Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Bachelorarbeit

Christian Bargmann

Ausgewählte Testkonzepte für Microservice-Architekturen

Fakultät Technik und Informatik
Studiendepartment Informatik

Faculty of Engineering and Computer Science
Department of Computer Science

Christian Bargmann

Ausgewählte Testkonzepte für Microservice-Architekturen

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung

im Studiengang Bachelor of Science Wirtschaftsinformatik
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer: Prof. Dr. Stefan Sarstedt
Zweitgutachter: Prof. Dr. Ulrike Steffens

Eingereicht am: 3. August 2018

Christian Bargmann

Thema der Arbeit
Ausgewählte Testkonzepte für Microservice-Architekturen

Stichworte
Softwareentwicklung, Softwarearchitektur, Softwaretests, Qualitätssicherung, Microservices, Continuous Deployment, Cloud Computing

Kurzzusammenfassung
Das Testen von lose gekoppelten Microservices, die als verteiltes System einen Dienst anbieten und über standardisierte Schnittstellen miteinander kommunizieren, stellt Tester und Entwickler vor neue Herausforderungen. Diese Bachelorarbeit soll einen Weg aufzeigen, wie auf Grundlage von ausgewählten Testkonzepten, qualitativ hochwertige, performante und fehlerfreie Services als Teil eines Microservice-Ökosystems entwickelt werden können. Dazu wurden bestehende Testkonzepte miteinander verglichen und bewertet. Darauf basierend wird eine Implementierung eines erarbeiteten Testkonzeptes am Beispiel eines Versuchssystems vorgestellt.

Christian Bargmann

Title of the paper
Test Concepts For Microservice Architectures

Keywords
Software Development, Software Architecture, Software Tests, Quality Assurance, Microservices, Continuous Deployment, Cloud Computing

Abstract
Testing loosely coupled microservices, which provide a single application in form of a distributed system and communicate with each other via standardized interfaces, presents new challenges for testers and developers. This bachelor thesis should demonstrate a way to show how high-quality, high-performance and error-free services can be developed as part of a microservice ecosystem on the basis of selected test concepts. Existing test concepts were compared and evaluated. Based on the results, an implementation of a developed test concept is presented and exemplarily evaluated with a microservice system.

Danksagung

Danke an meinen Betreuer *Prof. Dr. Stefan Sarstedt* für die Motivation und Unterstützung während der Bearbeitungszeit. Ein Dankeschön geht ebenfalls an *Prof. Dr. Ulrike Steffens*. Eure Begeisterung für Software-Engineering und Software-Architektur hat meine Orientierung im Studium sehr geprägt und ohne das HAWAI-Projekt wäre diese Arbeit nicht zustande gekommen.

Danke an meine *Familie*, die mir das Studieren ermöglichte und mich in meinen Entscheidungen immer unterstützte. Besonders an *Uwe*, der sich vor allem im ersten Semester sehr viel Zeit für meine offenen Fragen genommen hat und stets für eine Diskussion offen war.

Danke an *Gantsatsral*, die während des Schreibens dieser Arbeit in Gedanken bei mir war und auf viel gemeinsame Zeit verzichten musste.

Danke an *Gerhard Simon*, *Andrea Miller* und das Team der Abteilung für Informationstechnologie der Hamburg Messe und Congress GmbH, die mir nach meiner Berufsausbildung ermöglichten, das Studium an der HAW zu beginnen. Durch euch habe ich in meiner Studienzeit den Bezug zur Praxis nie verloren und ihr habt viel zu meiner Freude am Studium beigetragen.

Und zu guter Letzt: Danke an meine Kommilitonen. In unseren gemeinsamen Semestern existierte immer ein fester Zusammenhalt untereinander und ein reger Austausch, der zum Erfolg des Studiums beigetragen hat und durch den ich mein Bachelorstudium in sehr schöner Erinnerung behalten werde.

Inhaltsverzeichnis

Tabellenverzeichnis

Abbildungsverzeichnis

Listings

1 Einleitung

Microservice-basierte Systeme haben sich als Architekturstil der Wahl für die Entwicklung und den Betrieb von Software für „Cloud Computing“ durchgesetzt. Unternehmen wie Zalando, Soundcloud, Netflix und Amazon haben ihre Applikationen umstrukturiert, um von den Vorteilen dieses Architekturstils zu profitieren (vgl. Fowler, 2014; Kolesnikov, 2018).
Microservices sind ein starkes Modularisierungskonzept, das ein komplexes Softwaresystem in kleine, unabhängige Services unterteilt, die in ihrer Gesamtheit einen Dienst zur Verfügung stellen. Dabei hat das Konzept Auswirkungen auf die Organisation des Entwicklungsprozesses von Software. Durch die Modularisierung ergeben sich Vorteile für die Anwendung hinsichtlich Wartbarkeit, Austauschbarkeit und Skalierbarkeit. Außerdem unterstützen sie die nachhaltige Softwareentwicklung und eine technologieunabhängige, agile Arbeitsweise einzelner Entwicklerteams (vgl. Wolff, 2015, S.3 f.). Über die Möglichkeiten, aber auch über die Herausforderungen und Schwierigkeiten, die mit diesem Architekturstil einhergehen, wird in den letzten Jahren sehr viel diskutiert.

Ein wichtiger Aspekt bei der Entwicklung eines Softwaresystems ist das Testen der Anwendung. Das Testen einer Software ist notwendig, um die Mängel und Fehler aufzudecken, die während der Entwicklungsphase aufgetreten sind (vgl. Homès, 2013, S.3) und wird bei Microservice-basierten Systemen im Vergleich zu anderen Aspekten weniger ausführlich dargestellt.

In dieser Arbeit sollen deshalb bestehende Testkonzepte und Ansätze für das Testen von Microservice-Architekturen untersucht, zusammengefasst und bewertet werden. Auf dieser Grundlage soll ein Testkonzept für eine beispielhafte Microservice-Architektur entwickelt und vorgestellt werden. Das Testkonzept soll anwendbar sein auf vergleichbare Microservice-basierte Systeme und eine Möglichkeit aufzeigen, zuverlässige, robuste und qualitativ hochwertige Microservices zu entwicken und als Gesamtsystem zu betreiben.

Das erste Kapitel gibt einen Überblick über Motivation, Zielsetzung und Aufbau dieser Arbeit.

1.1 Motivation

Das Interesse für die Themenstellung entstand durch die Teilnahme an dem Studienfach „*Software Engineering in der Cloud*“ im Rahmen des Bachelorstudiums an der Hochschule für Angewandte Wissenschaften Hamburg, in dem eine Anwendung mit einer Microservice-Architektur entwickelt wurde.

Die Teilnehmer der Projektgruppe hatten Erfahrungen in der Entwicklung von monolithischen Softwaresystemen, allerdings waren Microservices für die Meisten ein neuer Ansatz für die Entwicklung von Software. Den Teilnehmern wurden zunächst das grundlegende Konzept von Microservices und die Entwicklung von Cloud-Native Applikationen vermittelt und im Anschluss wurde ein kleines System mit wenigen Services entwickelt. Während es Entwicklungsprozesses wurden die Herausforderungen einer Microservice-Architektur spürbar, vor allem der Aspekt des Testens ließ viele Fragen nach Lösungsansätzen offen. Der Abschluss des Projektes erbrachte kein Produkt, das veröffentlicht werden konnte. Viele Komponenten waren unzureichend mit Testfällen abgedeckt und es mangelte an einem Konzept für das systematische Testen des Anwendungssystems.

Aufgrund der im Projekt gewonnenen Erkenntnisse und ähnlicher Erfahrungen in Forschungsprojekten an der Hochschule für Angewandte Wissenschaften ergab sich die Idee, bestehende Testkonzepte für Microservice-Architekturen genauer zu untersuchen und einen Vorschlag für ein Testkonzept zu machen, das auf zukünftige Microservice-basierte Anwendungen angewendet werden kann.

1.2 Zielsetzung

Diese Arbeit soll bestehende Testkonzepte und Ansätze für das Testen von Microservice-Architekturen untersuchen, zusammenfassen und bewerten. Auf Grundlage bereits bestehender Testkonzepte für Microservice-basierte Systeme, soll ein individuelles Testkonzept für ein Versuchssystem entwickelt und beispielhaft implementiert werden.
Um dieses Testkonzept zu entwerfen, werden zunächst die Grundlagen in Bezug auf Microservices, Softwaretests und Testautomatisierung erläutert. Dabei werden die Schwierigkeiten und Anforderungen, die sich im Zusammenhang mit dem Testen einer solchen Architektur ergeben, dargestellt. Darauf aufbauend werden verschiedene Testkonzepte untersucht und bewertet. Auf Grundlage der Bewertung wird ein Testkonzept für eine Microservice-Architektur entwickelt und vorgestellt.

1.3 Aufbau der Arbeit

Diese Arbeit ist in fünf Kapitel gegliedert.
Kapitel 1, Einleitung führt in die Thematik dieser Arbeit ein und gibt einen Überblick über die Themenstellung.
Kapitel 2, Grundlagen erläutert die Grundlagen zu Microservices, dem Testen von Software und nennt Möglichkeiten zur Testautomatisierung.
Kapitel 3, Testen von Microservices untersucht, vergleicht und bewertet bestehende Testkonzepte für Microservice-Architekturen. Der Vergleich und die Bewertung bilden die Basis für die Entwicklung eines individuellen Testkonzeptes für das Versuchssystem.
Kapitel 4, Entwicklung und Implementierung eines Testkonzeptes erläutert, wie auf Grundlage der vorgestellten Testkonzepte ein individuelles Testkonzept für ein Versuchssystem entwickelt und beispielhaft implementiert wird. Anhand einer konkreten Deployment-Pipeline wird erläutert, wie sich systematisch und automatisiert Testmethoden in einer Microservice-Architektur anwenden lassen.
Kapitel 5, Fazit fasst die Inhalte der Arbeit zusammen und reflektiert das Ergebnis. Der Ausblick erläutert weiterführende Fragestellungen und schließt die Arbeit ab.

2 Grundlagen

In diesem Kapitel werden die Grundlagen zu Microservice-basierten Systemen, deren Bereitstellung sowie Grundbegriffe für das Testen von Anwendungssoftware erläutert.

2.1 Begriffserklärung „Microservices“

Microservices sind ein Architektursstil aus der Softwareentwicklung, welcher in den letzten Jahren zunehmend an Bedeutung gewonnen hat. Der Begriff *Microservices* wird oft mehrdeutig verwendet. Zum einen beschreibt der Begriff eine Architektur für Anwendungssysteme, zum anderen die einzelnen Komponenten dieser Architektur selbst. Mit der Verwendung des Begriffs Microservices wird auch eine Reihe von Lösungsansätzen für die Organisation des Softwareentwicklungsprozesses und die Bewältigung aktueller Probleme und Herausforderungen mit Software assoziiert. Microservices sind auch ein Modularisierungskonzept, dass dazu dient komplexe Systeme in kleinere Komponenten zu zerlegen, um ein System einfacher zu erstellen, zu verstehen und zu entwicklen (vgl. Wolff, 2015, S.2 f.). Zusammenfassend lässt sich der Begriff in die drei folgenden Kategorien einteilen:

1. *Architekturstil* — Microservices sind ein Architekturstil. Eine Architektur beschreibt die innere und äußere Struktur eines Softwaresystems (vgl. Hohmann, 2003, S.4 ff.).

2. *Komponenten eines Softwaresystems* — Microservices sind Komponenten. Sie sind Softwareelemente, welche unabhängig von anderer Software verwendet und ohne Änderungen mit anderen Komponenten zusammengesetzt werden können (vgl. George T. Heinemann, 2006, S.34 f.).

3. *Modularisierungskonzept* — Microservices sind ein Modularisierungskonzept. Sie beschreiben die Organisation eines Systems und eine Vorgehensweise für nachhaltige Softwareentwicklung (vgl. Wolff, 2015, S.4 f.)

2.2 Ein Microservice

Ein *Microservice* ist eine kleine Anwendung, die unabhängig eingesetzt, unabhängig skaliert und unabhängig getestet werden kann und eine einzige Verantwortung besitzt (vgl. Thones, 2015, S.116).

Verantwortung bedeutet hier, dass es nie mehr als einen Grund geben sollte, einen Service zu ändern oder zu ersetzen. Des Weiteren sollte ein Microservice eine einzige, eigenständige Aufgabe innerhalb des Gesamtsystems übernehmen und gleichzeitig einfach zu verstehen sein. Letzteres besagt auch die Entwurfsrichtlinie des *Single-Responsibility-Prinzips* aus der objektorientierten Programmierung, die sich auf das Konzept von Microservices anwenden lässt (vgl. Martin, 2005).

Der Begriff „Microservice“ macht dabei indirekt eine Aussage über den Umfang eines solchen Services. Als Richtlinie lässt sich festhalten, dass der Code der Aufgabe eines Services vom Entwickler verstanden werden sollte, ohne dabei viel Komplexität aufzuweisen und nicht über ein paar tausend Zeilen Quellcode zur Erfüllung dieser Aufgabe hinauszugehen (vgl. Thones, 2015, S.117). Hierüber lässt sich anhand der reinen Zahl der Codezeilen schwierig eine Aussage treffen. Die einzelnen Microservices sind nach Möglichkeit loose gekoppelt und Abhängigkeiten zwischen einzelnen Services sind zu vermeiden. Durch diese loose Kopplung wirken sich Änderungen an Services nicht unmittelbar auf das gesamte System aus, sondern in erster Linie auf den Service selbst (vgl. Newman, 2016, S.5 f.).

Bei Microservices handelt es sich um ein verteiltes System. Die Komponenten rufen sich nicht gegenseitig mit Methoden- oder Funktionsaufrufen auf Sprachebene auf, sondern die einzelnen Services kommunizieren untereinander über standardisierte Schnittstellen auf Netzwerkebene (vgl. Wolff, 2015, S.4). Abhängig von der Art des Services müssen diese bei der Interaktion ein prozessinternes Kommunikationsprotokoll wie zum Beispiel HTTP oder AMQP verwenden. Savchenko u. a. (2015) beschreiben vier wesentliche Funktionalitäten von Microservices.

- **Open Interface** — Ein Microservice soll eine öffentliche Beschreibung der Service-Schnittstelle und des Nachrichtenformates dem Klienten gegenüber anbieten.
- **Specialization** — Ein Microservice kapselt einen eigenständigen Teil der Geschäftslogik des Gesamtsystems.

- **Containerization** — Ein Microservice soll unabhängig von seiner Laufzeitumgebung sein, wofür sich Container-Technologien durchgesetzt haben.
- **Autonomy** — Ein Microservice soll sich unabhängig von anderen Services entwickeln, testen, ausführen, bewegen und beenden lassen. Dies ist nur mit Hilfe von Continuous Integration (siehe Abschn. 2.4.1) und Automatisierung möglich.

2.3 Software-Architektur

Software-Architektur beschreibt die grundsätzliche Organisation eines System, verkörpert durch dessen Komponenten, deren Beziehungen zueinander und zur Umgebung sowie die Prinzipien, die für seinen Entwurf und seine Evolution gelten (vgl. IEEE, 2000, S.3).

> „Software architecture deals with abstraction, with decomposition and composition, with style and esthetics." (Kruchten, 1995)

Eine Softwarearchitektur muss die Komponenten eines Systems und deren Merkmale beschreiben, sowie die Beziehungen der Komponenten zueinander charakterisieren. Dabei lässt sich Software-Architektur in zwei Ebenen unterteilten. Die *Makro-Architektur* eines Softwaresystems spezifiziert das globale System, trennt die einzelnen Komponenten voneinander ab und gibt an, wie diese zueinander in Beziehung stehen. Die *Mikro-Architektur* spezifiziert hingegen die Organisation und die Beziehungen der Bausteine innerhalb von Komponenten.
Nach Kruchten (1995) lässt sich Software-Architektur durch fünf verschiedene Sichtweisen beschreiben:

- **Kontextsicht** — Die Kontextsicht betrachtet die Funktionalität des Systems aus Sicht des Endbenutzers. Sie betrachtet das System aus einer Vogelperspektive und bezieht Nachbarsysteme mit ein.
- **Bausteinsicht** — Die Bausteinsicht beschreibt die grundlegende, statische Struktur des Systems. Es werde die einzelnen Bausteine und die Beziehungen zwischen diesen betrachtet.
- **Laufzeitsicht** — Die Laufzeitsicht betrachtet die dynamischen Aspekte des Systems und erläutert, wie die einzelnen Bausteine des Systems zur Laufzeit miteinander arbeiten.
- **Verteilungssicht** — Die Verteilungssicht beschreibt die Umgebung in der das System ausgeführt wird.

- **Szenariensicht** — Zusätzlich gibt es noch die Szenariensicht, welche wichtige Anwendungsfälle des Systems darstellen soll.

