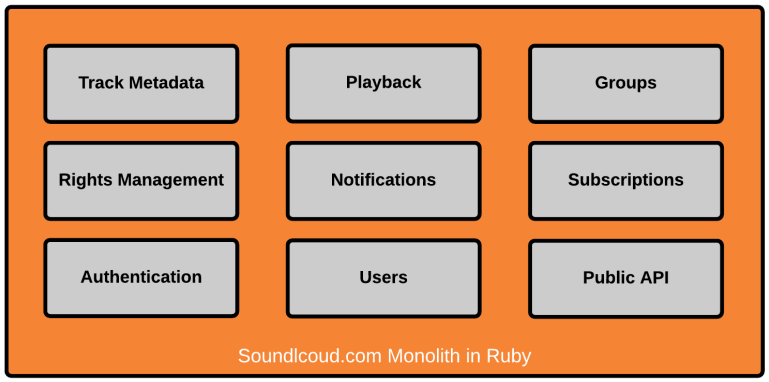

Abbildung 4.1: Interne Komponenten des Soundcloud-Monoliths. In Anlehnung an (Calçado, 2015).

4.4.6 Zweiter Lösungsansatz

Die Entwicklung neuer Features findet daher auf der grünen Wiese in Form von Microservices statt, um die sich spezielle Feature-Teams kümmern.
Backend-Entwickler, die keinem Feature-Team zugeteilt sind, kümmern sich um die immer noch vorhandene Kernapplikation. Weil es dort aber keine klaren Verantwortlichkeiten gab, wollte auch niemand das Risiko eingehen, die einzelnen Module in Microservices zu extrahieren, weshalb der blockierte Entwicklungszyklus am Monolith immer noch bestand. Die Entwickler wurden deshalb zufälligen Gruppen von 3-4 Leuten zugeteilt, die volle Verantwortung für die ihnen zugewiesenen Module haben und selbständig darüber entscheiden können. Den Teams obliegt auch die Entscheidung ob und wann das Modul aus dem Monolith als Microservice extrahiert wird. Bei Problemen im Betrieb müssen diese Gruppen auch für die Behebung sorgen. Erst durch die Zuteilung der Verantwortlichkeiten an den Kernmodulen ließ sich eine zunehmende Ausgliederung der Module aus dem Monolithen hin zu Microservices erkennen.
Bei Soundcloud befindet sich ein Teil der Funktionalität der Kernanwendung noch immer im Monolith und läuft dort stabil. Das Unternehmen sieht keinen Grund darin, diese Funktionalität unnötig zu extrahieren, solange dort keine Änderungen zu erwarten sind (Calçado, 2015).

4.5 Erkenntnis

An der Liste der Unternehmen, die als *„Early Adopter“* in Sachen Microservice gelten, kann man auch erkennen, dass ihre IT und Dienstleistungen endkundenorientiert sind. In der Enterprise-IT für Geschäftsprozessunterstützung sind Microservices noch nicht erkennbar angekommen.
Endkundenorientierte Unternehmen müssen eine enorme Anzahl von Serveranfragen bewältigen, gleichzeitig zuverlässigen Betrieb gewährleisten und regelmäßig neue Features auf ihren Plattformen bereitstellen, um gegen die Konkurrenz zu bestehen. So haben diese Unternehmen eigenständig passende Infrastrukturen, Tools und neue Organisationsformen eingeführt, welche den Microservice-Stil geprägt haben.

Die folgenden Abschnitte fassen verschiedene Motivationen für die Umsetzung des Microservices-Stil und dessen Zielarchitektur zusammen.

4.5.1 Skalierung

Die Hauptmotivation, aus der heraus Microservices entstanden sind, liegt in Problemen rund um die Skalierung. Hier kann in 3 Kategorien unterschieden werden:

- **Skalierung bezüglich Anwendungsdaten:** Herkömmliche Datenbanksysteme setzen auf das ACID-Paradigma und sind wegen der Einhaltung strenger Konsistenz schlecht horizontal skalierbar. Große Datenmengen und hohe Last machen es notwendig, Daten aufzuteilen.
- **Effektive Skalierung von Anwendungen:** Große Anwendungen können nur als gesamte Einheit horizontal skaliert werden. Moderne Cloudplattformen unterstützen aber sehr flexible Möglichkeiten, Systeme einzeln passend zu konfigurieren und der Last entsprechende Ressourcen zu nutzen. Einzelne kleine Anwendungen starten schneller und können je nach Anforderung mit mehr CPU oder RAM besetzt werden, was insgesamt eine elastischere und effektivere Skalierung ermöglicht.
- **Skalierung der Entwicklung:** Die Entwicklung an einer monolithischen Anwendung skaliert nicht gut. Gemeinsame Releasezyklen, interne Abhängigkeiten und lange Build- und Testlaufzeiten behindern die Produktivität der einzelnen Entwicklungsteams gegenseitig. Zusätzlich bedeuten große Anwendungen eine längere Einarbeitungszeit für neue Entwickler, da diese einen großen Umfang der Anwendung verstehen müssen.

4.5.2 Continuous Delivery und DevOps

Ebenfalls sehr wichtig ist die Geschwindigkeit, mit der Software weiterentwickelt werden kann. Die betrachteten Unternehmen beruhen sehr stark auf dem Vorsprung zur Konkurrenz und versuchen, diesen durch Anwendung von Continuous Delivery und DevOps auszubauen. Microservices erlauben es, durch kleinere Einheiten, bessere Abgrenzungen und klare Zuteilungen zwischen den Teams, den Release-Feedback Zyklus sehr schnell zu durchlaufen. Kleine Einheiten sind die Voraussetzung dafür, dass Continuous Delivery effizient umgesetzt werden kann und die Eigenständigkeit der Einheiten erlaubt es den Teams, selbst zu entscheiden, welche Features mit einem entsprechenden Reifegrad in Produktion gehen.

4.5.3 Innovationsfreundliche Unternehmenskultur

Der Microservice-Stil verkörpert eine sehr agile und flexible Unternehmenskultur. Die Unternehmen erhoffen sich von der freiheitlichen Entwicklerkultur ein besseres Image bei potentiellen Entwicklern und dadurch höhere Kompetenz bei der Rekrutierung von Mitarbeitern. Microservices erlaubt es diesen Mitarbeitern, möglichst unabhängig von anderen ihre Kompetenz einzubringen und dabei ihre eigenen gewohnten Werkzeuge einzusetzen.
Gleichzeitig soll die Hemmschwelle für innovative Änderungen herabgesetzt werden. In monolithischen Systemen gibt es wegen Abhängigkeiten oft Hemmungen, Änderungen durchzuführen,

welche für weitere Innovationen notwendig sind. Microservices sollen Entwickler wieder in die Lage versetzen, einen Überblick über kritische Änderungen zu bekommen und im Zweifel statt einer Änderung eine Neuentwicklung des ganzen Microservices umzusetzen.

4.5.4 Betriebsstabilität

Microservices können auch eine positive Auswirkung auf die Betriebsstabilität haben.
Die Unternehmen erhoffen sich durch die Unterteilung komplexer Anwendungen zu kleinen Teilen eine erhebliche Reduzierung der Fehleranfälligkeit. Besonders Fehler wegen undurchsichtigen Abhängigkeiten sollen dank der klaren Schnittstellen vermieden werden. Ein weiterer positiver Effekt ist die Verkleinerung der Fehlerdomäne auf den einzelnen Microservice. Unternehmen versprechen sich durch Microservices eine höhere subjektive Verfügbarkeit beim Benutzer, da im Fehlerfall eines Microservice der Rest des Systems trotzdem funktionieren sollte.

4.5.5 Cloud-Native

Der Microservices-Stil ist zwar nicht direkt Cloud abhängig, jedoch wurde er sehr stark von den Eigenschaften der Cloud geprägt. Besonders die Microservice-Architektur hat sich als sehr passend für große Systeme in Cloud-Umgebungen erwiesen, da sie es ermöglicht, die ganze Flexibilität der Cloud im Operations-Bereich auszunutzen. Des Weiteren kann ein Unternehmen dank der Verbindung von Microservices und der Cloud sehr einfach weltweit Experimente durchführen, indem zum Beispiel verschiedene Versionen von einzelnen Microservices betrieben werden, welche sich im Fehlerfall sehr schnell austauschen lassen.

5 Microservice-Stil und SOA

Microservices werden oft mit serviceorientierten Architekturen in Verbindung gebracht, da sich die Kernkonzepte ähneln. Viele Vergleiche haben aber nur wenig Aussagekraft, da weder Microservices noch SOA nicht konkret und allgemeingültig definiert sind. Oft ist unklar, auf welcher der Ebenen (Stil, Service, Architektur) eine Diskussion geführt wird. Für SOA gilt zusätzlich, dass das heutige Verständnis, was SOA eigentlich ist, oft über die ursprünglichen Definitionen hinausgeht.
Die Folge sind semantische Fehler im Vergleich. Beispielhaft sind hierfür:

- **Von Services auf Architekturen schließen:** Die eigentliche Intention hinter SOA ist ein reiner Architekturstil, der aus Konzepten besteht. Wie einzelne Services aussehen sollen und welchen Funktionsbereich ein Service abdecken, soll wird von SOA nicht spezifiziert.
  Der Begriff Microservice soll nur die konkrete Art der Services beschreiben. Zu dieser Verwirrung führt in erster Linie auch der missverständliche synonyme Gebrauch des Begriffs „Microservice“ in der Fachwelt.

  Sich einzelne Microservices anzuschauen und zu behaupten, dass die Art der Services genau gleich ist, wie wir sie in SOA auch schon gesehen haben, kann nicht dazu dienen, daraus zu schließen, dass Microservices und SOA dasselbe sind. Da SOA keine Implementierungsdetails der Services spezifiziert, sind einzelne Microservices prinzipiell auch in einer SOA zu finden, deswegen sind aber Microservice-Architekturen und SOA noch lange nicht identisch.
- **SOA ist nicht Webservices:** Zusammenhängend mit dem ersten Punkt ist die falsche Voraussetzung, dass SOA ausschließlich auf Webservices basiert. Webservices sind lediglich eine mögliche Form der Services, die in einer SOA eingesetzt werden können. Das Limitieren von SOA auf den Einsatz von Webservices mit seinen zusätzlichen Technologien ist zu einschränkend und widerspricht dem sehr generell gehaltenen Architekturparadigma.

5.1 Definitionen von SOA

Es werden im Folgenden verschiedene Definitionen von SOA dargestellt und zu einer Vergleichsbasis für Microservices vereint.

Thomas Erl definiert SOA über folgende serviceorientierte Grundprinzipien:

- **Standardized Service Contract:** Services innerhalb eines bestimmten Bereichs besitzen Schnittstellenspezifikationen nach demselben Standard.
- **Service Loose Coupling:** Services stellen geringe Anforderungen an eine Kopplung der Aufrufer und sind selbst unabhängig von ihren umgebenden Services.
- **Service Abstraction:** Schnittstellenspezifikationen enthalten nur relevante Informationen für den Aufruf und erlauben keine Annahmen über Implementierungsdetails.
- **Service Reusability:** Services sind abgeschlossene Einheiten, deren Funktionalität von anderen Services wiederverwendet werden kann.
- **Service Autonomy:** Services haben während der Ausführung ein hohes Maß an Kontrolle über darunterliegende Systeme oder Basis-Anwendungen.
- **Service Statelessness:** Der Service merkt sich keinen Zustand des Anfragekontexts und spart damit Ressourcen.
- **Service Discoverability:** Services verfügen über sprechende Metdaten, welche dabei helfen den Service zu finden und seine Funktion zu verstehen.
- **Service Composability:** Mehrere Services zusammen können zu einem Service größerer Funktionalität zusammengefasst werden.

Eine serviceorientierte Architektur bezeichnet demnach eine technologische Architektur, die dem speziellen Verhalten und den Anforderungen von Services entgegenkommt, damit diese die Ziele von *Service-oriented Computing*[1] realisieren können. SOA zeigt sich in Form von vier Charakteristiken (Erl et al., 2014):

- **Business-Driven:** Die Architektur orientiert sich an der geschäftlichen Organisation und kann sich im Einklang mit dieser verändern.
- **Vendor-Independent:** Die Architektur baut nicht auf einem Ökosystem eines speziellen Herstellers auf, sondern muss viele verschiedene Technologien zusammen unterstützen.
- **Enterprise-Centric:** Sie umfängt nicht nur einzelne Anwendungen, sondern einen wesentlichen Teil des Unternehmens.
- **Composition-Centric:** Kontinuierliche Veränderung soll durch wiederholtes Kombinieren von Services unterstützt werden.

Gartner beschreibt SOA 2003 als eine Software-Architektur. Sie besteht aus einer Topologie von Schnittstellen, den konkreten Schnittstellenimplementierungen und Aufrufen zwischen diesen. Services werden durch diese Schnittstellen beschrieben und sind groß genug, um komplette Geschäftsfunktionen abzubilden (Natis, 2003).

Papazoglu et al. bezeichnet SOA als logischen Weg, Software zu entwerfen, die Services über öffentliche und automatisch auffindbare Schnittstellen für Endnutzeranwendungen oder andere Services zur Verfügung stellt.
SOA bietet ein flexibles Geschäftsumfeld, indem es unabhängige und automatisch ausführbare

[1] Paradigma, das Services als Basis für die einfache und günstige Entwicklung von verteilten Anwendungen für Organisations- und Plattformübergreifende Umgebungen sieht. Geschäftsprozesse sollen so flexibel und dynamisch über ein Netzwerk von lose gekoppelten Services realisiert und modelliert werden.

Prozesse in Form von Services bereitstellt und eine Grundlage bietet, diese sicher zu nutzen (Papazoglou, Traverso, Dustdar & Leymann, 2008).

Die Definitionen für serviceorientierte Architekturen sind sehr weit gehalten. Im Kern beschäftigen sich alle drei Definitionen mit der Aufteilung von Software in einzelne Service Komponenten (Serviceorientierung). Erl und Papazoglu tendieren dazu, SOA als Architektur für Businessumgebungen zu definieren, da die Architektur flexibel in Bezug auf das Verändern von Geschäftsprozessen sein soll. Speziell Erl sieht SOA nur als notwendige Basis für viele weitere Konzepte in der verteilten Entwicklung von Businesssoftware auf Servicegrundlage. Er weitet SOA als Architektur für ganze Geschäftsfelder aus, womit SOA nicht mehr als reine Software-Architektur gelten kann, sondern eher ein Architekturkonzept für die gesamte Enterprise-IT eines Unternehmens ist. Darin können sich verschiedene IT-Systeme vereinen.

5.2 Motivation für SOA

Sehr hilfreich für die spätere Einordnung ist ein Blick auf die eigentliche Motivation für SOA. Jedes Unternehmen will durch wertschöpfende Geschäftsprozesse einen Gewinn erwirtschaften und befindet sich in einem Marktumfeld, an das es sich anpassen muss. Die Perspektive der Informationstechnologie vor einer SOA-Einführung erlaubt 4 Kernaussagen:

- Erhöhter Konkurrenzkampf und Anforderungen durch Globalisierung erfordern schnelle Anpassungen von Geschäftsprozessen.
- Informationstechnologie muss an Geschäftsprozesse angepasst werden, da die meisten Prozesse darauf angewiesen sind.
- Bestehende IT-Systeme sind unflexibel und eng verwoben, die Systeme werden obendrein von unternehmensinternen Organisationen verwaltet, die mehr eigene Interessen verfolgen als die Wertschöpfung des Unternehmens.
- Budget für IT geht in einzelne Anwendungen ohne konkreten Bezug zu wertschöpfenden Prozessen.