2.3.1 Monolithische Architektur

Der Begriff *Monolith* entstammt dem altgriechischen und beschreibt einen aus einer einzigen Gesteinsart bestehenden, großen, meist bearbeiteten Steinblock. Dieser Metapher bedient sich die *monolithische Softwarearchitektur* und beschreibt damit eine Menge an Softwareprodukten, die architektonisch aus einer einzigen, unteilbaren Einheit bestehen und die im Laufe des Entwicklungsprozesses zu einer gewissen Größe herangewachsen sind.

> „It's a single-tiered software application in which the user interface and data access code are combined into a single program on a single platform. It's also an application that runs multiple components in the same process, on the same system." (Nova, 2018)

Folgt man der Definition von Kris Nova, so können Software-Monolithen sehr wohl modularisiert sein, jedoch sind die einzelnen Komponenten nicht unabhängig voneinander. Der Begriff „Unabhängigkeit " ist dabei sehr offen formuliert. Die einzelnen Komponenten eines Monolithen können fachlich unabhängig voneinander sein. Da sie alle in einem gemeinsamen Prozess und auf einem gemeinsamen System laufen, gelten sie aus Sicht einer Microservice-Architektur aber nicht als unabhängig voneinander. Allerdings werden auch Microservice-basierte Systeme, die aufgrund eines schlechten Designs die Eigenschaften eines Monolithen aufweisen, als *monolithisch* bezeichnet. Wolff definiert den Deployment-Monolithen als großes Softwaresystem, das nur als Ganzes auf einmal deployed werden kann. Das hat zu Folge, dass bei Änderungen am Softwaresystem die gesamte Anwendung neu kompiliert und deployed werden muss (vgl. Wolff, 2015, S.3).

2.3.2 Microservice-Architektur

Eine *Microservice-Architektur* ist ein Ansatz zur Entwicklung einer einzelnen Anwendung als eine Zusammenstellung von kleinen Diensten, die jeweils in einem eigenen Prozess laufen und über leichtgewichtige, standardisierte Schnittstellen miteinander kommunizieren. Diese Dienste kapseln einen eigenständigen Teil der Business-Logik und werden vollautomatisch bereitgestellt. (vgl. Martin Fowler, 2014).
Das Konzept von Microservices beruht dabei auf Idee der „*Smart Endpoints und Dumb Pipes*" aus der UNIX-Welt. Um die Komplexität einer Software zu beherrschen gilt es, die globale

Komplexität der Anwendung möglichst gering zu halten. Dies gelingt durch die Zerlegung in einfache Bausteine mit einheitlichen, wohl definierten Schnittstellen, sodass sich Probleme lokal auf einen einzelnen Baustein auswirken und nicht das gesamte System gefährden (vgl. Raymond, 2003, S.107).
Eine Microservice-Architektur ist ein verteiltes System und bringt im Vergleich zu monolithischen Anwendungen Vorteile und Herausforderungen mit sich, welche in den Abschnitten 2.3.2.2 und 2.3.2.3 im Detail dargestellt werden.

2.3.2.1 Kommunikation

Ein besonderer Schwerpunkt bei Microservices liegt in der Art und Weise der Kommunikation zwischen den einzelnen Services. Anders als bei monolithischen Anwendungen, in denen alle Komponenten in einem Prozess ausgeführt werden und über Funktions- oder Methodenaufrufe miteinander kommunizieren können, laufen einzelne Microservices in jeweils eigenen Prozessen. Die Kommunikation findet deshalb meistens auf Netzwerkebene statt. Für die Netzwerkkommunikation beschreibt Newman zwei verschiedene Verfahren (vgl. Newman, 2016, S.70), die bei Microservices Anwendung finden:

- **Request / Response** — Das Request-Response Verfahren lässt sich der *synchronen Kommunikation* zuordnen. Bei der synchronen Kommunikation wird ein Aufruf an einen externen Service gesendet und gewartet, bis die Transaktion entweder erfolgreich oder erfolglos ausgeführt wurde. Der Aufrufer erhält somit direktes Feedback und der aufgerufene Service kann in einem bestimmten Umfang den Prozessablauf steuern.
- **Event-Based** — Das Event-Based Verfahren lässt sich der *asynchronen Kommunikation* zuordnen. Die asynchrone Kommunikation eignet sich für zeitaufwendige Aufgaben, bei denen ein längeres Aufrechterhalten der Verbindung zwischen Aufrufer und einem externen Service schwierig sein kann. Weil ein auf Antwort wartender Service den Programmablauf verzögert, eignet sich die asynchrone Kommunikation für Anwendungen mit geringer Latenzzeit. Beim Event-Based Verfahren sendet ein Microservice einen Nachrichtentyp als Reaktion auf ein bestimmtes Ereignis. Andere Microservices können diese Nachrichtentypen abonnieren und anschließend wiederum entsprechende Aktionen durchführen.

Representational State Transfer (REST) *Representational State Transfer* ist ein durch das World Wide Web inspirierter Architekturstil und eine Möglichkeit, um Schnittstellen für Microservices zu entwerfen. REST nutzt in der Regel die in HTTP enthaltenen Request-Methoden

und Statuscodes aus und erlaubt die Verwendung unterschiedlicher Standardtextformate wie XML oder JSON. (vgl. Newman, 2016, S.83). Gegenüber vielen Architekturstilen die operationsbasiert sind, steht bei REST die Ressource im Vordergrund. Eine Ressource ist dabei eine konkrete Entität wie beispielsweise ein Kunde oder eine Bestellung. Diese Ressourcen können durch unterschiedliche Repräsentationen dargestellt werden. Die Interaktion mit Schnittstellen die auf REST basieren, sollte überwiegend zustandslos sein, was bedeutet, dass ein externer Service sich den Anwendungszustand eines Aufrufers nicht merken muss. Der Zustand einer Ressource wiederum liegt im Verantwortungsbereich des externen Services. REST selbst besitzt keinen Standard, definiert jedoch eine Menge an Prinzipien, die ein bestimmten Verhalten der Operationen einer Schnittstelle auf eine Ressource fordern (vgl. Fielding, 2000, S.75).

Messaging *Messaging* ist ein in der Regel asynchrones Kommunikationsverfahren, das auf dem Verschicken von Nachrichten beruht. Messaging kann auch synchron implementiert werden, in dem der Aufrufer auf die Antwort des externen Services wartet und blockiert. Es wird oft für ereignisgesteuerte Kommunikation verwendet und bietet viele Vorteile, um den Anforderungen in einer Microservice-Architektur gerecht zu werden. Durch das Nutzen von asynchroner Kommunikation können Nachrichten sehr gut mit Netzwerklatenzen zwischen Services umgehen und überleben auch den Ausfall eines Netzwerkes, weil sie in den jeweiligen Services zwischengespeichert werden. Microservices als „Cloud-Native“ Anwendungen sollten in der Lage sein, mit jeder Art von Misserfolg bei einem Serviceaufruf umgehen zu können. Messaging unterstützt diesen Ansatz und bei Fehlern die Bearbeitung neu initiieren. Einige Implementationen unterstützten Transaktionen und somit die Erhaltung der Konsistenz in verteilten Systemen. Messaging wird oft in Form eines zentralen, leichtgewichtigen Message-Brokers für die Verteilung der Nachrichten an die Services und die Durchführung von Transaktionen implementiert (vgl. Newman, 2016, S.87).

Remote Procedure Call (RPC) *Remote Procedure Call* ist ein Mechanismus zum Nachrichtenaustausch von verteilten Systemen und eine Technik für den Funktionsaufruf über Prozessgrenzen hinweg. Aus diesem Grund eignet sich RPC für die Kommunikation von Microservices. RPC basiert auf der synchronen Kommunikation, weshalb ein Aufrufer über die gesamte Kommunikationsdauer eine Verbindung zu einem externen Service halten muss. Der Aufrufer blockiert deshalb nach dem Senden einer Anfrage und führt seine Prozedur erst dann weiter aus, wenn eine Antwort angekommen ist. Der Aufrufer sendet die auszuführende Funktion, gemeinsam mit Funktionsargumenten an den externen Service. Dieser muss wiederum die angeforderte Funktion implementieren und führt diese mit den Funktionsargumenten des

Aufrufers aus. Die Antwort wird anschließend an den Aufrufer zurückgesendet. Viele Implementationen von RPC setzen Binärdaten als Nachrichtenformat ein. Einige Implementationen wie zum Beispiel SOAP, das HTTP verwendet, sind an bestimmte Netwerkprotokolle gebunden. Andere sind frei in der Wahl von Netzwerkprotokollen und können je nach Anwendungsfall die Funktionalitäten der Netzwerkprotokolle TCP oder UDP nutzen (vgl. Newman, 2016, S.75).

2.3.2.2 Vorteile

Die in diesem Abschnitt genannten Aspekte beziehen sich auf den Vergleich zwischen Microservices und den klassischen Software-Monolithen.
Durch die starke Modularisierung von Microservices können einzelne Services leichter ausgetauscht werden. Ein Service definiert eine explizite Schnittstelle, über welcher er mit anderen Services kommuniziert und seine Dienste anbietet. Wenn ein Service dieselbe Schnittstelle anbietet, kann ein Service einen anderen Service ersetzen. Der neue Service, der die Implementation der Schnittstelle erfüllt, kann eine völlig unterschiedliche Codebasis beinhalten oder sogar eine andere Technologie verwenden (vgl. Wolff, 2015, S.4). Durch die Modularisierung lassen sich neue Technologien leichter adaptieren und ausprobieren, ohne teure technische Fehlentscheidungen zu riskieren. Durch diese unabhängige Wahl der Technologie ergeben sich vor allem in größeren Softwareprojekten Freiheiten für einzelne Entwicklerteams und sie unterstützt die Anwendung von agilen Methoden und Arbeitsweisen. Ein weiter Vorteil ist die Robustheit einer Microservice-Architektur. Bei Ausfall eines Services ist nicht das komplette System betroffen, sondern lediglich einzelne Prozesse innerhalb der Anwendung. In einer monolithischen Anwendung hingehen ist das komplette System potentiell gefährdet. Zwar lässt sich dieser Ausfall durch horizontale Skalierung der monolithischen Anwendung vorbeugen, Microservices hingegen unterstützen darüber hinaus das Modellieren eines verteilten Systems, welches mit dem Ausfall einzelner Services umgehen kann und lediglich Funktionalitäten für die Dauer des Ausfalls einschränkt (vgl. Newman, 2016, S.5). Eine Microservice-Architektur erlaubt auch eine feingranulare Skalierung einzelner Services unter Last. Durch die Unabhängigkeit der einzelnen Microservices lassen sich diese bei Performanzproblemen unabhängig von anderen Services skalieren. In einem monolithischen System muss bei Performanzproblemen einer Komponente oft das komplette System mit allen Komponenten skaliert werden, also auch mit denen, die nicht von Performanzproblemen betroffen sind (vgl. Newman, 2016, S.5 f.).

Als weiterer Aspekt ist das Deployment von Microservices zu nennen. Die Unterteilung in einzelne Services erlaubt nicht nur eine unabhängige Entwicklung, sondern auch ein unabhängiges Deployment der Services. Dies erlaubt kürzere, risikoärmere Releasezyklen und unterstützt

die schnellere Auslieferung von neuen Features an die Benutzer. Auftretende Probleme lassen sich auf individuelle Services isolieren und ein Rollback lässt sich einfach durchführen (vgl. Newman, 2016, S.6).
Die Entscheidung für Microservices bringt viele Vorteile mit sich. Allerdings ist es wichtig, gleichzeitig ein Bewusstsein für die Herausforderungen des Architekturstils zu entwickeln.

2.3.2.3 Herausforderungen

Obwohl Microservices viele positive Eigenschaften besitzen, bringen sie neue Herausforderungen mit sich. Microservices sind ein verteiltes System und bringen auch sämtliche Probleme mit sich, die ein verteiltes System aufweist.

Der Einsatz von Microservices erfordert umfangreiche Expertise, da für den Betrieb ein hohes Maß an Automation, ein umfangreiches Monitoring und Wissen im Umgang mit der Orchestrierung von Services notwendig ist. Der Betrieb wird mit steigender Anzahl an Services komplexer, weil mehr Dienste gleichzeitig verwaltet werden müssen und die wachsende Anzahl an Kommunikationswegen in der Anwendung mehr Potenzial für Ausfälle liefert (vgl. Wolff, 2015, S.6). Die Beziehungen der einzelnen Microservices zueinander bildet die Architektur des Gesamtsystems. Microservices verstecken ihre Beziehungen zueinander und ohne geeignete Lösungen können die Serviceaufrufe nicht nachvollzogen werden. Dies erschwert die Architekturarbeit. Auch ein geeigneter Komponentenschnitt ist wichtig, da Services ansonsten zwar technisch unabhängig, allerdings fachlich stark gekoppelt sein können, wodurch die Vorteile einer Microservice-Architektur verbaut werden. Auch die Versionierung von Services und das Verwalten von Abhängigkeiten zwischen einzelnen Services kann eine Herausforderung darstellen. Als eigenständige Prozesse oder Programme werden diese bei Änderungen auch unabhängig voneinander versioniert. Services müssen definieren, welche Version von aufzurufenden Schnittstellen sie voraussetzen (vgl. Kerr, 2018). Da sich die Beantwortung einer Anfrage in einer Microservice-Architektur durch die Kommunikation mehrerer Services ergibt, wird das Halten eines Zustandes entlang des Kommunikationspfades schwierig (vgl. Kerr, 2018). Deshalb eignen sich Microservices nur bedingt für Anforderungen, die das Halten eines Zustands voraussetzen. Viel Komplexität wird von der eigentlichen Anwendung in die Kommunikation mit anderen Services verlagert. Automatische Erkennung von Diensten in einem Rechnernetz mithilfe von Service Discovery wird notwendig, um die Kommunikation von aktiven Services zu gewährleisten. Aufrufe zwischen einzelnen Services können aufgrund von Netzwerkprobleme fehlschlagen und müssen behandelt werden. Auch sind Aufrufe über ein Netzwerk deutlich langsamer als Aufrufe innerhalb eines Prozesses (vgl. Wolff, 2015, S.6).

Das Testen eines solchen verteilten Systems ist eine große Herausforderung. Bei Auftreten eines Fehlers lässt sich der Fehlerzustand des Gesamtsystems schwer reproduzieren, wodurch eine Fehlerbehandlung äußerst schwierig wird. Durch die Kommunikation der Services über das Netzwerk ergeben sich neue Testgegenstände, die bei monolithischen Systemen weniger stark ins Gewicht fallen. Durch die Bereitstellung eines Dienstes durch Kommunikation mehrerer Services untereinander, werden auch Testmethoden (siehe Abschn. 2.5.3) wie zum Beispiel die Durchführung von Integrationstests schwieriger. Diese Thematik wird in Kapitel 3 weiter beleuchtet.

2.4 Deployment von Microservices

Dieser Abschnitt führt in das Deployment von Microservices ein. Dazu wird als Erstes das Konzept von Continuous Integration, Delivery und Deployment erläutert. Darauf aufbauend wird Gitlab als CI-Tool und Pipeline vorgestellt. Anschließend wird auf Docker als Container-Technologie und Kubernetes als Tool für Container-Orchestrierung eingegangen.

2.4.1 Continuous Integration

Unter *Continuous Integration*, oder kurz CI, wird das kontinuierliche Integrieren, Bauen und Testen von Code innerhalb der Entwicklungsumgebung verstanden mit dem Ziel, bei Änderungen an der Codebasis einer Anwendung möglichst schnell Integrationsfehler aufzudecken. Dies erlaubt Entwicklerteams, zusammenhängende Software schneller zu entwickeln (vgl. Farcic, 2016, S.11 f.). Microservice-basierte Systeme bestehen in der Regel aus einer einzigen Codebasis, die in einem Versionskontrollsystem verwaltet wird. Die Codebasis ist in entweder einem oder einer Menge von Sourcecode-Repositories abgelegt, die einem Projekt zuordenbar sind. Die Codebasis für eine Anwendung wird verwendet, um eine beliebige Anzahl von unveränderlichen Versionen zu erzeugen, die für verschiedene Laufzeitumgebungen bestimmt sind (vgl. Hoffman, 2016, S.1 ff.). Entwickler arbeiten an neuen Features auf verschiedenen Branches. Sobald ein Arbeitsfortschritt erzielt worden ist, wird der aktuelle Stand des Codes in den Master-Branch eines Repositories integriert.

CI-Tools überwachen dieses Repository. Bei Änderungen am Code wird eine Pipeline gestartet. Eine Pipeline ist eine Menge von automatisierten Arbeitsschritten, die entweder parallel oder sequentiell auf den ausgecheckten oder geklonten Code des Repositories angewendet werden und die versuchen zu verifizieren, dass der veränderte Code sich wie erwartet verhält. Das Ergebnis nach dem Durchlaufen der Pipeline ist ein Grundvertrauen in den neuen Code,

welcher dann für weitere zeitintensivere, manuelle Tests an die Qualitätssicherung gegeben werden kann (vgl. Farcic, 2016, S.13).

2.4.2 Continuous Delivery und Deployment

Die Pipeline bei *Continuous Delivery* unterscheidet sich in den meisten Fällen nicht von der Continous Integration Pipeline. Der einzige Unterschied besteht im Vertrauen gegenüber dem Paket oder Artefakt bei erfolgreichem Durchlaufen der Pipeline. Continuous Integration vertraut auf manuelle Tests nach erfolgreichem Durchlaufen der Pipeline, bevor ein neuer Releasekandidat in die Produktion gegeben werden kann. Continuous Delivery geht einen Schritt weiter und folgt der Annahme, dass jedes erfolgreiche Durchlaufen der Pipeline am Ende ein Paket oder Artefakt bereitstellt, das sofort als Releasekandidat in Produktion gegeben werden kann (vgl. Farcic, 2016, S.19 ff.).

Continuous Deployment geht noch einen Schritt weiter und automatisiert das Deployment in die Produktivumgebung bei jedem erfolgreichen Durchlaufen der Pipeline. Die Pipeline startet mit einem Commit von Code in das vom CI-Tool überwachte Repository und endet mit einer Anwendung oder einem Service der in Produktion geht. Der gesamte Prozess ist dabei automatisiert und erfordert kein manuelles Eingreifen (vgl. Humble, 2010, S.266).

Im Unterschied zu Continuous Delivery und Continuous Deployment benötigt Continuous Integration nicht unbedingt das Testen der Anwendung in Produktion. Weil bei Continuous Delivery und Continuous Deployment kein manuelles Eingreifen vorgesehen ist, wird die Durchführung von (überwiegend) Integrationstests zu einem notwendigen Mittel, da sichergestellt werden muss, dass jede Anwendung oder Service der in Produktion geht, sich so verhält wie erwartet (vgl. Farcic, 2016, S.22).

2.4.3 Technologien

In diesem Absatz werden verschiedene Technologien, die im Rahmen dieser Arbeit für das Deployment von Microservices relevant sind, kurz vorgestellt. Zuerst wird auf Gitlab als CI/CD Tool eingegangen. Anschließend folgt eine kurze Erläuterung von Docker als Containertechnologie und Kubernetes als Plattform für Containerorchestrierung.

2.4.3.1 Gitlab

Gitlab ist ein webbasiertes Tool zur Versionsverwaltung von Softwareprojekten auf Basis von Git. Gitlab wird derzeit in zwei Versionen angeboten, einer Enterprise Edition, die zusätzliche Funktionen für den Einsatz in Unternehmen anbietet und einer Community Edition, die als Open-Source-Software unter der MIT-Lizenz entwickelt wird. Das Tool bietet Funktionen zum Bugtracking, Projektmanagement und der Kommunikation innerhalb von Softwareprojekten an. Mit *Gitlab CI* gibt es außerdem ein System für Continuous Integration (vgl. Gitlab, 2018a).

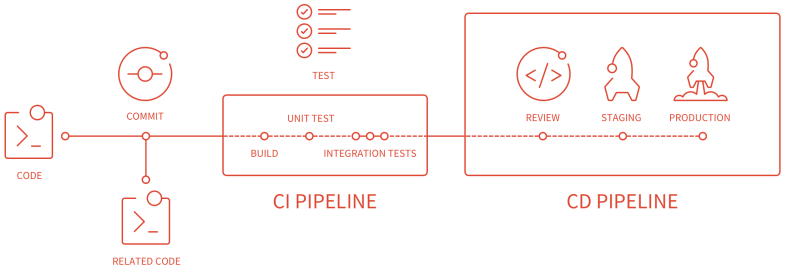

Abbildung 2.1: Funktionsweise der Gitlab CI Pipeline (vgl. Gitlab, 2018b)

Gitlab CI besteht im wesentlichen aus zwei Komponenten. Gitlab als CI-Tool überwacht Repositories auf Änderungen und startet bei Änderungen die Pipeline. Ein sogenannter *Runner* übernimmt die Ausführung der definierten Arbeitsschritte. Ein Runner kann dabei eine reelle oder virtuelle Maschine, ein Container oder ein Cluster aus mehreren Containern sein (siehe Abschn. 2.4.3.2 und 2.4.3.3).

In einer YML-Datei mit dem Namen *.gitlab-ci.yml* werden die einzelnen Arbeitspakete, die entweder parallel oder sequentiell abgearbeitet werden, definiert. Diese Datei wird in das überwachte Repository eingecheckt und von Gitlab beim Erkennen von Änderungen als Anweisungen für die Deployment Pipeline eingelesen (vgl. Gitlab, 2018b). Über die Weboberfläche wird die Pipeline mit den einzelnen Arbeitsschritten und deren Status visualisiert und die Ergebnisse können eingesehen werden.

```
1  compile:
2  stage: build
3  image: golang:$GOLANG_VERSION
4  before_script:
5  - mkdir -p $GOPATH/src/$ROOT_URL/$CI_PROJECT_NAMESPACE
6  - cd $GOPATH/src/$ROOT_URL/$CI_PROJECT_NAMESPACE
7  - ln -s $CI_PROJECT_DIR
8  - cd $CI_PROJECT_NAME
9  script:
10 - CGO_ENABLED=0 GOOS=linux go build -a -installsuffix cgo .
11 artifacts:
12 name: go-binary
13 paths:
14 - go-binary
```

Listing 2.1: Kompilieren eines Go-Artefakts als Arbeitsschritt in der .gitlab-ci.yml

2.4.3.2 Docker

Docker ist ein Open-Source Projekt, mit dem sich einzelne Anwendungen oder Services in Containern isolieren lassen. Docker nutzt hierzu die Eigenschaften des Linux-Kernels um Ressourcen wie Speicher, Prozessor oder Netzwerk voneinander zu isolieren. Docker ist eine Lösung zur Nutzung von Container-Technologien für die Softwareentwicklung und -bereitstellung (vgl. Farcic, 2016, S.24).

Ein Software-Container ist ein isoliertes und unveränderbares Image, das einer Anwendung inklusive ihrer Konfiguration und Abhängigkeiten entspricht. Dieses Image ist in einem definierten und wiederverwendbaren Format abgespeichert. Der Begriff „Container“ ist eine Analogie zu Containern aus der Seefahrt. Dort haben Container die Aufgabe, Waren zu umschließen und zu transportieren. Der Inhalt wird dabei von dem Inhalt anderer Container im selben Hafen isoliert. Sie sind standardisiert, robust und können von einem Ort zum anderen transportiert werden. Der Hafen kümmert sich um die Umgebung des Containers, der Inhalt ist dabei nicht relevant. Es existiert eine klare Aufteilung von Verantwortlichkeiten, die sich auf Software-Container übertragen lässt (vgl. Nickoloff, 2016, S.7 f.).

Anders als bei einer virtuellen Maschine benötigen Container kein eigenes Betriebssystem und auch keinen eigenen Kernel, stattdessen nutzen sie den Kernel des Hostsystems und dessen Funktionen. Dadurch sind Container deutlich leichtgewichtiger als virtuelle Maschinen.

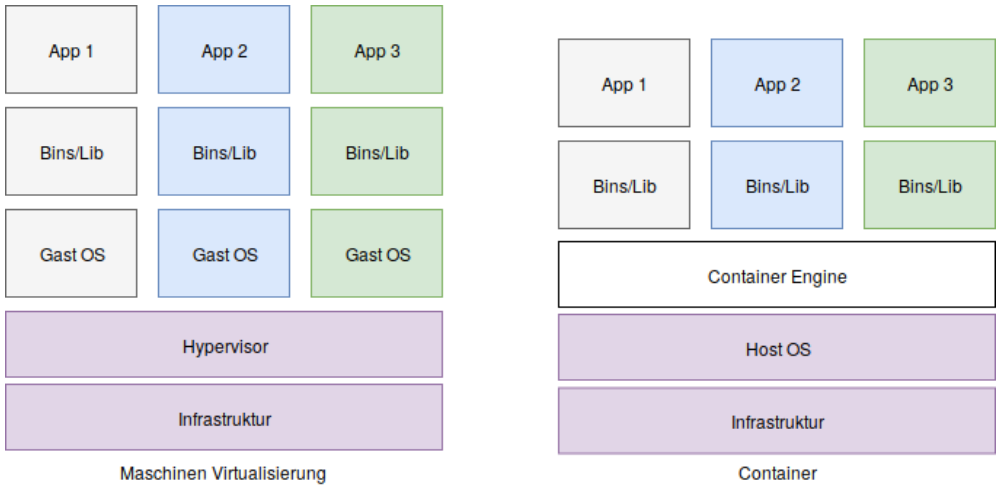

Abbildung 2.2: Vergleich der Ressourcennutzung von virtuellen Maschinen gegenüber Containern (vgl. Farcic, 2016, S.25)

Bei Microservices können viele Services gleichzeitig auf einem physikalischen Server laufen, weshalb der Ressourcengewinn durch die Nutzung von Containern statt virtuellen Maschinen zur Isolierung der Anwendungen signifikant ist (vgl. Nickoloff, 2016, S.5).

2.4.3.3 Kubernetes

Kubernetes ist eine Open-Source-Plattform zur Erstellung, Bereitstellung und Verwaltung verteilter Container-Anwendungen. Kubernetes wurde ursprünglich von Google entworfen und wird mittlerweile von der Cloud Native Computing Foundation unter der Apache-Lizenz 2.0 weiterentwickelt. Docker wird dabei standardmäßig als Container-Laufzeitumgebung unterstützt. Docker ermöglicht es, Anwendungen portabel zu machen und in unterschiedlichen Laufzeitumgebungen auszuführen. Während Docker eine Anwendung in einen Container verpackt und als solchen ausführbar macht, ist Kubernetes für den Betrieb der Container in Produktion zuständig.

Kubernetes unterstützt einen deklarativen Ansatz um Container-Anwendungen bei Bedarf zu starten oder zu stoppen, bietet automatisches Service Discovery an, unterstützt das Monitoring und skaliert Anwendungen manuell oder automatisch an Hand von Metriken. Als „Cloud-Native“ Plattform bietet Kubernetes auch Netzwerkfunktionen, Load-Balancing und Ressourcenbegrenzung an (vgl. Hightower u. a., 2017, S.9).

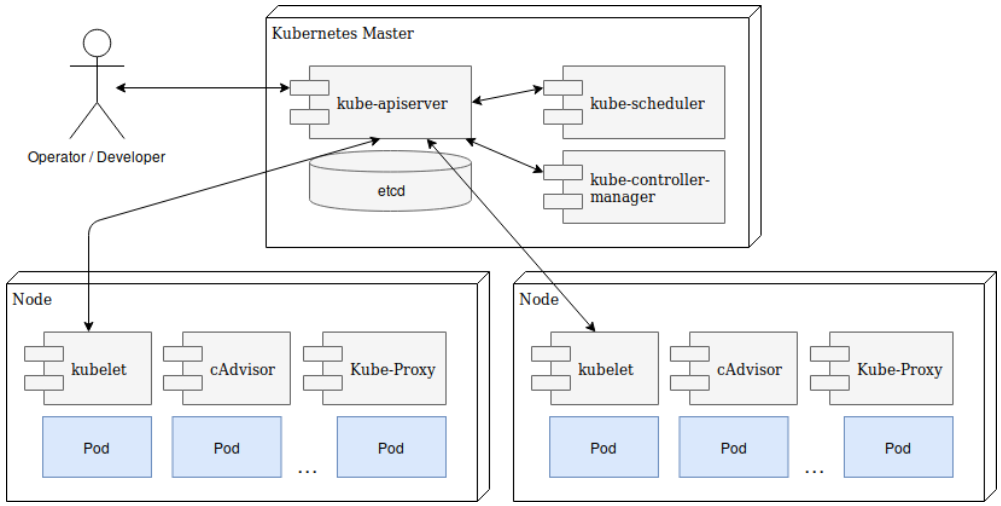

Abbildung 2.3: Übersicht über die Architektur von Kubernetes

Kubernetes besteht aus *Master- und Node-Komponenten*, die gemeinsam die Funktionalität des Clusters gewährleisten. Über die Master-Komponenten lassen sich globale Modifikationen am Zustand des Clusters durchführen. Die Node-Komponenten laufen auf jedem Knoten im Cluster und sind verantwortlich für die Verwaltung zugewiesener Ressourcen durch den Master und die Bereitstellung der Laufzeitumgebung.

Durch eine Reihe an Abstraktionen wird der aktuelle Zustand des Clusters durch Kubernetes beschrieben. Eine Komponente des Masters ist ein API-Server, durch den diese Abstraktionen in Form von Objekten erstellt, verändert und gelöscht werden können (vgl. Kubernetes.io, 2018).

Als kleinste Einheit werden in Kubernetes sogenannte *Pods* verwaltet. Pods sind eine Menge an zusammenhängenden Container-Anwendungen. Diese Pods werden als Arbeitsprozesse auf physikalische oder virtuelle Maschinen, den sogenannten Nodes, im Cluster verteilt. Pods besitzen einen Lebenszyklus und werden dynamisch erstellt und zerstört. Jedem Pod in einem Kubernetes-Cluster wird eine eigene IP-Adresse zugewiesen, die sich bedingt durch die endliche Lebensdauer eines Pods ändern kann. *Services* werden als Abstraktion in Kubernetes eingesetzt, um eine logische Menge an Pods und deren Zugriff zu steuern. Ein Service kann beispielsweise eine Microservice-Anwendung sein. Neben Pods und Services gibt es die Möglichkeit, Daten mithilfe von *Volume*-Objekten, die als Abstraktion für Objektspeicher dienen, an unterschiedlichen Speicherorten persistieren. *Namespaces* ermöglichen die logische Trennung von Ressourcen eines Clusters. Namespaces sind virtuelle Cluster, die auf einem

physikalischen Cluster betrieben werden können und so unterschiedliche Umgebungen für Benutzer, Teams oder Projekte zur Verfügung stellen.

Neben diesen Basisobjekten gibt es noch eine Menge an höheren Abstraktionen, die sogenannten *Controller*. Controller bauen auf den beschriebenen Basisobjekten auf und erweitern diese um weitere Funktionalitäten. Hier ist für das Verständnis der Arbeit das *Deployment*-Objekt zu nennen, das eine Definition für eine Menge an Pods und dessen Zustand repräsentiert. In einem Deployment-Objekt wird beschrieben, wie der gewünschte Zustand eines Pods und die Anzahl der Replikate auszusehen hat und der Deployment-Controller versucht, durch regelmäßigen Verlgeich zwischen Ist- und Sollzustand, diesen Zustand im Cluster zu halten (vgl. Kubernetes.io, 2018).

2.5 Grundlagen zum Testen

Die für diese Arbeit wichtigen Begriffe aus dem Bereich des *Testens von Software* werden in diesem Abschnitt vorgestellt. Dazu wird als Erstes auf verschiedene Grundbegriffe eingegangen und anschließend werden ausgewählte Testmethoden beschrieben. Die erläuterten Begriffe zum Thema Softwaretests orientieren sich dabei an den Definitionen der gemeinnützigen Zertifizierungsstelle für Softwaretester, dem *International Software Testing Qualifications Board* (ISTQB), auf welche sich auch das *German Testing Board* (GTB) stützt. Sofern andere Definitionen für die verwendeten Begriffe benutzt werden, wird an den entsprechenden Stellen in dieser Arbeit explizit darauf hingewiesen.

2.5.1 Testarten

Eine Möglichkeit der Klassifikation von Testmethoden sind Testarten. Die für diese Arbeit relevanten Testarten werden nachfolgend erläutert. Die Begriffe entsprechen der Definition des ISTQB.

2.5.1.1 Funktionale und Nicht-Funktionale Tests

Durch *funktionale Tests* wird überprüft, ob funktionale Anforderungen an das Testobjekt gemäß seiner Spezifikation erfüllt sind. Ein Beispiel hierfür ist das Abrufen oder Speichern einer konkreten Benutzerinformation. *Nicht-funktionale Tests* überprüfen, ob nicht-funktionale Anforderungen mit Blick auf die Qualität erfüllt sind (vgl. Andreas Spillner, 2012, S. 72 ff.). Nicht-funktionale Anforderungen können Sicherheit, Benutzbarkeit oder Performanz einer Anwendung sein. Die ISO/IEC 9126 Norm definiert eine Reihe von Qualitätsmerkmalen für

Software, die sich einerseits an der Gebrauchsqualität, andererseits an der äußeren und inneren Qualität einer Anwendung messen lässt. Durch sie können weitere nicht-funktionale Anforderungen an eine Anwendung spezifiziert werden.