SOA versucht, diese Missstände durch ein besseres *Business/IT-Alignment* zu beheben. Das allem übergeordnete Ziel besteht darin, wertschöpfende Geschäftsprozesse schnell an veränderte Marktbedingungen anpassen zu können, ohne dabei lange auf die passenden Änderungen der IT zu warten (Starke, 2015).
Prozesslogik soll in einer SOA von Anwendungslogik getrennt werden, um geteilte Verantwortlichkeiten zu erreichen. So müssen Geschäftsprozesse nicht von IT-Abteilungen umgesetzt werden, sondern können von den fachlichen Entscheidungsträgern ohne IT-Kenntnisse entworfen und im Optimalfall automatisch ausgeführt werden (Bauer, Buchwald & Reichert, 2013). Des Weiteren spielen Vorteile aus technischer Sicht eine Rolle. Es soll mehr Flexibilität bei der Umsetzung von Projekten bezüglich Unternehmenssoftware erreicht werden und durch Wiederverwendung und die lose Kopplung von Services Entwicklungs- und Wartungskosten gespart werden (Buchwald, 2012). Die Motivation für SOA findet sich demnach in erster Linie auf unternehmerischer Seite und lässt sich zu einem großen Teil an dem Wunsch nach flexiblen Geschäftsprozess-Veränderungen festmachen. Untergeordnet spielen auch technische Vorteile eine Rolle.

5.3 Verständnis von SOA

Obwohl SOA hauptsächlich als Architekturstil für lose gekoppelte Software definiert wurde, haben sich mit der Zeit viele weitere Aspekte herausgebildet, die im allgemeinen Verständnis zu SOA dazugehören.

Integration In der Praxis wird SOA von IT-Architekten als Evolutionsstufe von Enterprise Application Integration (EAI) gesehen und beinhaltet somit auch zwangsläufig die Integration von Legacy-Systemen. SOA hat aus dieser Perspektive das Ziel, die aus EAI bekannten Wrapper-Adapter und die Message-oriented-Middleware durch Service-Adapter zu ersetzen. So soll bestehende Funktionalität aus Legacy-Systemen ebenfalls standardisiert zentral integriert werden, zum Beispiel über einen Enterprise Service Bus. Diese Sichtweise impliziert auch, dass SOA nicht nur als Architektur für einzelne Anwendungskontexte zu sehen ist, sondern einen viel größeren Rahmen der IT umfasst und somit als unternehmensweites Konzept gesehen werden kann (Clark, 2016).

Anwendungsentwicklung Gleichzeitig existiert die in Abschnitt 5.1 definierte Sicht, in der SOA als Strategie für lose gekoppelte Anwendungsentwicklung gesehen wird. Funktionalität von bestehenden Anwendungen soll in verschiedene unabhängige Services herausgebrochen werden und daraus ein Netzwerk von verschiedenen Services erschaffen werden. Über Komposition von Services soll schlussendlich die ursprüngliche Funktionalität erhalten bleiben und einfach erweitert werden können. Neue Anforderungen an Software soll ausschließlich als zusätzliche Services zu den bestehenden hinzugefügt werden.

Business In der Business-Perspektive sollen die vorhandenen technischen Services und die komponierten Services schlussendlich über Orchestrierung und Choreographie zu technischen Abbildungen von fachlichen Prozessen geformt werden. Über eine neue Orchestrierung einzelner Services lassen sich IT-unterstützte Geschäftsprozesse in der Theorie schnell abändern.

5.4 SOA im Vergleich mit Microservices

Der angesetzte Vergleich versucht die beiden Konzepte auf den unterschiedlichen Ebenen zu vergleichen, um daraus ein besseres Verständnis über Gemeinsamkeiten und Unterschiede zu erlangen.

5.4.1 Stil-Ebene

Flexibilität Beide Stile legen sehr viel Wert auf Flexibilität. Jedoch unterscheiden sie sich hierbei grundsätzlich in Flexibilität für Geschäftsprozesse (SOA) und Flexibilität in der Softwareentwicklung/Produktevolution (Microservices).

- **Buisness Process Felxibility:** Ein großer Punkt in einer aus der Business Perspektive getriebenen SOA ist flexibles Business Process Management (BPM). Die Anpassungsfähigkeit der Unternehmens-IT an veränderte Anforderungen und den damit einherge-

henden veränderten Prozessen ist das oberste Ziel einer erfolgreichen SOA-Einführung (Buchwald, Bauer & Reichert, 2011).
In bisher bekannten Microservice-Systemen lässt sich kein Hinweis auf eine Umsetzung von BPM erkennen. Generell sind bisherige Umsetzungen nicht darauf ausgerichtet, Geschäftsprozessänderungen zu vereinfachen, sondern sollen eine einfachere technische Handhabung der steigenden Komplexität im Zusammenhang mit Skalierung und Software-Entwicklung ermöglichen.

- **Software Development Flexibility:** Flexible Softwareentwicklung ist immer wünschenswert und ist einer der Hauptgründe für den Einsatz einer Microservice-Architektur. Dennoch sind auch die Grundprinzipien der Serviceorientierung darauf ausgelegt, Software modularer und entkoppelter zu entwickeln, als bei monolithischen Anwendungen der Fall ist.
  Aus technischer Perspektive unterscheiden sich SOA und MSA weniger stark, als wenn SOA aus der Business-Perspektive betrachtet wird. Die angewendeten serviceorientierten Prinzipien zielen beides mal auch darauf ab, lose gekoppelte Software zu entwickeln und einfacher deployen zu können.

Struktur der Funktionalität Stark unterschiedlich, vielleicht sogar das größte Unterscheidungsmerkmal, ist jedoch die Herangehensweise an die Serviceorientierung.
Der Microservice-Stil legt sehr starken Wert auf Domain-Driven-Design mit seinen Bounded Contexts und strukturiert die gesamte Entwicklung, inklusive der betrieblichen Verantwortlichkeiten und der Datenhoheit danach.
In einer SOA wird Wert auf Geschäftsprozesse gelegt und demnach die Funktionalität der Services daran orientiert. Auch die Datenhoheit obliegt meistens großen relationalen Systemen und wird zentral von Datenbank-Administratoren verwaltet.

Organisation Ein ebenfalls großer Unterschied existiert in der Organisation der Entwicklung. Microservices setzen sehr stark auf die Erkenntnisse aus Conway's Law und versuchen die Software-Entwicklung so weit wie möglich in unabhängige fachlich-orientierte Teams zu verlagern. Oft sind in diesen Teams nicht nur Softwareentwickler, die sich um die IT kümmern, sondern sie decken den kompletten Produktlebenszyklus ab, mit allen beteiligten funktionalen Ebenen, wie Produktplanung, Management, Entwicklung und Support.
In einer SOA sind Business- und IT-Abteilungen typischerweise getrennt, das „Allignment "dieser beiden Organe funktioniert in einer SOA über das Komponieren von technischen Services der IT-Abteilungen zu größeren Businessservices, die komplette Business-Funktionalitäten abdecken und von den Business-Abteilungen zu Geschäftsprozessen geformt werden können.

Flexibilität der Infrastruktur Microservices sind um einige Jahre jünger als SOA, weshalb der Stil auch auf ganz andere Grundlagen aufbauen kann. Eine klassische SOA basiert typischerweise auf der üblichen Unterteilung zwischen Entwicklern der Services und den Mitarbeitern für den Betrieb der Services. Die Services an sich sind typischerweise auch nicht täglichen Änderungen unterworfen und daher eher als stabil anzusehen. Veränderung geschieht in einer SOA mehr über eine veränderte Komposition oder eine neue Orchestrierung der Services. Das Augenmerk liegt daher nicht auf durchgängiger Automatisierung, welche ein schnelles Deployment

und flexible Skalierbarkeit erlauben würde.
Der Microservices-Stil bringt typischerweise eine ständige Veränderung der Services mit sich. Das Produkt, welches durch den Service repräsentiert ist, muss sich kontinuierlich weiterentwickeln und den schnell wechselnden Anforderungen der Nutzer folgen. Dazu werden verstärkt neue Technologien wie Container oder Virtual-Machine Images genutzt, die über automatisierte Pipelines in private oder öffentliche Cloud-Umgebungen deployt werden. Der Einsatz der Cloud erlaubt eine Angleichung von Entwicklung und Betrieb und daher das Umsetzen von DevOps innerhalb der unabhängiger Teams.

Technologie SOA hat sich in den letzten Jahren zu einem Stil entwickelt, der hauptsächlich von Anbietern teurer Softwareprodukte zur Umsetzung von SOA getrieben und gesteuert wird. Nicht selten basieren SOA-Umsetzungen auf sehr kostspieliger Middleware mit unendlich vielen Funktionen, von denen nur wenige effektiv genutzt werden. Innerhalb der Unternehmen, die Microservices umsetzen, wird momentan noch größtenteils auf Open-Source gesetzt. Das hängt unter anderem auch damit zusammen, dass sie Pioniere sind und mit technologischen Herausforderungen zu kämpfen haben, die vor ihnen noch niemand zu bewältigen hatte. Unternehmen wie Netflix, Zalando oder Amazon haben in Eigenregie Software-Tools und Technologien rund um Microservices herum entwickelt, weil es schlicht keine Anbieter gab, die ihre unüblich hohen Anforderungen erfüllen konnten.

5.4.2 Architektur-Ebene

Service Spezifikation Auf der Architektur-Ebene ist zu bemerken, dass eine Microservice-Architektur ihre Komponenten klar definiert, es dürfen nur Microservices vorkommen. Eine SOA hält sich mit der Spezifikation der Services sehr zurück und fordert lediglich, dass diese unabhängig sind und über klar definierte Schnittstellen angesprochen werden sollen. Für die Services gibt es keine konkreten Vorgaben, sie sollen vielmehr möglichst viele der serviceorientierten Prinzipien (vgl. Abschnitt 5.1) erfüllen. Daher kann es viele verschiedene Servicearten innerhalb einer SOA geben.

Dezentralisierung Bei SOA ist ein erster Ansatz der Dezentralisierung erkennbar gewesen. Geschäftslogik, welche früher monolithisch implementiert wurde, ist bei SOA in einzelne Services unterteilt. Allerdings verläuft die Kommunikation meistens über eine zentrale Middleware, die eigene Logik enthält und entsprechend in den Workflow der Anwendung eingreift. Die Services basieren häufig auf unterliegenden zentralen Systemen, wie beispielsweise einem System für *Customer Relationship Management (CRM) oder Enterprise Resource Planning (ERP)* und müssen sich an dem gegebenen Datenschema orientieren beziehungsweise die zentrale Middleware muss eine Datentransformation unterstützten.

Microservices treiben die Dezentralisierung weiter voran und lassen neben Geschäftslogik und Daten auch Teile des User-Interfaces in Services auslagern. Der Sinn von Microservices ist es nicht, bestehende Systeme modular zu kapseln und deren Funktionalität in einzelnen Services zur Verfügung zu stellen, vielmehr ist jeder Microservice eine eigene Anwendung mit eigener Logik. Ein Microservice basiert daher nicht auf gegebenen CRM- oder ERP-Systemen, sondern das ERP- oder CRM-System ist über viele Microservices in Form einer Microservice-

Architektur realisiert. Aus Performance Gründen setzen Microservices auf leichtgewichtige und effiziente Kommunikation. Einen zentralen ESB, wie er sich in SOA etabliert hat, wird man in Microservice-Architekturen nicht finden, da dieser durch zentrale Abhängigkeiten die flexible Entwicklung an den einzelnen Services behindern würde und Probleme im Zusammenhang mit der Skalierbarkeit des Systems hat.

5.4.3 Service-Ebene

Arten In einer SOA findet man Services typischerweise eingeteilt in verschiedene Service-Ebenen (Buchwald, Bauer & Pryss, 2009):

- **Elementare Services:** Sie beschreiben die eigentlichen technischen Services, welche auf der untersten Ebene vorhandene IT-Systeme kapseln und modular über standardisierte Schnittstellen zur Verfügung stellen.
- **Zusammengesetzte Services:** Geschäftsprozesse überspannen meistens mehrere unterliegende IT-Systeme und erfordern deshalb, dass Services unterschiedlicher Funktionalität zu einem größeren Service komponiert werden. Beispielsweise kann dies über eine Orchestrierung innerhalb des ESB erfolgen. Sie bilden die fachliche Funktionalität des Geschäftsprozesses ab und abstrahieren die unterschiedlichen technischen Basis-Systeme.
- **Öffentliche Services:** Sind elementare oder zusammengesetzte Services. Sie sind letztendlich die für fachliche Personen veröffentlichte Services, welche diese in ihrem Geschäftsprozessdesign verwenden können.

Microservices nehmen eine solche Einteilung nicht vor, da die Services nicht auf unterliegenden Systemen beruhen, sondern direkt eine fachlich abgeschlossene Aufgabe implementieren sollen. Ein Team kann sich jedoch innerhalb eines Bounded Context auch dazu entscheiden, ihre Services in verschiedene Layer einzuteilen. So könnte es Microservices für das User-Interface dieses Bounded Context geben, andere Microservices, welche die Persistenz abstrahieren und schließlich die Microservices für die Logik des Bounded Context. Von außerhalb des Bounded Context bleibt die Funktionalität aber auch die Schnittstellen der Logik beschränkt.

Schnittstellen Weder Microservices, noch SOA machen konkrete Vorgaben über Schnittstellentechnologien. Trotzdem hat sich mit der *Web Service Description Language* und SOAP ein eigener Standard für die Schnittstellenspezifikation und die Kommunikation zusammen mit SOA für deren Webservices entwickelt. Schnittstellen sind dadurch standardisiert und maschinenlesbar spezifiziert. Mit der Zeit hat sich aber auch REST als Alternative in SOA-Umgebungen hervorgetan, weshalb man in einer SOA heute meist von SOAP-Services beziehungsweise Webservices oder REST-Services redet.
Auch Microservices nutzen wegen ihrer Einfachheit gerne REST-Schnittstellen, wobei große Unternehmen wegen der Effizienz inzwischen auch binäre-RPC-Technologien wie zum Beispiel *Thrift* einsetzen. Dafür wird man SOAP-Schnittstellen in einer Microservice-Architektur wegen ihrer Komplexität und der Ineffizienz eher selten sehen.

Wiederverwendbarkeit Bei SOA sind Services auf maximale Wiederverwendbarkeit ausgelegt, sodass verschiedene Funktionalitäten durch Komposition von mehreren Services geschehen

kann. Der Entwicklungsaufwand soll durch das mehrfache Verwenden von Services reduziert werden, weil bestimmte Funktionalitäten nicht mehrfach implementiert werden. SOA folgt dem *Don't repeat yourself (DRY)*-Prinzip, was auch als *Share-Everything* angesehen werden kann. Ein geschriebener Code sollte möglichst nicht an einer anderen Stelle im System nochmals geschrieben werden müssen. Dafür gibt es oft Shared-Libraries innerhalb des Anwendungsservers oder im ESB, welche gemeinsam genutzte Geschäftslogik enthalten. Alternativ dazu wird eine solche Funktionalität auch in eigenen Core-Services implementiert und kann von anderen Services über Aufrufe genutzt werden.
Microservices legen keinen Fokus auf serviceübergreifende beziehungsweise kontextübergreifende Wiederverwendbarkeit. Im Gegenteil zu SOA favorisiert man bei Microservices einen *Shared-nothing*-Ansatz, um die Eigenständigkeit der Service-Evolution zu wahren. Geteilte Funktionalität über Teamgrenzen hinweg integriert Abhängigkeiten zwischen den Teams, da beim Verändern des geteilten Codes alle nutzenden Teams betroffen sind. In der Regel schreiben alle Teams ihre komplette Funktionalität selber und verwenden nur kontextspezifische Funktionalitäten von anderen Services. Es gibt jedoch trotzdem Möglichkeiten, ein zentrales Repository im Unternehmen zu führen, in das Teams versionierte Bibliotheken als Open-Source einbringen können, von denen sie denken, dass andere Teams daraus einen Nutzen ziehen können. Teams können bestimmte Versionen solcher Bibliotheken eigenverantwortlich nutzen und unabhängig Updaten.