2.5.1.2 Regressionstests

Der *Regressionstest* ist ein erneuter Test einer bereits getesteten Anwendung, nachdem diese verändert worden ist. Durch Regressionstests wird nachgewiesen, dass durch die vorgenommenen Änderungen keine neuen Fehlerzustände eingebaut, oder bereits abgedeckte Fehlerzustände wieder freigelegt wurden (vgl. Andreas Spillner, 2012, S. 77). Regressionstests können einen unterschiedlich definierten Umfang besitzen. Dieser Umfang kann von einem Fehlernachtest bis hin zu einem vollständigen Regressionstest, bei dem das komplette System erneut getestet wird, reichen.

2.5.2 Teststufen

Die Einteilung in *Teststufen* ist nicht nur eine zeitliche Einteilung verschiedener Testmethoden. Die Einteilung in Teststufen kann zum Beispiel durch die Art der Testobjekte, die Art der Teststrategie, dem Einsatz von Testwerkzeugen oder der Awendung unterschiedlicher Testverfahren vorgenommen werden. Bei der Einteilung in Teststufen wird versucht, mit möglichst kleinen Einheiten zu beginnen und schrittweise das gesamte System zu integrieren, bis es vollständig getestet ist und in Produktion gegeben werden kann.

Beim *Komponententest*, auch Unit-Test genannt, wird die kleinste Softwareeinheit getestet. Dies geschieht in Isolation zu den anderen Softwarebausteinen des Systems. Die kleinste Softwareeinheit ist abhängig von der gewählten Programmiersprache eines Projektes. In der im Rahmen dieser Arbeit verwendeten Programmiersprache Go wäre ein Unit-Test das Testen einer Funktion aus einem Package. Die Testbasis für den Komponententest ist die Spezifikation der Komponenten und der hierfür implementierte Quellcode. Das Ziel eines Komponententests ist es sicherzustellen, dass das Testobjekt die durch die Spezifikation geforderte Funktionalität korrekt und vollständig realisiert. Der Komponententest ist daher ein funktionaler Test. Neben der Funktionalität stehen weitere Aspekte wie die Robustheit, Effizienz und Wartbarkeit einer Komponente im Vordergrund (vgl. Andreas Spillner, 2012, S.44 ff.).

Als zweite Teststufe folgt der *Integrationstest*. Der Integrationstest testet, ob alle Einzelteile des Systems miteinander zusammenspielen. Dadurch sollen Fehlerzustände in Schnittstellen

und in der Zusammenarbeit zwischen integrierten Komponenten aufgedeckt werden. Die Komponenten werden schrittweise zu größeren Einheiten zusammengefasst und getestet. Ziele des Integrationstests können zum Beispiel das Senden von syntaktisch korrekten Daten, das korrekte Interpretieren der übergebenen Daten und das Aufdecken von Timing-, Durchsatz- oder Lastproblemen sein (vgl. Andreas Spillner, 2012, S.52 ff.).

Die dritte Teststufe ist der *Systemtest*. Beim Systemtest wird überprüft, ob die in der Spezifikation festgelegten Anforderungen an das Produkt erfüllt sind. Beim Systemtest wird daher das System als Ganzes betrachtet, weshalb die Testumgebung möglichst der Produktivumgebung gleichen sollte. Es werden sowohl funktionale Anforderungen als auch nicht-funktionale Anforderungen beim Systemtest berücksichtigt. Geeignete funktionale Systemtests können durch Anwendungsfälle, Geschäftsprozesse oder die spezifizierten Anforderungen an das System erstellt werden. Nicht-funktionale Systemtests wie beispielsweise Tests auf Datensicherheit, Robustheit, Lastverhalten oder Performanz können anhand der Qualitätsziele nach ISO 9126 oder ISO 25010 durchgeführt werden (vgl. Sneed u. a., 2011, S.9 ff.).

Der Vollständigkeit halber ist in diesem Zusammenhang der *Abnahmetest* als vierte Teststufe zu nennen, der in erster Linie auf die Gebrauchstauglichkeit und äußere Qualität des Systems abzielt (vgl. Winter u. a., 2016, S.64 ff.). Dieser ist im Kontext von Continuous Deployment und dieser Arbeit nicht weiter relevant.

2.5.3 Testmethoden

Auf konkrete Testmethoden, die in dieser Arbeit verwendet werden, wird in diesem Abschnitt eingegangen.

2.5.3.1 Statische Code-Analyse

Bei der *statischen Analyse* werden die Testobjekte nicht ausgeführt. Eine statische Analyse ist nur mit geeigneter Werkzeugunterstützung sinnvoll und hat das Ziel, vorhandene Fehlerzustände oder fehlerträchtige Stellen in einem Dokument aufzudecken. Diese Dokumente müssen vor der der Durchführung einer statischen Analyse eine formale Struktur besitzen. Compilerbasierte Sprachen wie zum Beispiel Go führen vor dem eigentlichen Kompilieren eine statische Analyse des Programmtextes durch, um Syntaxfehler zu entdecken. Neben Compilern gibt es eine Menge an Werkzeugen, die für die Durchführung weiterer Analysen verwendet werden. Durch statische Analysen können neben Syntaxverletzungen auch Abweichungen von

Konventionen und Standards (z.B. fehlende Kommentare oder falsche Funktionsnamen), Sicherheitslücken, Kontrollfluss- oder Datenflussanomalien gefunden werden (vgl. Andreas Spillner, 2012, S.99 ff.).

2.5.3.2 Contract Test

Durch die Verbreitung von Microservices gewinnen *Contract Tests* zunehmend an Bedeutung. Microservices als verteiltes System kommunizieren über standardisierte Schnittstellen miteinander. Diese Schnittstellen sind je nach Implementierung des Systems unterschiedlich in ihrer Art und der verwendeten Technologie. In der Regel findet die Kommunikation entweder über REST, Remote Procedure Calls, Messaging oder in Form von Events in einer Event-Gesteuerten Architektur mithilfe von Queue-Datenstrukturen statt. Contract Tests abstrahieren dieses Verhalten und betrachten die Kommunikation über eine Schnittstelle als Vertrag zwischen zwei Parteien, dem Anbieter und dem Verbraucher. Ein Anbieter liefert Daten an den Verbraucher, während der Verbraucher die erhaltenen Daten vom Anbieter verarbeitet (vgl. Pact.io, 2018). Bei einer Kommunikation via REST wäre beispielsweise der Service, der einen API-Endpunkt zur Verfügung stellt ein Anbieter. Ein anderer Service, der diesen Endpoint verwenden, um Daten abzufragen oder zu manipulieren, ist ein Verbraucher. In einer Event-Gesteuerten Architektur wäre der Publisher, der Daten in einer Queue veröffentlicht, ein Anbieter, während ein Subscriber, der diese Queues abonniert, liest und verarbeitet, ein Verbraucher ist.
Contract Tests überprüfen, ob sich die Implementation des Anbieters und die des Verbrauchers einer Schnittstelle an den gemeinsamen Vertrag halten. Sie eignen sich als eine Testmethode für die Durchführung von Regressionstests um abzusichern, dass die Kommunikation über eine Schnittstelle gemäß des Vertrages zwischen Verbraucher und Anbieter bei Änderungen am System weiterhin wie spezifiziert funktioniert (vgl. Fowler, 2011).

Consumer Driven Contracts Consumer Driven Contracts basieren auf der Idee, dass die Aufrufer einer Schnittstelle besser wissen, wie sie diese nutzen. Bei Consumer Driven Contracts stellt ein Verbaucher der einen Provider nutzen möchte, diejenigen Tests zur Verfügung, die aus seiner Sicht die erwartete Nutzung des Anbieters beschreiben. Der Verbraucher spezifiziert den Vertrag, wie die Nutzung der Schnittstelle auszusehen hat. Liefert der Anbieter eine Antwort, die der Verbraucher korrekt verarbeiten kann, gilt der Vertrag als erfüllt (vgl. Robinson, 2006). Die so definierten Verträge zwischen Anbieter und Verbraucher lassen sich automatisieren, um die intern genutzten Schnittstellen zwischen den einzelnen Services eines Systems zu definieren und zu prüfen. Fehler bei Änderungen einer Schnittstelle zwischen zwei Services können so leichter gefunden und behandelt werden.

2.5.3.3 Exploratory Testing

Exploratory Testing ist ein erfahrungsbasiertes Testentwurfsverfahren. Die Besonderheit beim Exploratory Testing ist, dass keine explizite Testvorbereitung stattfindet. Auch werden keine erwarteten Ergebnisse vor Durchführung der Tests spezifiziert und die Testdurchführung erfolgt willkürlich. Der Tester „erkundet“ die Software durch das Ausprobieren von Funktionen und das Spielen mit unterschiedlichen Eingaben. Der Testentwurf und die Testdurchführung finden dabei parallel statt (vgl. Society, 2014, S.89).

Der Tester kann beim Exploratory Testing auf unterschiedliche Testmuster zurückgreifen, wie zum Beispiel seine Erfahrung, Intuition, Modelle, realistische Benutzungsszenarien oder auch User Stories. Exploratives Testing ergänzt systematische Testentwurfsverfahren, ersetzt diese aber nicht. Durch den Einsatz des Verfahrens können Fehler aufgedeckt werden, die nur schwer durch systematische Testentwurfsverfahren gefunden werden können und das Verfahren ist durch das Entfallen einer langen Testplanung kurzfristig einsetzbar. Exploratory Testing lässt sich auch zu einem Entwicklungszeitpunkt einsetzen, an dem noch wenig Dokumentation zur Software oder Domänenwissen im Entwicklerteam vorhanden ist (vgl. Whittaker, 2009, S.21 ff.). Leider ist Exploratory Testing nicht automatisierbar und die Ergebnisse hängen stark vom Wissen und der Erfahrung des Testers ab. Trotzdem können Tools eingesetzt werden, die zum Beispiel Bildschirm- oder Mausbewegungen aufzeichnen, woraus Testfälle abgeleitet werden können. Es kann auch passieren, dass der Tester in bestimmte Denkmuster verfällt und so wichtige Probleme nicht aufgedeckt werden (vgl. Whittaker, 2009, S.24 ff.).

3 Testen von Microservices

In diesem Kapitel wird die Testpyramide als klassischer Ansatz für das Testen von Anwendungen in Kontrast zu aktuellen Testkonzepten für Microservices gestellt. Dazu wird zuerst auf die Testpyramide nach Mike Cohn eingegangen und Probleme im Bezug auf die Anwendung bei Microservices erläutert. Anschließend werden aktuelle Testkonzepte dargestellt, miteinander verglichen und als Abschluss dieses Kapitels bewertet. Auf Grundlage dieser Bewertung wird in Kapitel 4 ein individuelles Testkonzept für ein Versuchssystem entwickelt.

3.1 Testpyramide nach Mike Cohn

In folgendem Abschnitt wird die Testpyramide nach Mike Cohn beschrieben, die vor allem bei dem Testen von monolithischen Softwaresystemen genutzt wird. Die Testpyramide wird als Darstellungsmittel oft für weitere Testansätze adaptiert. Außerdem wird in diesem Abschnitt diskutiert, warum die Anwendung dieses Modells auf Microservice-basierte Systeme Schwierigkeiten bereitet. Die Bezeichnungen der einzelnen Teststufen weichen dabei von der Definition nach ISTQB ab.

3.1.1 Testpyramide

Cohn (2009b) beschreibt in seinem Buch „Succeeding with Agile“ erstmalig die sogenannte Testpyramide. Die Testpyramide besteht aus drei verschiedenen Teststufen, die Cohn selbst in Unit-Tests, Service-Tests und UI-Tests unterteilt (vgl. Cohn, 2009a). Das Bild einer Testpyramide versucht dabei grafisch die Verteilung unterschiedlicher Teststufen darzustellen und zu verdeutlichen.
Das Fundament der Pyramide bildet ein hoher Anteil an Unit-Tests. Sie sind schnell ausführbar und einfach in der Wartung, weshalb ein System möglichst breit von Unit-Tests abgedeckt sein sollte. Die Automatisierung von Unit-Tests eignet sich hervorragend für das Durchführen von Regressionstests bei Änderungen an der Codebasis und gibt dem Entwicklerteam schnelles Feedback. Aufgrund der hohen Isolation eines Unit-Tests für einen einzelnen Codeabschnitt, lassen sich Fehlerbehebungen ohne großen Aufwand durchführen (vgl. Cohn, 2009a).

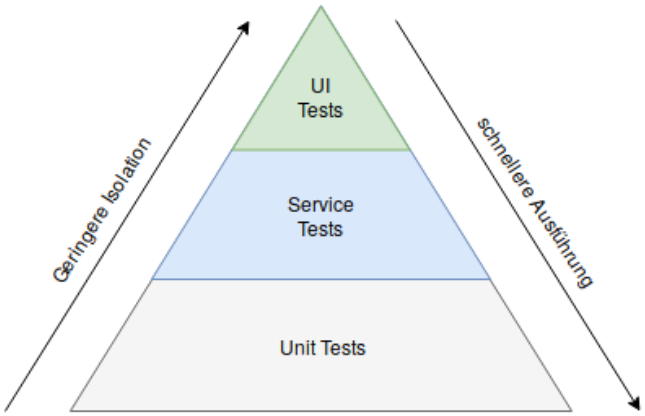

Abbildung 3.1: Die klassische Testpyramide, angelehnt an Cohn (2009a)

Die zweite Ebene bilden die sogenannten Service-Tests. Cohn fasst unter dem Begriff Service-Tests eine Reihe von Integrationstests zusammen. Sie haben längere Ausführungszeiten als Unit-Tests und sind aufwendiger in der Wartung, weshalb Service-Tests zur Überprüfung wichtiger Schnittstellen eingesetzt werden sollten. Er betrachtet Service-Tests als diejenigen Tests, welche die Business Logik der Anwendung ohne das Hinzuziehen der Bedienungsoberfläche (UI) und der verwendeten Frameworks testen. Am Beispiel einer Webanwendung kann dies das Testen über eine Schnittstelle sein, ohne die grafische Oberfläche aufzurufen (vgl. Fowler, 2012).
Die Spitze der Testpyramide bilden letztendlich die UI-Tests. Durch Testen des Codes auf erster Ebene und der Anwendungslogik auf zweiter Ebene, werden bereits viele Testfälle abgedeckt. UI-Tests testen ausschließlich, ob die Bedienungsoberfläche sich fehlerfrei verwenden lässt. Sie sind sehr aufwendig in ihrer Pflege und eignen sich um zu überprüfen, ob die Anwendung in Gänze funktioniert. Allerdings ist das Testen einer möglichst großen Anzahl an Zweigen im Code mit ihnen schwer möglich (vgl. Cohn, 2009a).

Das Modell der Testpyramide nach Cohn wurde seit seiner Vorstellung viel diskutiert. Heute wird meist eine Darstellung für die Testpyramide verwendet, welche die Service Tests in drei weitere Ebenen unterteilt. Dabei wird zusätzlich zwischen Komponententests, Integrationstests und API-Tests unterschieden (vgl. Scott, 2013). Auch werden manuelle Tests meist als Wolke dargestellt, was bedeutet, dass diese separat zu der Testpyramide jederzeit durchgeführt werden können. Dieser Ansatz ermöglicht neue Blickweisen auf den Begriff der von Cohn definierten Service-Tests. Komponententests als zweite Ebene nach den Unit-Tests testen die Logik der Anwendung anhand der spezifizierten Grundfunktionalitäten. Integrationstests testen das Zusammenspiel mehrerer Komponenten und überprüfen dadurch verschiedene Anwendungs-

szenarien der Software (vgl. Froslie, 2016). API-Tests hingegen testen die Schnittstellen auf Aspekte wie Funktionalität, Verlässlichkeit, Performance und Sicherheit.

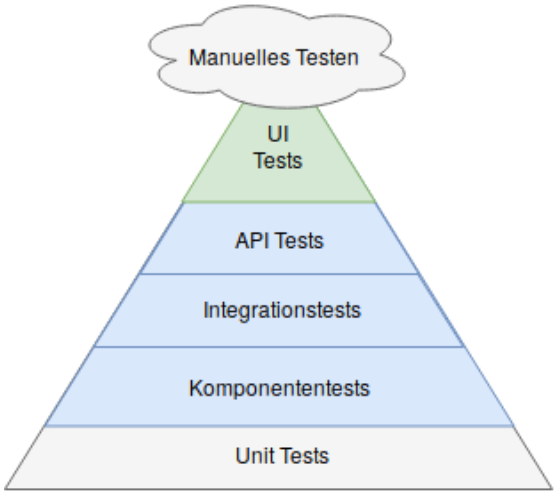

Abbildung 3.2: Die erweiterte Testpyramide, angelehnt an Scott (2013)

3.1.2 Herausforderungen im Bezug auf Microservices

Die Testpyramide als Modell für die Verteilung und Reihenfolge unterschiedlicher Teststufen lässt sich auf monolithische Systeme anwenden. Mit Blick auf Microservice-basierte Systeme ergeben sich neue Herausforderungen, die das Anwenden der Testpyramide als Modell für die Entwicklung einer geeigneten Teststrategie schwierig machen. Die Testpyramide ist aus der Annahme heraus entstanden, dass ein großer Teil der Komplexität eines Systems innerhalb der Anwendung selbst liegt. Ein solides Fundament aus vielen kleinen, isolierten und einfach zu wartenden Unit-Tests, sowie die Mehrheit der Service-Tests soll garantieren, dass die Logik innerhalb der Anwendung wie spezifiziert funktioniert. In Richtung der Pyramidenspitze gibt es weniger Service-Tests, welche auf die Integration mit externen Systemen abzielt. Die Komplexität bei Microservices stellt allerdings nicht der einzelne Service selbst dar, sondern die Interaktion der einzelnen Services untereinander.

Als verteiltes System rufen sich die Services gegenseitig über standardisierte Schnittstellen auf Netzwerkebene auf, um die eigentlichen Funktionalitäten des System zur Verfügung zu stellen. Für das Testen von Microservices ist es daher wichtig zu wissen, wie die einzelnen Services miteinander interagieren und welche Abhängigkeiten zwischen den Services bestehen. Um möglichst geringe Testlaufzeiten zu erreichen, werden Microservices von ihren Abhängigkeiten

isoliert. Diese Isolierung funktioniert in der Theorie durch die lose Kopplung einzelner Services ohne großen Aufwand. In der Praxis kann sie aufgrund von fachlichen Designentscheidungen oder falsch abgebildeten Aufgabenverteilungen jedoch schnell herausfordernd werden. Die Schnittstellen der Services bilden die Kommunikationsgrundlage des Systems und müssen daher intensiv geprüft werden. Das Testen der Schnittstellen ist komplex, da oft sehr viele unterschiedliche Aufrufpfade existieren und auch hier eine Isolierung schwierig sein kann. Ein einzelner Microservice besitzt möglichst wenig Komplexität und nur so viele Zeilen Code, wie für das Erfüllen einer einzelnen gekapselten Aufgabe innerhalb des Gesamtsystems notwendig ist. Diese Aufgabe und der dafür implementierte Code sollte dabei ohne viel Aufwand von einem Entwickler nachvollzogen werden können. Diese Eigenschaften führen dazu, dass vor allem Service-Tests, welche die Kommunikation und das korrekte Zusammenspiel mit externen Systemen überprüfen, eine große Bedeutung haben und weniger der einzelne Service mit einer großen Anzahl an Unit-Tests im Vordergrund steht.

Der Umgang mit Änderungen erhält im Kontext von Microservices eine besondere Bedeutung. Als eigenständige Systeme lassen sich unterschiedliche Microservices unabhängig voneinander ändern und neue Versionen eines komplexen Anwendungssystems können in kürzeren Releasezyklen bereitgestellt werden. Dadurch ergeben sich neue Anforderungen an Testkonzepte, die das schnelle und unabhängige Durchführen von Änderungen unterstützen und den Änderungsprozess nicht verlangsamen oder sogar aufhalten.

Die Verlagerung dieser Komplexität in einer Microservice-Architektur aus einem einzelnen Service heraus in die Interaktion der Services untereinander hat zu unterschiedlichen Ansätzen für das Testen von Microservices geführt, wodurch sich neue Testkonzepte für das Validieren von Microservice-basierten Systemen ergeben haben.

3.2 Testkonzepte für Microservices

In diesem Abschnitt werden bestehende Testkonzepte für Microservices vorgestellt. Dabei werden die Konzepte nach Ansicht der jeweiligen Autoren wiedergegeben. Die genannten Teststufen weichen von den Definitionen nach ISTQB ab und entsprechen den Definitionen der jeweiligen Autoren.

3.2.1 Testkonzept nach Eberhard Wolff

Eberhard Wolff (2015) erläutert in seinem Buch „Microservices - Grundlagen flexibler Softwarearchitekturen“ ein Konzept für das Testen von Microservices. Wolff nimmt dabei Bezug auf die Testpyramide nach Cohn, unterteilt sein Testkonzept allerdings in zwei verschiedene Teile. Zum einen wird das Gesamtsystem getestet, zum anderen der individuelle Microservice selbst. Aus diesem Grund gibt es nach Ansicht Wolffs nicht nur eine Testpyramide für das Gesamtsystem, sondern jeweils eine zusätzliche Testpyramide für jeden einzelnen Service. Für Wolff ist Testautomatisierung im Zusammenhang mit dem Testen von Microservices essentiell. Um die Idee des unabhängigen und häufigen Deployments der Services zu ermöglichen, müssen Tests automatisiert werden, um die Qualität der Microservices sicherzustellen und Risiken beim Deployment zu vermeiden. Wolff betont daher die Bedeutung der Deployment-Pipeline als Werkzeug für automatisiertes Testen von Microservices, die nacheinander mehrere Tests ausführt und in angemessener Zeit Rückmeldung erteilt (vgl. Wolff, 2015, S.223). Manuelles Testen wäre aufgrund der Versionsvielfalt von vielen kleinen Microservices ein unnötiger Mehraufwand. Die Flexibilität, welche ein großer Vorteil von Microservices gegenüber monolithischen Systemen darstellt, würde dadurch stark eingeschränkt werden (vgl. Wolff, 2015, S.227).

3.2.1.1 Testen eines Microservices

Das Testkonzept für einen Microservice stellt Wolff als Pyramidenmodell dar. Dieses Testkonzept betrachtet einen einzelnen Service in Isolation zu seinen Abhängigkeiten. Diese sollen simuliert werden, wodurch auf eine gemeinsame Testumgebung mit anderen Services verzichtet werden kann. Nach Ansicht Wolffs hat das Verzichten auf eine gemeinsame Testumgebung und die Simulation von Abhängigkeiten eines Services positive Auswirkungen auf die Beanspruchung von Ressourcen und die Trennung von Verantwortlichkeiten für das Erstellen von Testfällen. Die Tests auf Ebene des einzelnen Microservices sollen vom jeweiligen Entwicklerteam eigenständig implementiert und gewartet werden (vgl. Wolff, 2015, S.231 f.).

Unit-Tests Mit Unit-Tests sollen die kleinstmöglichen Einheiten der einzelnen Bestandteile eines System getestet werden. Diese kleinstmöglichen Einheiten sind nach Wolff isolierte Funktionen und Methoden und die Tests sind vom dem Entwickler selbst zu implementieren. Unit-Tests lassen sich schnell ausführen und können direkt Feedback bei Veränderungen an der Codebasis geben. Wolff empfiehlt das Nutzen von Platzhaltern, sogenannten Mocks und Stubs, um kleinstmögliche Einheiten voneinander zu isolieren und die Ausführungsdauer von Unit-Tests gering zu halten (vgl. Wolff, 2015, S.231 ff.).

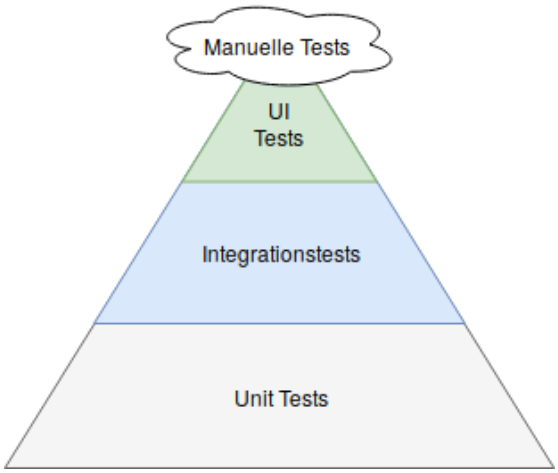

Abbildung 3.3: Die Testpyramide für einen Microservice, angelehnt an Wolff (2015)

Integrationstests Integrationstests überprüfen die korrekte Interaktion eines Microservices mit seinen Abhängigkeiten. Für die Durchführung von Integrationstests werden die Abhängigkeiten eines Services isoliert und mit stattdessen mit Stubs simuliert. Dies verringert die Ausführungsdauer eines Integrationstests und kann mögliche Fehler eingrenzen. Wolff empfiehlt den Einsatz von Consumer-Driven Contract Tests als zusätzliche Möglichkeit für das Durchführen von Integrationstests für einen Microservice (vgl. Wolff, 2015, S.233 f.).

UI Tests UI-Tests testen die Benutzeroberfläche eines Microservices. Wolff ist der Ansicht, dass UI-Tests sehr fragil und aufwendig zu implementieren sind. Benutzeroberflächen unterliegen oft Änderungen, welche dazu führen können, dass auch Tests angepasst werden müssen und diese oft fehlschlagen können. Aus diesen Gründen sollten nur wenige UI-Tests für das Testen der Benutzeroberfläche eines Microservices implementiert werden (vgl. Wolff, 2015, S.220).

Manuelle Tests Manuelle Tests können als Ergänzung für das Testen eines Microservices genutzt werden. Wolff erläutert, dass neben dem manuellen Testen von neuen Funktionalitäten auch Performanz- oder Sicherheitsaspekte in die manuellen Tests miteinbezogen werden können (vgl. Wolff, 2015, S.220).

Ergänzungen Wolff erwähnt weitere Möglichkeiten, die sein Testkonzept zusätzlich zu den genannten Teststufen ergänzen.

- **Logging** — Durch die zentrale Speicherung und das automatische Auswerten von Logdateien können auftretende Probleme nachvollzogen werden. Durch Logging können Entwickler über fehlerhaftes Verhalten informiert werden, auch wenn der Anwender des Systems von den Auswirkungen nicht betroffen ist (vgl. Wolff, 2015, S.244).
- **Monitoring** — Durch das Analysieren von Logdateien können Metriken als formale Vergleichs- und Bewertungsmöglichkeiten für einzelne Microservices und das Gesamtsystem gebildet werden. Durch Überwachung dieser Metriken kann falsches Verhalten abgeleitet und es können Maßnahmen für Nachbesserung ergriffen werden (vgl. Wolff, 2015, S.249).
- **Lasttests** — Durch Lasttest, welche Wolff in Performanz- und Kapazitätstests unterteilt, kann die Leistungsfähigkeit des Systems beurteilt werden. Performanztests testen die Ausführungsdauer von Systemfunktionen, wohingegen Kapazitätstests die parallele Bearbeitung von Systemfunktionen überprüfen. Beide Arten von Lasttests gehören für Wolff zu seinem Testkonzept dazu, allerdings erwähnt er diese nicht explizit in seinem Pyramidenmodell (vgl. Wolff, 2015, S.230 f.).

3.2.1.2 Testen des Gesamtsystems

Die Tests für das Gesamtsystem sollen Probleme im Zusammenspiel der einzelnen Services identifizieren. Diese Tests auf Makro-Ebene der Systemarchitektur müssen von allen Entwicklerteams gemeinsam weiterentwickelt werden, da sie nicht in die Zuständigkeit eines einzelnen Teams fallen. Beim Testen des Gesamtsystems sind alle Microservices betroffen, weshalb die Implementierung der Tests in Abstimmung mit allen Teams geschehen sollte.
Gegenüber dem Testen auf Mikro-Ebene fällt auf, dass Unit-Tests als Teststufe entfallen. Die Grundlage für das Testen des Gesamtsystems bildet eine breite Basis an Integrationstests. Darauf aufbauend werden UI-Tests durchgeführt, sowie wenige manuelle Tests (vgl. Wolff, 2015, S.225 f.).

Gemeinsame Integrationstests Durch gemeinsame Integrationstest wird das Zusammenspiel zwischen den einzelnen Microservices überprüft. Wolff verweist auf die Herausforderungen von verteilten Systemen. Microservices als solche bieten mehr Fehlerpotential durch ihre gegenseitige Interaktion als monolithische Systeme, weshalb das Testen dieser Interaktionen sehr wichtig ist. Er empfiehlt deshalb vor dem Deployment eines geänderten Services eine gemeinsame Teststufe für Integrationstests, in der alle Microservices gestartet und überprüft werden. Dabei darf nur ein geänderter Service zur Zeit in der gemeinsamen Testumgebung

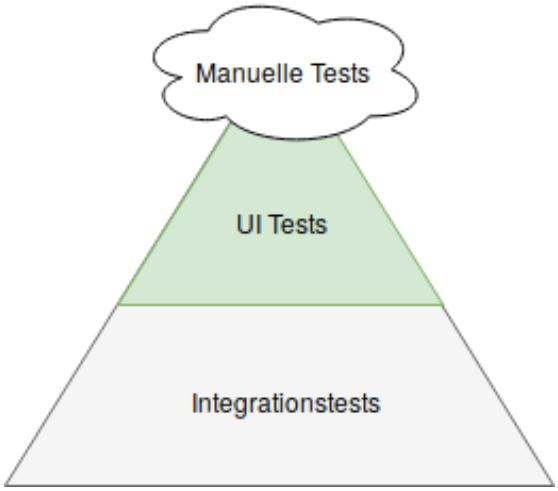

Abbildung 3.4: Die Testpyramide für das Gesamtsystem, angelehnt an Wolff (2015)

gestartet und mit den aktuellen Versionen anderer getestet werden. Wenn für mehrere veränderte Versionen unterschiedlicher Services gleichzeitig Integrationstests durchgeführt werden, kann das Analysieren von fehlerhaften Verhalten komplex sein.
Die gemeinsame Teststufe für Integrationstest kann allerdings den Entwicklungsprozess von Services einschränken. Durch eine lange Testausführung von Integrationstests auf Ebene des Gesamtsystems und die Beschränkung für veränderte Versionen von Services in der gemeinsamen Testumgebung, kann das Veröffentlichen von neuen Versionen in die Produktivumgebung verlangsamt werden. Wolff empfiehlt daher einen großen Teil der Funktionalitäten eines Microservices durch Integrationstest auf Mikro-Ebene und nicht auf Makro-Ebene des Gesamtsystems zu überprüfen (vgl. Wolff, 2015, S.227 f.).

UI-Tests UI-Tests für das Gesamtsystem sind nach Wolff analog zu UI-Tests der Benutzeroberfläche einzelner Microservices. Für die Durchführung von UI-Tests auf Makro-Ebene müssen alle Microservices gestartet werden, was zu einer vergleichsweise langen Testlaufzeit führen kann. Deshalb empfiehlt Wolff auch hier die Anzahl an UI-Tests möglichst gering zu halten (vgl. Wolff, 2015, S.220).

Manuelles Testen Manuelles Tests ergänzen die vorherigen Teststufen des Gesamtsystems. Durch manuelle Tests können neue Funktionalitäten getestet werden, die noch nicht in der Produktivumgebung zu finden sind. Wolff erläutert, dass manuelles Testen die Veröffentlichung eines Services verlangsamen kann. Daher sollten Microservices auch ohne die Durchführung

von manuellen Tests in die Produktion gegeben werden können, um schnelle Releasezyklen nicht zu blockieren (vgl. Wolff, 2015, S.227).

3.2.2 Testkonzept nach Sam Newman

In seinem Buch „Microservices - Konzeption und Design“ erörtert der Autor Sam Newman seinen Ansatz zum Testen von Microservice-Architekturen. Newman bezieht Stellung zu der Testpyramide nach Cohn und erläutert, dass dieses Modell aufgrund unterschiedlicher Interpretationen der verwendeten Terminologie für Teststufen schwierig ist. Dabei kritisiert er die durch Cohn definierten Begriffe der Unit- und Service-Tests als zu unkonkret. Trotzdem hält Newman an der Darstellung der einzelnen Teststufen als Pyramidenmodell fest und schlägt auf der Grundlage von Cohn's Testpyramide ein eigenes Modell vor.

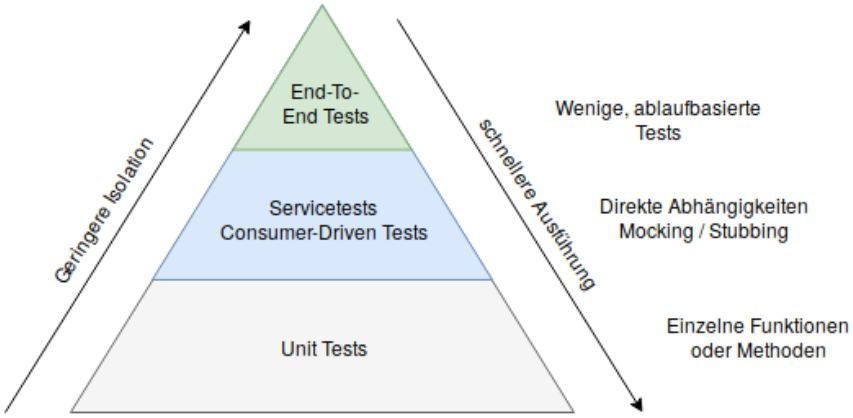

Abbildung 3.5: Die Testpyramide, angelehnt an Newman (2016)

Newman ersetzt Cohn's Begriff der UI-Tests durch End-To-End Tests, welche das gesamte System miteinbeziehen. Trotz der geäußerten Kritik an den Begriffen Service- und Unit-Tests, finden sich diese auch in Newman's Testpyramide wieder. Er ergänzt auf Ebene der Service-Tests die Consumer-Driven Tests, welche direkt zusammenhängende Services und Platzhalterservices als Testobjekte betrachten. Auf unterster Ebene finden sich die Unit-Tests wieder, welche einzelne Funktionen oder Methoden prüfen. Ähnlich wie Cohn möchte auch Newman mit dem Pyramidenmodell verdeutlichen, dass Tests auf den unteren Ebenen schneller und besser in Isolation ausgeführt werden können, wohingegen in Richtung Pyramidenspitze Stück für Stück weitere Systemteile in die Tests integriert werden und die Isolierung in den Hintergrund rückt (vgl. Newman, 2016, S.179).