Deployment SOA-Systeme in Unternehmen basieren häufig auf JavaEE oder der .Net Platform. Dabei werden die einzelnen Services meist in Anwendungsserver deployt, der den Lebenszyklus verwaltet und die Shared-Libraries bereitstellt. Der Microservice-Stil verwendet meist das *One Service per Host*-Pattern, wobei der Host meistens eine virtuelle Maschine ist. Durch aufkommende Container-Technologien wie *Docker* kann dieses Pattern noch effizienter genutzt werden, da die Unterteilung der Rechenkapazität wegen des kleineren Overheads noch feiner eingeteilt werden kann.

5.4.4 Motivations-Ebene

Kapitel 4 untersucht die Motivation für bisherige Umsetzungen des Microservices-Stils. Die Gründe für Microservices unterscheiden sich von den Gründen, warum ein Unternehmen eine SOA im klassischen Verständnis (vgl. Abschnitt 5.3) einführt.

Tabelle 5.1: Unterschiedliche Gründe für Microservices und SOA

Microservices	**SOA**
-Skalierungsprobleme	-Business-IT Alignment
-Unabhängigkeit der entwickelnden Teams	-Flexible Veränderung von Geschäftsprozessen
-Continuous Delivery / DevOps	-Enterprise Application Integration vereinfachen
-Innovationsfreundliche Kultur	-Wiederverwendung bestehender Funktionalitäten
-Passend für einen Betrieb in der Cloud	-Einfachere Wartung durch lose Kopplung
-Betriebsstabilität für Hochverfügbarkeit	-Vereinfachtes zentralisiertes Reporting

5.5 Ähnlichkeiten

Offensichtlich haben SOA und Microservices auch viele Gemeinsamkeiten und historisch hat sich der Microservice-Stil aus dem SOA-Stil heraus entwickelt. Bevor es den Begriff an sich gab, haben Microservice-Pioniere ihre Konzepte ebenfalls als SOA (Amazon) oder *fine-grained SOA* (Netflix) bezeichnet. Die gemeinsamen Kernpunkte sind gleichzeitig ein gemeinsames Unterscheidungsmerkmal gegenüber einem monolithischen Stil:

- Aufteilung der Funktionalität in abgeschlossene Services
- Bereitstellung von Funktionen über definierte Schnittstellen
- Möglichst unabhängiges Deployment der einzelnen Services für eine einfache Wartung

Zusätzlich haben beide Stile gemeinsam, dass sie mehr sind als nur reine Software-Architekturen. Weder Microservices noch SOA können einfach als Produkt *gekauft* werden. Sie bestehen aus Best-Practices, Guidelines und können in unterschiedlichen Varianten auftreten. Damit die gewünschten Effekte einer SOA oder von Microservices erzielt werden können, reicht es nicht, sich an die technologischen Vorgaben zu halten. Beide Stile müssen mit einer passenden Unternehmenskultur verbunden werden, welche weit über die IT hinaus Veränderungen mit sich bringt und konsequent gelebt werden muss.

5.6 Versuch einer Einordnung der Begriffe

Basierend auf den verschiedenen Definitionen für SOA und den Erkenntnissen dieser Ausarbeitung, wird eine Einordnung der Begrifflichkeiten nach dem Schema in Abbildung 5.1 vorgeschlagen. Wir halten dazu folgende Fakten fest:

- **Serviceorientierte Architekturen** beschränken sich auf die Kernidee, Software in Services aufzuteilen. Sie haben keinen expliziten Fokus auf BPM oder EAI.
- Eine **Mediation-Driven-SOA** bezeichnet zentralisierte Realisierungen auf Basis einer intelligenten Middleware für EAI und BPM.
- Mediation-Driven-SOAs sind im globalen Unternehmenskontext zu sehen.
- **Microservice-Architekturen** beschränken sich in erster Linie auf Anwendungssysteme und enthalten ausschließlich Microservices.
- Eine **Microservice-Architektur** ist eine spezielle Form von serviceorientierten Architekturen.
- **Microservices, REST-Services und SOAP-Webservices** sind unterschiedliche Arten von Services im Kontext von serviceorientierten Architekturen.
- Eine SOA kann verschiedene Servicearten vereinen und gemeinsam nutzen.
- Der **Microservices-Stil** existiert im Zusammenhang mit einem Unternehmen oder einer eigenständigen und bevollmächtigten Unternehmenssparte, welche ein konkretes Anwendungssystem entwickelt und führt in jedem Fall zu einer Microservice-Architektur.

- Der **SOA-Stil** ist im klassischen Enterprise-Kontext zu sehen und fokussiert auf die Optimierung und Flexibilität von Geschäftsprozessen und Integration. Daraus wird in der Regel eine zentralisierte Mediation-Driven-SOA resultieren.

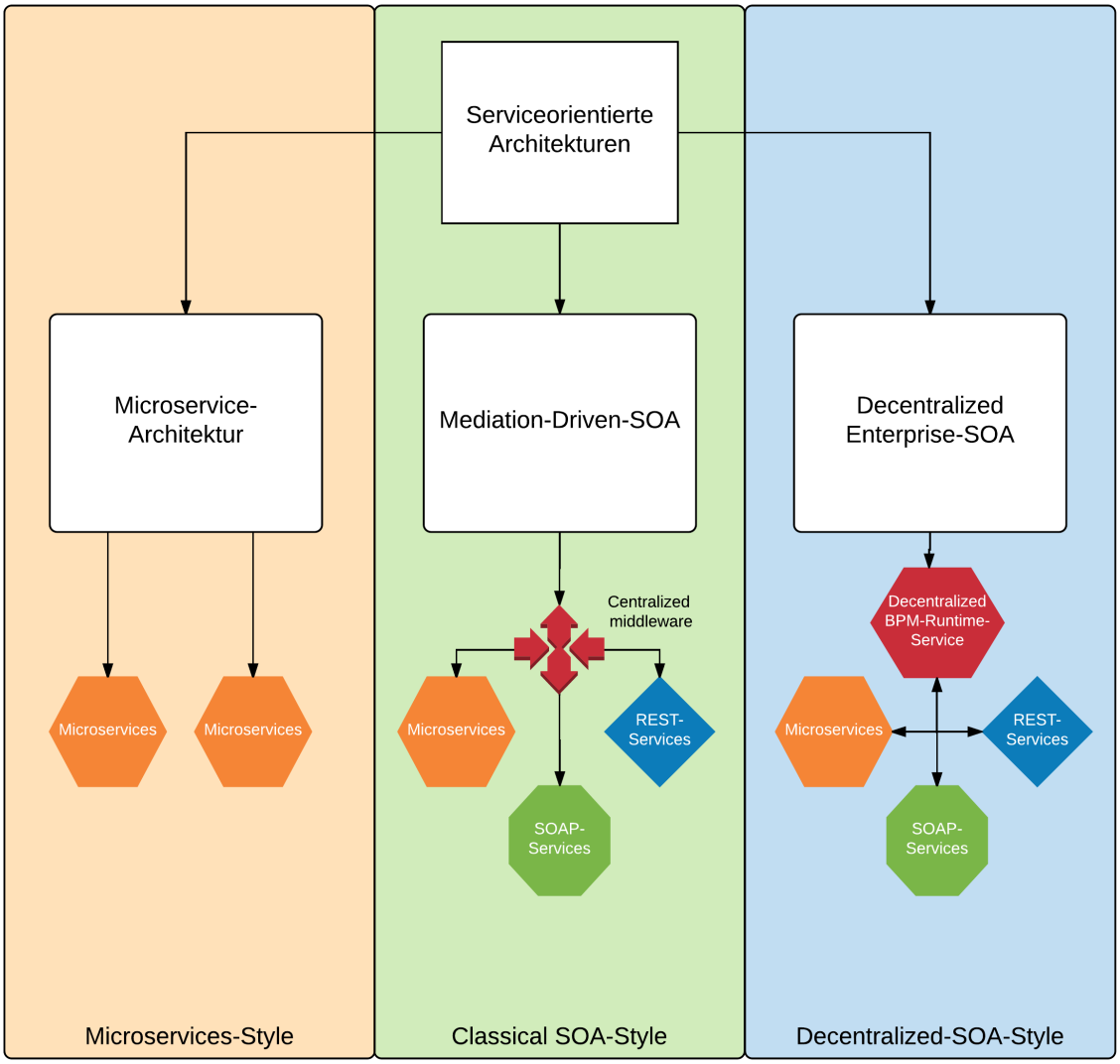

Abbildung 5.1: Grafische Darstellung von Zusammenhängen der Neuordnung.

5.6.1 Einordnung

Eine klare Trennung zwischen Microservices und SOA kann nicht vorgenommen werden. Es ist eine Tatsache, dass die Microservice-Architektur bei Amazon und anderen Pionieren aus serviceorientierten Architekturen heraus entstanden ist und zu einem eigenen Stil abgewandelt und erweitert wurde, um ihren speziellen Anforderungen zu genügen. Die Microservice-Architektur ist daher vielmehr eine Konkretisierung der ursprünglichen Ideen für eine serviceorientierte Architektur, genauso wie Mediation-Driven-SOAs eine Konkretisierung der ursprünglichen SOA-

Idee mit Fokus auf BPM und EAI ist. Jede SOA-Umsetzung folgt einem speziellen Stil, der festlegt was für eine konkrete SOA letztendlich erreicht wird und wie SOA umgesetzt wird.

Die Microservice-Architektur ist demnach ein Ergebnis einer konsequenten Umsetzung des Microservice-Stils und soll für Flexibilität in der Weiterentwicklung und Skalierung eines Anwendungssystems sorgen.

Die Mediation-Driven-SOA ist die klassische SOA-Umsetzung der letzten Jahre und basiert auf zentralen Middlewaresystemen, wie beispielsweise einem Enterprise-Service-Bus. Ihr Fokus liegt auf der unternehmensweiten Integration von verschiedenen Legacy-Systemen und der flexiblen Geschäftsprozessanpassung der IT über Orchestrierung und Choreographie von unterschiedlichen Services. Diese Services können ebenfalls Microservices sein.

Die dezentralisierte Enterprise-SOA umfasst andere SOA-Konzepte, die nicht ausschließlich auf Microservices basieren, aber trotzdem dezentral organisiert sind. So könnte beispielsweise eine dezentralisierte Enterprise-SOA verschiedene Systeme über herkömmliche Services zur Verfügung stellen, neue Microservices nutzen und zugleich eine Prozess-Engine als eigenständigen Service zur Verfügung stellen, mit der Geschäftsprozesse organisiert werden können.

5.7 Fazit

Zusammenfassend kann man sagen, dass SOA und Microservices in ihren grundlegenden Ansätzen viel gemeinsam haben. Da sich SOA über die Jahre in sehr vielen Fassetten gezeigt hat, kann man auch Kritikern, die meinen, „Microservices sind genau das, was wir in unserem Unternehmen seit Jahren als SOA bezeichnen und daher nichts Neues," durchaus recht geben. Auch in bisherigen SOA Implementierungen können Services Charakteristiken von Microservices aufweisen, da SOA seine Services nicht konkret spezifiziert.
Allerdings besteht Microservices nicht nur aus der Spezifikation der Services, sondern ist ein Stil, wie Software flexibel und skalierbar entwickelt werden kann. Hierbei unterscheiden sich SOA und Microservices sehr stark und sind durch ganz neue Anforderungen entstanden. Der SOA-Stil ist Enterprise-zentriert und fokussiert auf Probleme mit der Integration von vorhandenen Systemen und der Optimierung von Geschäftsprozessen. Microservices fokussiert auf eine intelligente Skalierung der Systeme in der Cloud mit einer sehr großen Anzahl von Nutzern. Der SOA-Stil legt seinen Fokus auf zentrale Regulierung und Steuerung der Services über einen ESB, Microservices hingegen versuchen den einzelnen Teams der Services so viel Autonomie wie möglich zu geben und fokussieren auf ein möglichst dezentrales System ohne *Single Point of Failures*.

6 Zusammenfassung

6.1 Ausblick: Serverlose Architekturen

Der Microservices-Stil ist aufgrund seines jungen Alters noch in ständigem Wandel. Neue Technologien und Tools ermöglichen eine immer umfangreichere Automatisierung des Betriebs und stärkeren Fokus auf die DevOps-Praxis. Aktuell werden im Zusammenhang mit Microservices die Gattung der *Serverlosen Architekturen* immer interessanter.

Redet man bei Monolithen noch von großen leistungsstarken Serverinstanzen und bei Microservices von vielen kleineren Rechnerinstanzen, so geht die serverlose Architektur dazu über, Rechnerinstanzen komplett zu abstrahieren und bei einem Anbieter von *Function as a Service (FaaS)* verwalten zu lassen. Dieser ermöglicht es, eigenen Code über ein System von Regeln und APIs zu triggern und damit zum Beispiel eigene Java oder Ruby Funktionen kurzlebig ohne das Aufsetzen einer Serverinstanz auszuführen. Somit entfällt das Vorhalten einer eigenen Instanz für sehr schwach belastete Microservices. Stattdessen gibt es eine Art Pay-per-Use Bezahlmodell. Das kann bei schwach belasteten Microservices oder bei Services, die bis auf wenige Lastspitzen wenig Anfrageaufkommen haben, viel effektiver in Bezug auf die Betriebskosten der Infrastruktur sein (Villamizar et al., 2016).
Ein weiterer Vorteil ist, dass ein aufwändiger Deployment-Prozess entfällt und das verantwortliche Team sich nicht mehr um die intelligente Skalierung der eigenen Funktionalität kümmern muss, diese wird vom Anbieter elastisch automatisiert vorgenommen (Roberts, 2016).

Bekannte Anbieter von FaaS sind Google Cloud Functions (Google, 2016), Microsoft Azure Functions (Microsoft, 2016), IBM OpenWhisk (IBM, 2015) und Amazon AWS Lambda (Amazon Web Services, 2016a).

6.2 Fazit

Microservices als Stil ist in der Tat etwas Neuartiges und ist klar abzugrenzen von herkömmlichen Praxen in der Softwareentwicklung und IT-Architektur. Er zeichnet sich gegenüber anderen Stilen als sehr flexibel und innovationsfördernd für Softwareentwicklung im großen Stil aus. Das ist aus heutiger Sicht auch der Haupteinsatzzweck eines Microservice-Stils. Er ermöglicht es Online-Unternehmen, ihre Softwaredienste flexibel weiterzuentwickeln und den Anforderungen entsprechend beliebig stark horizontal zu skalieren, sodass diese dem erwarteten Wachstum ihres Geschäftsfeldes gerecht werden können.

In Kapitel 2 wurde eine mögliche Einteilung des Begriffs Microservices in seine Katego-