Unit-Tests *Unit-Tests* testen einzelne Methoden oder Funktionsaufrufe. Für Newman ist es bei der Durchführung von Unit-Tests wichtig, dass keine weiteren Services gestartet werden. Auch die Nutzung von externen Dateien oder Netzwerkverbindungen soll unterlassen werden. Aufgrund der einfachen Implementation und der schnellen Ausführungszeit, müssen möglichst viele Unit-Tests als Grundlage für das Testen eines Microservices eingesetzt werden. Für das Refactoring von Code sind Unit-Tests nach Ansicht Newmans besonders wichtig, weil sie dem Entwickler sehr schnell Feedback liefern können und die meisten Fehler vor weiteren Teststufen aufdecken (vgl. Newman, 2016, S.180 f.).

Service-Tests *Service-Tests* testen einen einzelnen Microservice und dessen Funktionalitäten in Isolation zu anderen Services. Sie umgehen die Benutzerschnittstelle des Gesamtsystems und rufen Funktionen unmittelbar am Service direkt auf. Um die Isolation eines Services zu gewährleisten, werden seine Abhängigkeiten durch Platzhalterservices ersetzt. Als Platzhalterservices unterscheidet Newman ebenfalls zwischen Stubs und Mocks.
Ein Stub simuliert Antworten auf Anfragen. Ist ein Service für die Ausführung seiner Aufgabe auf die Antworten von Abhängigkeiten angewiesen, können diese in Form von statisch hinterlegten Rückgabewerten von Schnittstellenaufrufen durch Stubs zur Verfügung gestellt werden. Ein Mock hingegen ist ein Platzhalter, der ein intelligentes Verhalten eines Services simuliert. Er bietet mehr Funktionen als ein einfacher Stub an und ist aufwendiger zu implementieren, da dieser selbst eine einfache Logik besitzt. Ein Mock kann zum Beispiel eine Logik implementieren die überprüft, ob ein Aufruf tatsächlich stattgefunden hat, wohin gegen ein Stub nur einen simplen Rückgabewert auf eine gestellte Anfrage liefert. Newman empfindet die die Abwägung zwischen der Verwendung von Mocks und Stubs als schwierig, empfiehlt jedoch die Verwendung von Stubs aufgrund ihrer Einfachheit (vgl. Newman, 2016, S.184 f.).

Consumer-Driven Tests Bei Consumer-Driven Tests unterscheidet Newman zwischen zwei Parteien, dem Consumer und dem Provider. Der Consumer ist ein Service der eine Schnittstelle nutzen möchte. Ein Provider wiederum bietet einen Dienst über eine Schnittstelle an. Consumer-Driven Tests definieren die Ansprüche des Consumers an einen Provider in Form eines Vertrages. Sie stellen Ansprüche an das Verhalten von direkten Abhängigkeiten eines Microservices, wobei Abhängigkeiten durch Platzhalter simuliert werden. Weil Consumer-Driven Tests überprüfen, wie ein Consumer von einem Service Gebrauch macht, ist der Auslöser für das Fehlschlagen eines Tests ein anderer als bei Servicetests. Aus diesem Grund sieht Newman beide Teststufen auf der selben Ebene innerhalb seines Pyramidenmodells, trennt sie allerdings

klar voneinander ab (vgl. Newman, 2016, S.192 f.).

End-To-End Tests End-To-End Tests testen das gesamte System, indem sie die über eine Benutzerschnittstelle bereitgestellten Funktionalitäten überprüfen. Diese Funktionalitäten werden dabei durch eine Reihe an unterschiedlichen Services realisiert, die sich gegenseitig über ihre Schnittstelle aufrufen. Sie integrieren weite Teile des Gesamtsystems und nutzen einen großen Teil der zugrundeliegenden Codebasis. Nach Ansicht Newmans liefert ein erfolgreich bestandener End-To-End Test ein hohes Maß an Vertrauen in das System, allerdings rät er aus unterschiedlichen Gründen von der Implementation von zahlreichen End-To-End Tests ab.
Wegen der hohen Integration der einzelnen Systembestandteile kann die Testausführungszeit bei End-To-End Tests sehr lang sein. Das Veröffentlichen und Starten aller Microservices um die Systemfunktionalitäten zu testen, kann sehr viel Zeit in Anspruch nehmen und den Umfang eines solchen Tests drastisch steigern. Aufgrund der vielen veränderlichen Systembestandteile, die ein End-To-End Test miteinbezieht, ist eine Ursache für das Scheitern schwer einzugrenzen und kann zu unzuverlässigen und fragilen Testergebnissen führen. Wenn ein End-To-End Test manchmal fehlschlägt, aber bei wiederholten Ausführung erfolgreich durchgeführt werden kann, dann ist dieser unzuverlässig und sollte nach Newman möglichst schnell entfernt werden, da ansonsten das Vertrauen in die Testsuite verloren gehen kann. Die lange Ausführungsdauer und eventuelle Ungenauigkeit der End-To-End Tests wirkt sich negativ auf die Zusammenarbeit zwischen Entwicklerteams und das iterativen Vorgehen beim agilen Entwicklungsprozess aus. Newman empfiehlt, sich beim Entwurf von Testsuiten auf grundlegende Benutzerabläufe zu beschränken und nicht für jede spezifizierte Funktionalität einen End-To-End Test zu entwerfen (vgl. Newman, 2016, S.184 ff.).

3.2.2.1 Testautomatisierung durch eine Deployment-Pipeline

Newman erläutert ein Standardverfahren zur Handhabung serviceübergreifender End-To-End Tests. Er präsentiert eine naive Herangehensweise, bei der jeder Service eine eigene Deployment-Pipeline und eine eigene Stufe für End-To-End Tests besitzt. Für Newman ist dieser Ansatz aus unterschiedlichen Gründen problematisch. Bei End-To-End Tests als letzte Teststufe in der Deployment-Pipeline eines Microservices ist nicht klar, welche Versionen von Services in der Testumgebung gestartet werden sollen. Die einzelnen Teststufen für alle Services müssen synchron gehalten werden, weshalb die Anpassung der Testsuite mit großem repetitiven Aufwand verbunden ist. Newman schlägt daher ein Zusammenführen der einzelnen Deployment-Pipelines zu einer gemeinsamen End-To-End Teststufe vor.

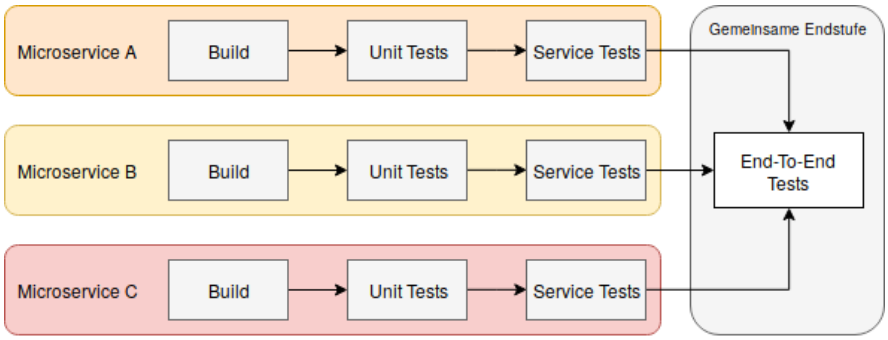

Abbildung 3.6: Eine Deployment-Pipeline für Microservices, angelehnt an Newman (2016)

Ein Service wird nach dem erfolgreichen Bestehen von Unit- und Servicetests in eine gemeinsame End-To-End Teststufe überführt. Newman betont hierbei, dass nur ein Service zur Zeit in der letzten Teststufe überprüft werden darf. Er zeigt gleichzeitig einen Nachteil seines Konzeptes auf, da die Beschränkung der End-To-End Teststufe die Menge an Releases an einem Werktag begrenzen kann. Dieses Problem wird zunehmend kritischer mit längerer Testausführung. Die mögliche Ungenauigkeit von End-To-End Tests kann dazu beitragen, dass durch das erneute Ausführen von Tests mit langer Ausführungsdauer sich Microservices in der Deployment-Pipeline anstauen und somit den Entwicklungsprozess gefährden. Trotzdem präferiert Newman eine gemeinsame Endstufe gegenüber einer End-To-End Teststufe pro Service (vgl. Newman, 2016, S.186 f.).

3.2.2.2 Testen nach der Veröffentlichung

Für Newman endet der Testprozess nicht mit dem Deployment eines neuen Services in die Produktivumgebung. Obwohl die meisten Tests vor dem Deployment in Produktion durchgeführt werden, ist es für Newman wichtig auch Fehler aufzufangen, die im laufenden Betrieb auftreten. Newman ist der Ansicht, dass es nicht möglich sei, durch eine große Menge an Tests vor der Veröffentlichung eines Services, die Fehlerwahrscheinlichkeit auf Null zu reduzieren. Er nennt aus diesem Grund zwei verschiedene Deployment-Strategien für Microservices, die zur Trennung von Deployment und Veröffentlichung beitragen (vgl. Newman, 2016, S.196 ff.):

- **Blue / Green-Deployment** — Beim Blue/Green-Deployment wird eine aktive (z.B. Blue) und eine inaktive (z.B. Green) Laufzeitumgebung desselben Systems betrieben. Eine neue Version wird stets in der Green Umgebung entwickelt. Anfragen mit echtem Datenverkehr werden nur an die Version in Blue gestellt. Ist die neue Version getestet, wird der echte Datenverkehr auf sie umgeleitet. Die frühere Version wird noch einige Zeit

parallel betrieben, damit bei unerwarteten Fehlern schnell zur letzten stabilen Version gewechselt werden kann.

- **Canary Releasing** — Beim Canary Releasing wird die neu veröffentlichte Version eines Services parallel zur bereits Laufenden in der Produktivumgebung gestartet. Anschließend wird in kleinen Schritten die Last auf den neuen Microservice umgeleitet oder kopiert. Durch dieses Vorgehen können funktionale und nicht-funktionale Anforderungen an die neue Version überprüft werden und bei Problemen ist lediglich ein geringer Teil der Nutzer des Systems betroffen.

Newman betont die Bedeutung des Testens von funktionsübergreifenden Anforderungen nach der Veröffentlichung. Diese Cross-Funktionalen Tests definiert er als eine Menge aus sowohl funktionalen, als auch nicht funktionalen Tests, wobei eine genaue Einordnung sich oft als schwierig herausstellt. Newman erläutert, dass in einer Microservice-Architektur Performanztests wichtig sind. Auf Grund der Kommunikation über Prozessgrenzen hinweg auf Netzwerkebene, ist die Last innerhalb des Netzwerkes höher als bei monolithischen Anwendungen. Deshalb sollten Performanztests möglichst früh angesetzt werden, um Probleme aufzudecken. Sie sollen in regelmäßigen, kurzen Abständen ausgeführt werden (vgl. Newman, 2016, S.202 f.).

3.2.3 Testkonzept nach Toby Clemson

In seinem Vortrag „Testing Strategies in a Microservice Architecture“ stellt Toby Clemson ein Konzept für das Testen von Microservices vor.

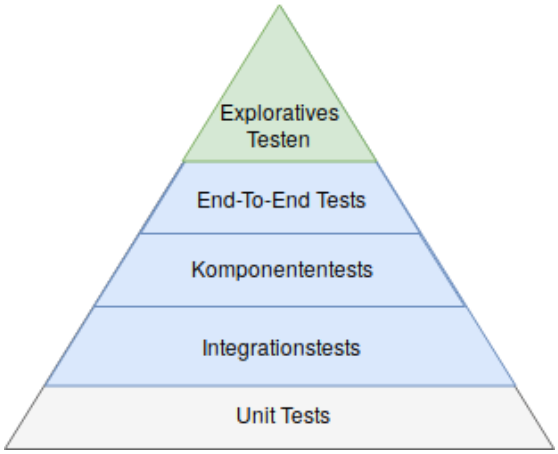

Abbildung 3.7: Die Testpyramide, angelehnt an Clemson (2014)

Clemson folgt der Annahme, dass Microservices in der Regel eine ähnliche interne und externe Struktur aufweisen, die aus verschiedenen Schichten besteht. Die angewandten Teststrategien sollen darauf abzielen, jede Schnittstelle zwischen den Schichten des Services, intern wie extern, abzudecken und gleichzeitig leichtgewichtig zu bleiben. Die klare Trennung zwischen interner und externer Struktur von Microservices erlaubt Freiheiten im Bezug auf den Umfang bei der Gestaltung von Tests. So können Services, die einen wichtigen Teil der Business Logik implementieren, umfangreicher getestet werden als Andere.
Clemson entwirft auf Grundlage seiner Erkenntnisse über die interne und externe Struktur von Microservices und Cohn's Testpyramide ein eigenes Pyramidenmodell (Abb. 3.7), um Microservices zu validieren. Die Darstellung als Pyramide soll wie auch bei dem Modell nach Cohn das Verhältnis der Menge an Tests pro einzelner Teststufe verdeutlichen.

Unit Testing Ein Unit-Test überprüft nach Clemson die kleinste testbare Einheit einer Anwendung, um festzustellen, ob sie sich wie erwartet verhält. Die Größe der kleinsten zu testenden Einheit ist nicht genau definiert, jedoch werden die Unit-Tests in der Regel auf Klassenebene oder um eine kleine Gruppe von zusammenhängenden Klassen herum geschrieben. Je kleiner die zu testende Einheit, desto einfacher ist es, das Verhalten mit einem Unit-Test auszudrücken, da die Zweigüberdeckung der Einheit selbst sehr gering ist.
Clemson weist darauf hin, dass ein Unit-Test nicht nur eine nützliche Testart darstellt, sondern auch ein hilfreiches Designwerkzeug in Kombination mit testgetriebenen Entwicklung sein kann. Wenn sich das Verhalten einer Einheit schwer in einem Unit-Test ausdrücken lässt, ist das ein Hinweis dafür, dass die Einheit in kleinere Einheiten zerlegt werden sollte und diese Teile wiederum einzeln getestet werden sollten. Clemson differenziert zwei verschiedene Arten von Unit-Tests (vgl. Clemson, 2014, S.7):

- **Sociable Unit Testing** — Im Mittelpunkt des *Sociable Unit Tests* steht die Überprüfung des Verhaltens einer Einheit durch Beobachtung von Zustandsänderungen. Mit der zu testenden Einheit wird lediglich über ihre Schnittstelle interagiert und sie selbst wird wie eine Black-Box behandelt.
- **Solitary Unit Testing** — *Solitary Unit Testing* betrachtet die Interaktion und Zusammenarbeit zwischen einer Einheit und ihrer Abhängigkeiten, die durch sogenannte „Test Doubles" in Form von Mocking ersetzt werden.

Clemson unterscheidet demnach zwischen Einheiten, die eher zustandsorientiert sind und Einheiten, die überwiegend Kommunikation- und Koordinationslogik implementieren. Je nach

Art der Einheit soll entweder Sociale- oder Solitary Unit-Testing zur Überprüfung des korrekten Verhaltens verwendet werden.

Integration Testing Ein Integrationstest überprüft die Kommunikationswege und Interaktionen zwischen den Komponenten, um frühzeitig Schnittstellenfehler aufzudecken. Integrationstests fassen mehrere Einheiten zusammen und testen sie als ein zusammenhängendes Teilsystem. Sie überprüfen ob die einzelnen Einheiten wie beabsichtigt zusammenarbeiten. In einer Microservice-Architektur sollen deshalb die Komponenten getestet werden, die mit externen Komponenten kommunizieren. Clemson betont jedoch, dass das Testen der externen Komponente nicht in den Aufgabenbereich des Integrationstests hineinfallt. Der Integrationstest hat lediglich das Ziel zu überprüfen, dass die Komponenten eines Microservices erfolgreich mit externen Komponenten anderer Services kommunizieren können. Clemson unterscheidet dabei zwischen zwei verschiedenen Arten von Integrationstests (vgl. Clemson, 2014, S.10 ff.):

- **Gateway Integration Test** — Gateway Integration Test sind Integrationstests, welche Fehler auf Protokollebene identifizieren sollen. Dazu zählen fehlerhafte HTTP-Header und Request/Response-Bodies, ebenso wie falsche TLS-Handhabungen und das Verhalten bei Netzwerkfehlern. Fehlerhaftes Verhalten wie Timeouts oder langsame Reaktionszeiten können schwierig für Testfälle provoziert werden. Clemson empfiehlt deshalb das Ersetzen von externen Komponenten durch Stubs.
- **Persistence Integration Test** — Persistence Integration Tests sind Integrationstest, die das Zusammenspiel zwischen dem Service und externen Datenspeichern überprüfen sollen. Konkret können dabei objektrelationale Abbildungen zwischen dem Service und seiner relationalen Datenbank getestet werden. Zum anderen können Persistence Integration Tests auf Zeitüberschreitungen und Netzwerkausfälle bei einer verteilten Partitionierung der verwendeten Persistenzlösung prüfen.