rien Service, Architektur und Stil gezeigt. Durch die Einteilung können die unterschiedlichen Gesichtspunkte von Microservices besser dargestellt werden.
Der Begriff umfasst zunächst die Art wie Microservices aussehen. Sie sind als kleine eigenständige Deployment-Einheiten und klarem fachlichem Bezug definiert und sind durch eigene Datenhaltung sowie oft auch ein eigenständiges User-Interface möglichst unabhängig vom Rest des Systems.
Des Weiteren umfasst Microservices eine eigene Architektur, die sich durch ein Höchstmaß an Dezentralisierung auszeichnet und ausschließlich aus Microservices besteht. Die Kommunikation erfolgt über einen leichtgewichtigen REST-Ansatz oder über sehr einfache Message-Queues für eventbasierte Kommunikation.
Zuletzt beschreibt der Begriff den ganz wichtigen Gesichtspunkt des Entwicklungs-Stils. Microservices sind vielmehr als nur eine neue Art von Services, die über eine bestimmte Architektur zusammengeschaltet sind. Der Begriff umfasst eine ganze Reihe an kulturellen Veränderungen bis hin zu autonomen Teams mit viel Entscheidungsfreiheit in Bezug auf Deployment und Auswahl der Technologie. Andererseits werden auch Methoden wie Domain-Driven Design angewandt, damit möglichst unabhängige Bounded Contexts identifiziert werden können, um welche sich die ganze Einteilung der Teams und Verantwortlichkeiten bei der Anwendung des Microservices-Stils dreht. Anwender des Stils ordnen alles einer hohen Entwicklungsgeschwindigkeit unter und nehmen dafür auch Fehlentscheidungen in Kauf, die durch viel Autonomie entstehen können. Kennzeichnend dafür ist das Streben nach Konzepten wie Continuous Delivery und DevOps, womit der Zyklus, in welchem neue Versionen entwickelt und deployt werden, immer schneller durchlaufen werden soll.
In Kapitel 3 wurden die von Fowler und Lews beschriebenen Charakteristiken des Microservices-Stils näher erläutert und um bekannte Herausforderungen und Beispiele von praktischen Vorkommen der Charakteristiken ergänzt.
In Kapitel 4 wurden Unternehmen, die Microservices umsetzen, analysiert. Anhand von Veröffentlichungen in eigenen Technik-Blogs, auf Konferenzen oder über Whitepaper wurden Gründe und Motivationen für ein Umsetzen von Microservices gesucht und zusammengefasst.
In Kapitel 5 wurde das Thema Microservices mit den älteren serviceorientierten Architekturen und dem damit einhergehenden SOA-Stil verglichen. Dazu wurden die ursprünglichen Ansätze von SOA um das heute allgemeine Verständnis über SOA erweitert und die Motivationen für SOA-Einführungen erläutert. Die Erkenntnisse aus den vorherigen Kapiteln wurden dahingehend verarbeitet, dass ein Vergleich von den unterschiedlichen Ebenen der beiden Konzepte vorgenommen wurde.
So gleichen sich die grundsätzlichen Ideen von SOA und Microservices. Sie sind aber vor einem ganz anderen Hintergrund entstanden und greifen unterschiedliche Probleme auf. SOA ist sehr Enterprise-basiert und fokussiert auf die Integration unterschiedlicher geschäftlicher Anwendungen als auch auf die Art und Weise, wie diese technischen Systeme möglichst flexibel auf fachlich definierte Prozesse abgebildet werden können.
Microservices kümmern sich um die Probleme in der Entwicklung und dem Betrieb von großen Online-Softwaresystemen. Die Geschwindigkeit mit der das System weiterentwickelt werden kann, steht absolut im Vordergrund, weshalb es möglichst keine zentralen Abhängigkeiten geben darf, die einzelne Teams blockieren könnte.

Abbildungsverzeichnis

Literaturverzeichnis

Amazon Web Services. (2014, November). *AWS re:Invent 2014 | (ENT209) Netflix Cloud Migration, DevOps and Distributed Systems.* Zugriff am 2016-08-31 auf `https://www.youtube.com/watch?v=Mjj7nIt5G_Y`

Amazon Web Services. (2016a). *AWS Lambda Data Processing - Datenverarbeitungsdienste.* Zugriff am 2016-11-16 auf `//aws.amazon.com/de/lambda/`

Amazon Web Services. (2016b). *SOAP Requests - Amazon Elastic Compute Cloud.* Zugriff am 2016-08-03 auf `http://docs.aws.amazon.com/AWSEC2/latest/APIReference/using-soap-api.html`

Apache Software Foundation. (2016). *Apache Thrift - Library.* Zugriff am 2016-08-31 auf `https://thrift.apache.org/lib/`

Apple, L. (2015a). *Radical Agility with Autonomous Teams and Microservices in the Cloud – Zalando Tech Blog.* Zugriff am 2016-11-01 auf `https://tech.zalando.com/blog/radical-agility-with-autonomous-teams-and-microservices-in-the-cloud/`

Apple, L. (2015b). *So, You've Heard About "Radical Agility"... (Video) – Zalando Tech Blog.* Zugriff am 2016-11-01 auf `https://tech.zalando.com/blog/so-youve-heard-about-radical-agility...-video/`

Atlassian. (2016). *SOAP and XML-RPC API Deprecation Notice - Atlassian Developers.* Zugriff am 2016-08-03 auf `https://developer.atlassian.com/jiradev/latest-updates/soap-and-xml-rpc-api-deprecation-notice`

Bauer, T., Buchwald, S. & Reichert, M. (2013, Juni). Improving the Quality and Cost-effectiveness of Process-oriented, Service-driven Applications: Techniques for Enriching Business Process Models. In *Service-Driven Approaches to Architecture and Enterprise Integration* (S. 104–134).

Bennett, K. H. & Rajlich, V. T. (2000). Software maintenance and evolution: A roadmap. In *Proceedings of the conference on the future of software engineering* (S. 73–87). ACM. doi: 10.1145/336512.336534

Brooks, F. P. (1995). *The mythical man-month: essays on software engineering* (Anniversary ed Aufl.). Addison-Wesley Pub. Co.

Brunnert, A., van Hoorn, A., Willnecker, F., Danciu, A., Hasselbring, W., Heger, C., ... Wert, A. (2015, August). Performance-oriented DevOps: A Research Agenda. *arXiv:1508.04752 [cs].* (arXiv: 1508.04752)

Buchwald, S. (2012). *Erhöhung der Flexibilität und Durchgängigkeit prozessorientierter Applikationen mittels Service-Orientierung* (phd). University of Ulm.

Buchwald, S., Bauer, T. & Pryss, R. (2009, März). IT-Infrastrukturen für flexible, service-orientierte Anwendungen - ein Rahmenwerk zur Bewertung. In (S. 524–543). Münster, Germany: Koellen-Verlag.

Buchwald, S., Bauer, T. & Reichert, M. (2011, November). Bridging the Gap Between Business Process Models and Service Composition Specifications. In *Service Life Cycle Tools and Technologies: Methods, Trends and Advances* (S. 124–153). Idea Group Referenc.

Calçado, P. (2015, August). *How we ended up with microservices.* [Blog]. Zugriff am 2016-08-18 auf `http://philcalcado.com/2015/09/08/how_we_ended_up_with_microservices.html`

Capgemini. (2014). *Studie IT-Trends 2014: IT-Kompetenz im Management steigt* [Studie]. Zugriff am 2016-07-24 auf `https://www.de.capgemini.com/resource-file-access/resource/pdf/capgemini-it-trends-studie-2014.pdf`

Clark, K. J. (2016, Januar). *Microservices, SOA, and APIs: Friends or enemies?* [Technical Library]. Zugriff am 2016-08-14 auf `http://www.ibm.com/developerworks/websphere/library/techarticles/1601_clark-trs/1601_clark.html`

Conway, M. E. (1968). How do committees invent. *Datamation*, *14* (4), 28–31.

Daigneau, R. (2011). *Service Design Patterns: Fundamental Design Solutions for SOAP/WSDL and RESTful Web Services.* Addison-Wesley. (Google-Books-ID: tK3_vB304bEC)

Dragoni, N., Giallorenzo, S., Lafuente, A. L., Mazzara, M., Montesi, F., Mustafin, R. & Safina, L. (2016, Juni). Microservices: yesterday, today, and tomorrow. *arXiv:1606.04036 [cs]*. (arXiv: 1606.04036)

Duvall, P. M., Matyas, S. & Glover, A. (2007). *Continuous integration: improving software quality and reducing risk.* Upper Saddle River, NJ: Addison-Wesley. (OCLC: ocn122423674)

Dyck, A., Penners, R. & Lichter, H. (2015). Towards Definitions for Release Engineering and DevOps. In *Proceedings of the Third International Workshop on Release Engineering* (S. 3–3). Piscataway, NJ, USA: IEEE Press.

Erl, T., Chelliah, P., Gee, C., Kress, J., Maier, B., Normann, H., ... Utschig, C. (2014, Oktober). Applying Service-Orientation. In *Next Generation SOA: A Concise Introduction to Service Technology & Service-Orientation.* Prentice Hall.

Familiar, B. (2015). *Microservices, IoT, and Azure.* Berkeley, CA: Apress.