Clemson ist der Ansicht, dass Integrationstests ein schnelles Feedback beim Refactoring oder bei der Erweiterung der extern kommunizierenden Komponenten liefern. Allerdings sind sie fragiler als andere Teststufen und können aufgrund der Abhängigkeiten zu externen Komponenten leichter fehlschlagen. Clemson empfiehlt daher nur eine Handvoll Integrationstests zu implementieren und zusätzliche Testabdeckung durch Unit-Tests und Contract Tests (siehe Abschn. 3.2.3) zu gewährleisten (vgl. Clemson, 2014, S.11).

Component Testing Beim Komponententest wird ein Microservices als eine zusammenhängende Einheit innerhalb des Gesamtsystems in Isolation zu seinen Abhängigkeiten getestet.

Diese Tests sollen das Verhalten eines Services durch Manipulation seiner Schnittstellen aus Sicht eines Aufrufers überprüfen. Clemson erläutert zwei verschiedene Vorgehensweisen für die Implementation von Komponententests. Bei dem *In-Process* Ansatz soll die Verwendung von Netzwerkverkehr vermieden werden. Test Doubles und Persistenzlösungen werden deshalb In-Memory gestartet, sodass mithilfe von Interprozesskommunikation der Komponententest ohne Netzwerkaufrufe ausgeführt werden kann. Dies hat Geschwindigkeitsvorteile bezüglich der Testlaufzeit, setzt allerdings voraus, dass ein Microservice für die Nutzung von In-Memory Ressourcen zu Testzwecken konfigurierbar ist. Als zweite Vorgehensweise nennt Clemson den *Out-Of-Process* Ansatz, nämlich Simulation von Abhängigkeiten außerhalb des Services als Stub. Dabei werden die externen Abhängigkeiten als separate Prozesse gestartet und die Interaktion findet über echte Netzwerkaufrufe statt. Der Microservice wird nicht in einem speziellen Testmodus gestartet, sondern kann ohne zusätzliche Testparameter ausgeführt werden. Dies hat Nachteile mit Blick auf die Testlaufzeit, kann allerdings bei komplexer Integrations-, Persistenz- oder Startlogik eines Microservices der geeignetere Ansatz für die Implementation eines Komponententests sein (vgl. Clemson, 2014, S.13 ff.).

Contract Testing Ein Integration Contract Test überprüft das Einhalten eines Vertrages zwischen der Schnittstelle eines Services und dessen Aufrufern. Diese Tests bieten einen Mechanismus an, um explizit zu überprüfen, ob eine Komponente einen Vertrag erfüllt oder nicht. Um das Verhalten eines Services zu testen, empfiehlt Clemson die Implementation von Consumer Driven Contracts (siehe Abschn. 2.5.3.2). Sie ermöglichen schnelles Feedback an den Anbieter und Verbraucher, ob Änderungen an einem Service die Vertragsbedingungen verletzen (vgl. Clemson, 2014, S.18 f.).

End-To-End Testing Ein End-To-End Test verifiziert, dass das gesamte System den spezifizierten Anforderungen und Nutzungsszenarien entspricht. Es wird das gesamte System von „einem Ende bis zum Anderen“ (Clemson, 2014, S.20) getestet. Das System wird dabei vollständig deployed und als eine Black-Box behandelt. Die Tests werden über die Benutzeroberfläche oder öffentliche Service Schnittstellen durchgeführt. Clemson betont, dass gerade in Microservice-Architekturen ein End-To-End Test ein hohes Maß an Vertrauen liefert, da das Zusammenarbeiten vieler einzelner Services getestet wird.
Obwohl für End-To-End Tests das gesamte System deployed werden soll, kann es sinnvoll sein, Drittanbieter-Services durch Platzhalter zu ersetzen. Das Mocken dieser Services kann das Vertrauen in einen End-To-End Test verringern, aber dafür positive Auswirkung auf die Stabilität eines Tests haben.

Clemson ist der Ansicht, dass das sinnvolle Entwerfen von End-To-End Tests eine Herausforderung darstellt. Aufgrund der vielen involvierten Services können End-To-End Tests schnell zu unzuverlässigen Testergebnisse führen. Clemson gibt daher eine Reihe an Empfehlungen für den Testentwurf:

1. *Wenige End-To-End Tests* — Es sollten so wenig End-To-End Tests wie möglich implementiert werden. Die Festlegung einer zeitlichen Beschränkung für die Dauer eines End-To-End Tests ist sinnvoll und kann dabei helfen, geeignete Tests zu definieren. Allgemein sind End-To-End Tests sehr fragil und können sehr leicht ungenaue Testergebnisse liefern. Die meisten Funktionalitäten sollten daher durch andere Teststufen abgedeckt werden.

2. *Bezug auf Personas / User Stories* — Beim Entwurf von End-To-End Tests ist es hilfreich sich an Personas und User Stories zu orientieren, um sinnvolle, ablauforientierte End-To-End Tests zu definieren. End-To-End Tests sollten sich auf die Aspekte eines Systems fokussieren, die aus Sicht des Benutzers wertvoll sind. Eine große Abdeckung der Codebasis wird anderen Teststufen überlassen.

3. *Geeignete Auswahl von Endpunkten* — Die Endpunkte für End-To-End Tests sollten geeignet ausgewählt werden. Wenn ein externer Service oder die Benutzeroberfläche der Auslöser für ungenaue Testergebnisse ist, dann sollten die Grenzen des Systems in der Testumgebung neu definiert werden und diese Komponenten aus dem End-To-End Test ausgeschlossen werden.

4. *Nutzung von Infrastructure-as-Code (IaC)* — Durch den Prozess des Verwaltens und Provisionierens von Rechen- und Netzwerkinfrastruktur in Form von maschinenlesbaren Definitionsdateien (vgl. Michael Wittig, 2015, S.93), lassen sich diese automatisiert und reproduzierbar aufbauen. Die Komplexität des Deployment einer Microservice-Architektur kann dadurch begrenzt werden. Durch den Einsatz von IaC lässt sich ohne großen Aufwand eine frische Laufzeitumgebungen für jede Durchführung eines End-To-End Tests erstellen.

5. *Unabhängigkeit von Testdaten* — End-To-End Tests sollten benötigte Testdaten über die Schnittstellen der Services generieren können, statt manuelle Datenimporte für die Testdurchführung vorauszusetzen.

Exploratory Testing Durch Explorative Tests wird das System manuell überprüft. Die gewonnenen Erfahrungen können nach Clemson in den Entwurf oder die Weiterentwicklung von geeigneten End-To-End Tests einfließen. Dabei soll sich auf Aspekte konzentriert werden, die bisher nicht durch automatisierte Tests abgedeckt worden sind.

3.2.4 Testkonzept nach Schaffer

In dem Artikel „Testing of Microservices“, veröffentlicht auf dem firmeneigenen Technologieblog der Spotify AB, erläutert André Schaffer (2018) gemeinsam mit seinem Kollegen Rickard Dybeck ein Testkonzept für Microservices. Schaffer ist der Ansicht, dass die klassische Testpyramide nach Cohn (Abbildung. 3.1) für die Organisation von Tests für Microservices nicht geeignet sei.

Die größte Komplexität in einem Microservice liegt nicht innerhalb des Services selbst, sondern darin, wie er mit anderen Services interagiert. Darauf sollte nach Ansicht Schaffers ein besonderes Augenmerk gelegt werden. Eine große Menge an Unit-Tests, wie es die Testpyramide als Fundament vorschlägt, schränken die Änderungsmöglichkeiten am Code ein. Wenn Quellcode geändert wird, muss ebenfalls der dazu erstellte Unit-Test angepasst werden. Wenn ein Unit-Test wiederum geändert wird, geht Vertrauen verloren, dass der Code immernoch das tut, wozu er ursprünglich implementiert worden war. Schaffer vertritt die Meinung, dass das Anpassen von Unit-Tests nach Codeänderungen einen negativen Einfluss auf die Geschwindigkeit in der agilen Softwareentwicklung besitze (vgl. André Schaffer, 2018).

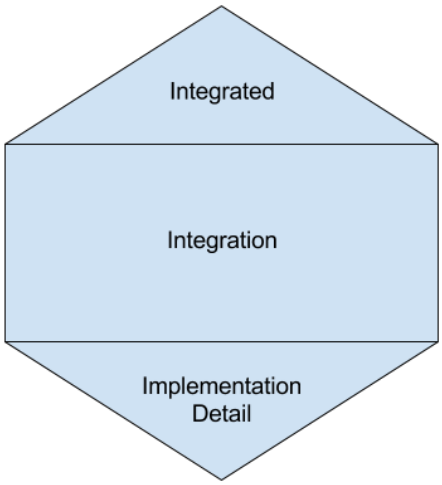

Abbildung 3.8: Die „Microservice Testing Honeycomb“ nach André Schaffer (2018)

Als Modell für das Testen von Microservices stellt Schaffer die „Microservice Testing Honeycomb“ vor, die sich in die drei Teststufen Integrated Tests, Integration Tests und Implementation Detail Tests unterteilt. Die Darstellungsform als Wabe verdeutlicht die Verteilung der einzelnen Testfälle. Der Schwerpunkt sollte hierbei auf der Implementierung von Integrationstests für Microservices liegen. Viel Komplexität in einer Microservice-Architektur liegt in der Art und Weise, wie die Services miteinander interagieren, weshalb eine hohe Anzahl an Integrationstest sicherstellen soll, dass diese Zusammenarbeit fehlerfrei funktioniert.

Integrated Tests Schaffer definiert den sogenannten „Integrated Test“ als ein Test, welcher auf Grundlage der Korrektheit eines anderen Systems bestanden oder nicht bestanden wird (vgl. André Schaffer, 2018). Integrated Tests sind charakterisiert durch z.B. die Ausführung eines anderen Services in einer lokalen Testumgebung, den Test gegen einen anderen Service in einer geteilten Testumgebung oder wenn Änderungen an dem eigenen Microservice die Tests eines anderen Services fehlschlagen lassen. Ein einzelner Microservice sollte Schaffers Ansicht nach möglichst wenig, bis idealerweise keine solcher Tests besitzen. Er bezieht sich dabei auf Rainsberger (2010), der in seinem Blogartikel „Integrated Tests Are A Scam“ anführt, dass Integrated Tests vergleichsweise aufwendig zu implementieren seien und nur einen sehr geringen Teil an möglichen Pfaden überdecken können.

Integration Tests Schaffer differenziert Integrationstests und Integrated Tests. Integrationstests unterscheiden sich von Integrated Tests in der Hinsicht, dass sie die Korrekheit eines Microservices isoliert überprüfen. Integrationstests fokussieren sich explizit auf die Interaktionspunkte mit anderen Services und testen diese. Die Integrationstest betrachten einen Microservice als Black-Box, Details über die Implementierung werden nicht weiter betrachtet. Der Service wird von seinen Abhängigkeiten isoliert, in dem andere Services gemockt werden. Eine oder mehrere Datenbanken mit Testdaten werden in der Testumgebung gestartet, die Schnittstellen des zu testenden Microservices werden aufgerufen und die Rückgabewerte werden mit erwarteten Rückgabewerten verglichen. Die Betrachtung eines Microservices als Black-Box erlaubt das Ändern von Implementationen, ohne das dadurch Tests geändert werden müssen. Auch die Datenbank lässt sich austauschen, da für die Tests entscheidend ist, ob die Schnittellen nach wie vor die korrekten Rückgabewerte bei Aufrufen zurückliefern. Der Nachteil bei diesem Vorgehen ist ein geringer Geschwindigkeitsverlust bei der Testausführung. Die einzelnen Integration Tests dauern zwischen Millisekunden bis hin zu einigen Sekunden pro Test, was allerdings aufgrund des schnelleren, iterativen Vorgehens bei der agilen Softwareentwicklung durch das Wegfallen vieler Testanpassungen einen Vorteil darstellt. Eine

weitere Schwierigkeit ist ein Verlust an Genauigkeit bei fehlgeschlagenen Testfällen. Dadurch, dass die Implementation selbst nicht getestet wird, sondern lediglich die Ein- und Ausgaben über die Schnittstellen nach Black-Box-Verfahren, werden Stacktraces von Fehlerzuständen notwendig um Fehlwirkungen nachvollziehen zu können.

Implementation Detail Tests Implementation Detail Tests sind nach Schaffer Tests, die bestimmte Codeteile, welche eine eigene interne Komplexität besitzen, im White-Box-Verfahren isoliert überprüfen. Als Beispiel für einen Implementation Detail Test nennt Schaffer das Parsen einer Log-Datei aus einem CI-Tool. Durch einen Integration Test wird überprüft, ob alle relevanten Felder in dem Rückgabewert bei Auftreten eines Fehlerzustandes vorhanden sind, nicht aber die Werte der Felder selbst. Implementation Detail Tests decken verschiedene Fehlerszenarien ab, die in der Logdatei analysiert werden sollen. Sie überprüfen, ob diese korrekt geparst werden können und ob die Werte der einzelnen Felder in der Fehlernachricht dem individuellen Fehlerszenario entsprechen. Implementation Detail Tests sind ähnlich zu Cohns Definition von Unit-Tests. Sie sind kleine, schnell ausführbare Tests, die einen einzelnen Codeabschnitt in Isolation überprüfen. Der Unterschied ist jedoch, dass nach Ansicht Schaffers nicht eine möglichst hohe Abdeckung auf Codeebene mit dieser Art von Tests erreicht werden soll, sondern sie nur an bestimmten Stellen im Code, die eine gewisse Komplexität aufweisen, eingesetzt werden sollen.

3.3 Vergleich

In diesem Abschnitt werden Gemeinsamkeiten und Unterschiede der vorgestellten Testkonzepte dargestellt. Hierbei betrachtet die Arbeit verschiedene Ebenen für das Testen eines Microservice-basierten Systems und wie die einzelnen Testkonzepte diese abbilden. Diese werden anschließend in Abschnitt 3.4 bewertet.

3.3.1 Internes Testen eines Microservices

Für das interne Testen eines Microservices verfolgen alle vier Testkonzepte einen ähnlichen Ansatz. Nach Wolff, Newman und Clemson soll durch einen Unit-Test die kleinste Einheit eines Systems getestet werden, oft auf Ebene von einzelnen Funktionen oder Methoden. Ein Ausnahme macht dabei der Implementation Detail Test (siehe Abschn. 3.2.4) nach Schaffer, der nur bestimmte Codeteile, welche eine eigene interne Komplexität besitzen, im White-Box-Verfahren isoliert überprüft.

Testkonzept von	Komponententests	Anzahl Testfälle
Wolff	Unit Tests	Viele
Newman	Unit Tests	Viele
Clemson	Solitary / Sociable Unit Tests	Viele
Schaffer	Implementation Detail Tests	Wenig

Tabelle 3.1: Vergleich von Teststufen für das interne Testen eines Microservices innerhalb einer Microservice-Architektur

Clemson nimmt eine Unterteilung der Unit-Tests vor, in dem er zwischen Sociable und Solitary Unit Tests differenziert. Ein Solitary Unit Test nach Clemson folgt der Auffassung von Wolff und Newman, die das Testen der kleinstmöglichen Einheit eines Systems in Isolation vorsieht. Clemson ist jedoch der Ansicht, dass es in bestimmten Situationen auch Sinn machen kann, die kleinsten Einheiten zu integrieren, was er in Form von Sociable Unit Tests genauer darstellt. Dieser Auffassung ist auch Schaffer, welcher vorschlägt, durch Implementation Detail Tests einzelne oder eine Menge von kleinen Einheiten eines Services mit eigener Komplexität zu überprüfen.

Wolff, Newman und Clemson stellen die Unit-Tests als breites Fundament ihrer Pyramidenmodelle dar. In allen drei Testkonzepten soll die Codebasis möglichst umfassend durch eine große Menge an Unit-Tests abgedeckt werden. Im Gegensatz dazu steht Schaffer, der in seiner „Microservice Testing Honeycomb“ (siehe Abschn. 3.8) nur wenige Implementation Detail Tests empfiehlt. Wolff, Newman und Clemson sind dabei der Ansicht, dass das vergleichsweise günstige Implementieren von vielen Unit-Tests eine große Anzahl an Fehlern auf den untersten Ebenen abfangen kann. Schaffer dagegen ist der Meinung, dass zu viele Unit-Tests Änderungsmöglichkeiten am Code einschränken und dass das Anpassen von Unit-Tests nach Codeänderungen einen negativen Einfluss auf die Geschwindigkeit in der agilen Softwareentwicklung besitze.