Feng, X., Shen, J. & Fan, Y. (2009). REST: An alternative to RPC for web services architecture. In *First international conference on future information networks, 2009. ICFIN 2009* (S. 7–10). doi: 10.1109/ICFIN.2009.5339611

Fielding, R. & Reschke, J. (2014). *Hypertext transfer protocol (HTTP/1.1): Semantics and content.* Zugriff am 2016-05-18 auf `https://tools.ietf.org/html/rfc7231`

Fielding, Roy Thomas. (2000). *Architectural styles and the design of network-based software architectures* (phdthesis). Zugriff am 2016-05-18 auf `https://www.ics.uci.edu/~fielding/pubs/dissertation/fielding_dissertation.pdf`

Fisher, M. (2008, Mai). *Splitting Applications or Services for Scale.* Zugriff am 2016-08-31 auf `http://akfpartners.com/techblog/2008/05/08/splitting-applications-or-services-for-scale/`

Fowler, M. (2006, Januar). *Continuous Integration.* Zugriff am 2016-08-06 auf `http://martinfowler.com/articles/continuousIntegration.html`

Fowler, M. & Lewis, J. (2014). *Microservices - a definition of this new architectural term.* Zugriff am 2016-07-30 auf `http://martinfowler.com/articles/microservices.html`

Gilbert, S. & Lynch, N. (2002, Juni). Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-tolerant Web Services. *SIGACT News*, *33* (2), 51–59. doi: 10.1145/564585.564601

Google. (2016). *Cloud Functions - Serverless Microservices.* Zugriff am 2016-11-16 auf `https://cloud.google.com/functions/`

Google Trends. (2016). Zugriff am 2016-08-17 auf `https://www.google.de/trends/explore`

Hardt, D. (2012, Oktober). *The OAuth 2.0 Authorization Framework* [Request for Comments]. Zugriff am 2016-10-29 auf `https://tools.ietf.org/html/rfc6749`

Hohmann, L. (2003). *Beyond Software Architecture: Creating and Sustaining Winning Solutions* (1edition Aufl.). Boston: Addison-Wesley Professional.

IBM. (2015, April). *OpenWhisk.* Zugriff am 2016-11-16 auf `https://developer.ibm.com/openwhisk/`

Jacobs, M. (2016, April). *Netflix/Hystrix.* Zugriff am 2016-10-21 auf `https://github.com/Netflix/Hystrix`

Kniberg, H. & Ivarsson, A. (2012). *Scaling agile @ spotify with tribes, squads, chapters and guilds.* Zugriff auf `https://dl.dropbox.com/u/1018963/Articles/SpotifyScaling.pdf`

Lantin, M. (2014, Februar). *Auto-Skalierung von Windows Azure Cloud Services.* Zugriff am 2016-08-08 auf `https://blogs.msdn.microsoft.com/malte_lantin/2014/02/11/auto-skalierung-von-windows-azure-cloud-services/`

Lews, J. (2012). *Micro services - java the unix way* [Conference Talk]. Conference Talk. Zugriff am 2016-07-30 auf `http://2012.33degree.org/pdf/JamesLewisMicroServices.pdf` (33rd Degree Conference)

Microsoft. (2016). *Azure Functions—Serverless Architecture | Microsoft Azure.* Zugriff am 2016-11-16 auf `https://azure.microsoft.com/en-us/services/functions/`

Natis, Y. V. (2003, April). *Service-Oriented Architecture Scenario.* Zugriff am 2016-08-13 auf `https://www.gartner.com/doc/391595/serviceoriented-architecture-scenario`

Newman, S. (2015). *Microservices: Konzeption und Design* (1. Aufl Aufl.; K. Lorenzen, Übers.). Frechen: mitp-Verlag.

Novotny, S. (2015). *"don't forget conway's law"* [Keynote]. Keynote. Zugriff am 2016-07-28 auf `https://www.youtube.com/watch?v=zA1EXJV1JWQ` (O'Reilly Software Architecture Conference)

O'Hanlon, C. (2006). A Conversation with Werner Vogels. *Queue*, *4* (4), 14:14–14:22. doi: 10.1145/1142055.1142065

Oracle. (2016, April). *SOAP API Deprecation (UPDATED) | Oracle Community.* Zugriff am 2016-08-03 auf `https://community.oracle.com/docs/DOC-922190`

O'Reilly, B., Molesky, J. & Humble, J. (2014, Dezember). The Fundamentals of Continuous Delivery. In *Lean Enterprise.* O'Reilly Media, Inc.

Papazoglou, M. P., Traverso, P., Dustdar, S. & Leymann, F. (2008, Juni). Service-oriented computing: a research roadmap. *International Journal of Cooperative Information Systems*, *17* (02), 223–255. doi: 10.1142/S0218843008001816

Raymond, E. S. (2008). *The art of UNIX programming: [with contributions from thirteen UNIX pioneers, including its inventor, ken thompson]* (Nachdr. Aufl.). Addison-Wesley.

Roberts, M. (2016, August). *Serverless Architectures.* Zugriff am 2016-11-16 auf `http://martinfowler.com/articles/serverless.html`

Sarkar, S., Ramachandran, S., Kumar, G. S., Iyengar, M. K., Rangarajan, K. & Sivagnanam, S. (2009). Modularization of a large-scale business application: A case study. *IEEE Software*, *26* (2), 28–35. doi: 10.1109/MS.2009.42

Sato, D. (2014, Juni). *CanaryRelease.* Zugriff am 2016-08-04 auf `http://martinfowler.com/bliki/CanaryRelease.html`

Skurrie, B. (2015). *Consumer-Driven Contracts with Pact (Sydney API Days 2015)* [Technology]. Zugriff am 2016-11-13 auf `http://www.slideshare.net/bethesque/pact-44565612`

Slee, M., Agarwal, A. & Kwiatkowski, M. (2007). Thrift: Scalable cross-language services implementation. *Facebook White Paper*, *5* (8). Zugriff am 2016-08-12 auf `http://docs.huihoo.com/thrift/thrift-20070401.pdf`

Starke, G. (2015). *Effektive Softwarearchitekturen: Ein praktischer Leitfaden.* Carl Hanser Verlag GmbH Co KG. (Google-Books-ID: zwURCgAAQBAJ)

Stine, M. (2014, Juni). *Cloud Foundry and Microservices: A Mutualistic Symbiotic Relationship.* Zugriff am 2016-08-01 auf `http://www.slideshare.net/mstine/microservices-cf-summit`

Sumaray, A. & Makki, S. K. (2012). A Comparison of Data Serialization Formats for Optimal Efficiency on a Mobile Platform. In *Proceedings of the 6th International Conference on Ubiquitous Information Management and Communication* (S. 48:1–48:6). New York, NY, USA: ACM. doi: 10.1145/2184751.2184810

Tholomé, E. (2009, August). *A well earned retirement for the SOAP Search API - The official Google Code blog* [Blog]. Zugriff am 2016-08-03 auf `http://googlecode.blogspot.com/2009/08/well-earned-retirement-for-soap-search.html`

Thones, J. (2015). Microservices. *Software, IEEE*, *32* (1), 116–116.

TiE SiliconValley. (2014, Juni). *TiEcon 2014: Netflix: Lessons Learned in Becoming a Cloud Native Company.* Zugriff am 2016-09-01 auf `https://www.youtube.com/watch?v=BTFRWsniH3k`

Villamizar, M., Garcés, O., Ochoa, L., Castro, H., Salamanca, L., Verano, M., ... Lang, M. (2016). Infrastructure Cost Comparison of Running Web Applications in the Cloud Using AWS Lambda and Monolithic and Microservice Architectures. In *2016 16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)* (S. 179–182). doi: 10.1109/CCGrid.2016.37

Vogels, W. (2011). *Amazon and the Lean Cloud.* Zugriff am 2016-10-24 auf `https://vimeo.com/29719577`

Warner, W. P. (1982). *What is a software engineering environment (SEE). (desired characteristics).* (Nr. NSWC/TR-82-465).

Wolff, E. (2016). *Microservices: Grundlagen flexibler softwarearchitekturen* (1. Auflage Aufl.). dpunkt.verlag.

Woolf, B. (2007). *ESB-oriented architecture: The wrong approach to adopting SOA* [Blog]. Zugriff am 2016-05-20 auf `https://web.archive.org/web/20071014193332/http://www.ibm.com/developerworks/library/ws-soa-esbarch/index.html`