Über die Testlaufzeit von Unit-Tests sind sich Wolff, Newman und Clemson einig, dass ein einzelner Unit-Test nur wenige Sekunden lang ausgeführt werden sollte. Zu der Laufzeit von Implementation Detail Tests äußert sich Schaffer nicht.

3.3.2 Testen eines Microservices und dessen Abhängigkeiten

Die vorgestellten Testkonzepte unterscheiden sich voneinander in der Art und Weise, wie Microservices und ihre Abhängigkeiten getestet werden sollten.

Testkonzept von	**Integrationstests**			
Wolff	Integrationstests	CDCT	UI Tests	Manuelles Testen
Newman	Service Tests		CDCT	
Clemson	Persistence / Gateway Integration Tests	Komponententest	CDCT	
Schaffer	Integration Tests		Integrated Tests	

Tabelle 3.2: Vergleich von Teststufen für das Testen eines einzelnen Microservices und dessen Abhängigkeiten

Wolff ist der Ansicht, dass durch Integrationstests auf Ebene eines Microservices die korrekte Interaktion eines Services mit seinen Abhängigkeiten überprüft werden sollte. Für die Durchführung von Integrationstests werden die Abhängigkeiten eines Services isoliert und mit Stubs simuliert.

Auch Newman beschreibt ein ähnliches Vorgehen in seinen Service-Tests, welche einen einzelnen Microservice und dessen Funktionalitäten in Isolation zu anderen Services testen, die Benutzerschnittstelle des Gesamtsystems umgehen und Funktionen unmittelbar am Service direkt aufrufen. Ähnlich zu Wolff nutzt Newman hierfür sogenannte Platzhalterservices, die er in Mocks und Stubs unterteilt. So können Abhängigkeiten nach Newman auch in Form von Mocks simuliert werden, er empfiehlt aber wie bei Wolff den überwiegenden Einsatz von Stubs.

Clemson unterscheidet Integrationstests, welche er in Gateway- und Persistence Integrationtests, sowie Komponententests unterteilt. Integrationstests testen explizit die Komponenten eines Microservices, die mit externen Diensten kommunizieren. Komponententests überprüfen einen Microservice selbst als eine zusammenhängende Einheit innerhalb des Gesamtsystems in Isolation zu seinen Abhängigkeiten und testen sein Verhalten. Letzteres ähnelt dem Vorgehen bei Wolffs Integrationstests und Newmans Service Tests. Clemson wird auch hier detaillierter, in dem er zwischen In-Process und Out-Of-Process Verfahren bei Komponententests als Vorgehensweise zur Isolation eines Services von seinen Abhängigkeiten unterscheidet.

Schaffer ist ähnlicher Ansicht wie Wolff, Newman und Clemson. Auch er fordert das Durchführen von Integrationstests durch Isolation der Abhängigkeiten eines Services. Anders als Clemson beispielsweise, betrachtet er einen Microservice selbst als Black-Box. Durch Integration Tests werden explizit die Interaktionspunkte mit anderen Services getestet, nicht aber deren detaillierte Implementation. Anders als Wolff, Newman und Clemson, die auf Ebene eines Microservices dessen Isolation von seinen Abhängigkeiten fordern, kann es nach Schaffer sinnvoll sein, in bestimmten Fällen Integrated Tests zu implementieren. Integrated Tests, welche auf Grundlage der Korrektheit eines anderen Systems bestanden oder nicht bestanden werden, erfordern das Starten anderer Services in einer Testumgebung. Es werden Integrationstests mit nicht-simulierten Abhängigkeiten durchgeführt. Schaffer rät hiervon ab und verweist auf eine große Menge an Integration Tests.

Bei der Anzahl an Testfällen für Integrationstest auf Ebene eines Microservices unterscheidet sich Schaffer von Wolff, Newman und Clemson. Wo letztere zu einer großen Anzahl an Unit-Tests raten und die Anzahl an Testfällen für Integrationstests gegenüber vorherigen Teststufen abnimmt, setzt Schaffer auf eine große Menge an Integration Tests.

Wolff empfiehlt Consumer-Driven Contract Tests als Egänzung zu Integrationstests. Auch Newman nutzt CDCTs in seiner Testpyramide und nennt diese explizit in der Teststufe von Service Tests. Im Vergleich zu Wolff sieht Newman CDCTs nicht nur als eine Ergänzung für Integrationstests, sondern empfiehlt die Implementation eines großen Anteils von Testfällen als solche. Clemson ist der Ansicht, dass Unit-Tests und CDCTs als automatisierte Schritte in der Deployment-Pipeline anstelle von Integrationstests genutzt werden sollten und Integrationstest stattdessen nach dem Durchlaufen der Pipeline ausgeführt werden können. Schaffer erläutert den Einsatz von CDCT in seinem Testkonzept nicht detailliert, gibt aber einen Ausblick darauf, dass sie ergänzend zu Integration Tests und Integrated Tests genutzt werden können.
Gegenüber Newman, Clemson und Schaffer, nennt Wolff UI-Tests und manuelles Testen als zwei Teststufen auf Ebene eines Microservices. Er empfiehlt das Testen der Benutzeroberfläche eines Services in Isolation zu dessen Abhängigkeiten und erläutert, dass manuelles Testen auf Ebene eines Microservices möglich sei um Fehler auf dieser Ebene zu ermitteln.

3.3.3 Testen des Gesamtsystems

Das Gesamtsystem einer Microservice-Architektur wird von den vorgestellten Testkonzepten unterschiedlich überprüft.

Testkonzept von	Integrationstests	Systemtests	
Wolff	Gemeinsame Integrationstests	UI-Tests	Manuelles Testen
Newman	-	End-To-End Tests	
Clemson	-	End-To-End Tests	Exploratives Testen
Schaffer	-	-	

Tabelle 3.3: Vergleich von Integrationstests und Systemtests für das Testen des Gesamtsystems in einer Microservice-Architektur

In Wolffs Testkonzept existiert eine zusätzliche Teststufe für gemeinsame Integrationstests. Dabei darf jeweils nur ein geänderter Microservice zur Zeit in der gemeinsamen Testumgebung gestartet und mit den aktuellen Versionen anderer Services getestet werden. In dieser gemeinsamen Stufe in der Deployment-Pipeline werden alle Services deployed. Damit müssen deren Abhängigkeiten nicht simuliert werden. Die anderen Testkonzepte verzichten auf die Durchführung von Integrationstests, für die das gesamte System in einer Testumgebung gestartet werden muss. Newmans Testkonzept sieht ebenfalls eine gemeinsame Stufe innerhalb der Deployment-Pipeline für alle Services vor. Jedoch werden in dieser Stufe keine gemeinsamen Integrationstests ausgeführt, sondern End-To-End Tests. End-To-End Tests nach Newman testen das gesamte System, indem sie die über die Benutzerschnittstelle bereitgestellte Funktionalitäten überprüfen und sich beim Testentwurf an diesen orientieren. Clemson definiert in seinem Testkonzept ebenfalls End-To-End Tests als eine Teststufe. Im Vergleich zu Newman orientieren sich diese jedoch an User Stories und Personas.

Wolff und Clemson erläutern Manuelles und Exploratives Testen als Teststufen in ihren Testkonzepten. Wolff ist der Ansicht, dass durch manuelle Tests neue Funktionalitäten getestet werden, die noch nicht in der Produktivumgebung zu finden sind. Er merkt jedoch an, dass manuelles Testen die Veröffentlichung eines Services verlangsamen kann. Daher sollten Microservices auch ohne die Durchführung von manuellen Tests in die Produktion gegeben werden können, um schnelle Releasezyklen nicht zu blockieren. Clemson hingehen nutzt Exploratives Testen, um die gewonnenen Erfahrungen in den Entwurf und die Weiterentwicklung von geeigneten End-To-End Tests einfließen zu lassen. Dabei soll sich auf Aspekte konzentriert werden, die bisher nicht durch automatisierte Tests abgedeckt worden sind.

Wolff nennt in seinem Testkonzept die UI-Tests als Teststufe, welche explizit die Benutzeroberfläche testet. Newman und Clemson nennen die Benutzeroberfläche in ihrer Definition von

End-To-End Tests, erwägen aber auch das Testen über öffentliche Serviceschnittstellen.

Clemson erläutert, dass es bei externen Komponenten und Diensten in bestimmten Fällen sinnvoll sein kann, diese bei End-To-End Tests zu simulieren. Wolff, Newman und Clemson sind sich einig, dass Systemtests für das Gesamtsystem in einer Microservice-Architektur sehr aufwendig zu implementieren sind und leicht zu ungenauen Testergebnissen führen können. Aufgrund dessen und der langen Ausführungsdauer, die alle drei im Stundenbereich ansiedeln, empfehlen sie möglichst wenig Systemtests für Microservices zu implementieren. Schaffer geht einen Schritt weiter und verzichtet in seinem Testkonzept vollständig auf die Durchführung von Systemtests.

3.3.4 Nutzung ergänzender Testverfahren

Einige der vorgestellten Testkonzepte erläutern ergänzende Testverfahren, welche zusätzlich zu den Teststufen Probleme und Fehler in einer Microservice-Architektur aufdecken sollen.

Unterteilung des Testkonzeptes Sein Testkonzept unterteilt Wolff in zwei unterschiedliche Abschnitte. Es existiert jeweils ein Pyramidenmodell für den einzelnen Microservice und ein Pyramidenmodell für das Testkonzept des Gesamtsystems. Newman, Clemson und Schaffer nehmen keine explizite Unterteilung ihres Testkonzeptes und Modells vor.

Logging & Monitoring Wolff erläutert den Einsatz von Logging und Monitoring, um Fehler nach der Veröffentlichung zu ermitteln. Er setzt dabei auf eine zentrale Logging-Instanz, welche die Interaktionen zwischen den einzelnen Services protokolliert. Mithilfe von Metriken und Monitoring kann bei fehlerhaftem Verhalten alarmiert werden. Auch Newman nennt den Einsatz von Monitoring in Produktion, um Eigenschaften wie Reaktionszeit, Latenzen, Verfügbarkeit und Datenverlust zu überwachen (vgl. Newman, 2016, S.55). Anders als bei Newman, der das Testen nach Veröffentlichung als Teil seines Testkonzeptes betrachtet, ist dies bei Wolff nicht der Fall.

Lasttests Wolff und Newman schlagen Lasttests als zusätzliches Werkzeug für das Überprüfen von Microservices vor. Wolff unterscheidet im Vergleich zu Newman zwischen Performanz- und Kapazitätstests. Newman nennt dabei nur den Begriff des Performanztests, ohne weitere Ausführung seines Verständnisses. Er ist der Ansicht, dass Performanztests möglichst frühzeitig in der Entwicklung angesetzt werden sollten, um Probleme aufzudecken. Sie sollen in

regelmäßigen, kurzen Abständen zueinander ausgeführt werden. Clemson und Schaffer treffen keine Aussage über den Einsatz von Lasttests.

3.3.5 Deployment und Veröffentlichung

Einige der vorgestellten Testkonzepte machen Empfehlungen für das Deployment und die Veröffentlichung von Microservices. Die wesentlichen Punkte werden in den folgenden Abschnitten dargestellt.

Testautomatisierung In vielen Fällen werden Testautomatisierung und CI-Tools genutzt, um Microservices zu überprüfen und bereitzustellen. Wolff und Newman geben dabei einen konkreten Aufbau für eine Deployment-Pipeline vor, die verschiedene Tests aus unterschiedlichen Teststufen automatisiert und somit schnell Feedback an das jeweilige Entwicklerteam gibt. Clemson und Schaffer erläutern die Verwendung von Automation für Testfälle nicht explizit in ihren Testkonzepten und geben keinen Aufbau vor.

Gemeinsame Teststufe für Integrations- / End-To-End Tests Wolff und Newman definieren jeweils eine gemeinsame Stufe innerhalb der Deployment-Pipeline, in der alle Microservices gestartet werden und in der das Gesamtsystem bei Wolff durch Integrationstests und bei Newman durch End-To-End Tests überprüft wird. Die einzelnen Microservices besitzen jeweils eine eigene Deployment-Pipeline, in der Komponententests und servicespezifische Arbeitsschritte automatisiert werden.

Deployment-Strategien Newman betont die Bedeutung des Testens von funktionsübergreifenden Anforderungen nach der Veröffentlichung. Für ihn ist das Testen eines neuen Microservices im laufenden Betrieb wichtig, weshalb er Deployment-Strategien wie das Blue-/Green Deployment und das Canary-Releasing empfiehlt und sie als Teil seines Testkonzeptes ansieht. Wolff, Clemson und Schaffer machen dazu keine Aussage.

3.4 Zusammenfassende Bewertung

Durch die verschiedenen Unterteilungen der einzelnen Teststufen und ihrer Gewichtung in den vorgestellten Testkonzepten, ergeben sich Vor- und Nachteile.

Wolff nimmt eine explizite Unterteilung seines Testkonzeptes in Testen eines Microservices als Komponente und Microservices als Gesamtsystem vor. Hierdurch wird deutlich, welche Teststufen er für die Microservices und welche er für das Gesamtsystem für geeignet hält.
Wolff, Newman und Clemson stimmen in ihrer Auffassung über eine möglichst große Menge an Unit-Tests auf unterster Ebene ihres Testkonzeptes überein. Schaffer plädiert im Gegensatz dazu dafür, Unit-Tests nur an Stellen die eine gewisse Komplexität aufweisen einzusetzen. Allerdings können sich auch in Codeabschnitten, die vergleichsweise wenig komplex sind, Fehler durch menschliches Versagen einschleichen. Durch die Beschränkung von Komponententests auf lediglich komplexe Codeabschnitte besteht das Risiko, dass Fehler in anderen Codeabschnitten nicht frühzeitig entdeckt werden und zu einer Fehlerwirkung einer Komponente innerhalb eines Microservices führen. Unter Umständen kann der Aufwand, welcher für Fehlerlokalisierung und Behebung aufgewendet wird, den Aufwand für das Formulieren eines geeigneten Unit-Tests übersteigen.

Auf Ebene eines Microservices sind sich alle Autoren einig, dass das Testen eines Services in Isolation zu seinen Abhängigkeiten geschehen muss. Wo Wolff, Newman und Schaffer einen Microservice als Black-Box betrachten, testet Clemson in Form von Persistence- und Gateway Integration Tests einzelne Komponenten für externe Kommunikation und Persistierung von Daten innerhalb eines Microservices explizit. Erst in einer folgenden Teststufe wird der Service als Black-Box betrachtet und das Zusammenspiel mit seinen Abhängigkeiten überprüft. Durch die zusätzliche Teststufe die Clemson definiert, können Fehler auf Protokollebene und der Persistierung von Daten frühzeitig entdeckt werden, die bei funktionalen Integrationstests in Form eines Black-Box Verfahrens schwieriger zu lokalisieren wären.
Wolff, Newman und Clemson nutzen Consumer Driven Contract Tests für das Testen eines Microservices in Isolation und als Mittel für Regressionstests. Diese Tests geben ein hohes Maß an Vertrauen in die Testsuite, während sie die Implementierungsdetails entkoppeln und komplexe Integrationstests, die das Starten vieler Services voraussetzen, vermeiden. Der Vorteil dieser Teststufe als Ergänzung zu Integrationstests liegt darin, dass sie vergleichsweise schnell, zuverlässig, unabhängig und auch lokal durchgeführt werden kann. Schaffer schlägt vor auf breiter Ebene Integrationstests für einen einzelnen Microservice in Isolation durchzuführen und gibt einen Ausblick auf den Einsatz von CDCT nur als eine Ergänzung dieser Teststufe.

Wolff schlägt im Gegensatz zu den anderen Autoren in seinem Testkonzept für einen einzelnen Microservice UI Tests und Manuelles Testen als zusätzliche Teststufen vor. Die UI-Tests finden sich auch im Testkonzept auf Ebene des Gesamtsystems von Wolff wieder. Abhängig

von der Bedeutung der Benutzeroberfläche für das Gesamtsystem, kann es sinnvoll sein, auf Ebene eines Microservices diese mit simulierten Abhängigkeiten zu überprüfen. Durch die Isolierung von Abhängigkeiten bei UI-Tests ergeben sich Vorteile. Durch den Zugang zu einem Microservice als Einzelkomponente können bestimmte Fehlerwirkungen besser provoziert und Fehlerzustände einfacher lokalisiert werden. Bei UI-Tests auf Ebene des Gesamtsystems kann es schwierig sein, den Entstehungsort und die Ursache für eine Fehlerwirkung einzugrenzen. Durch die Isolation von Abhängigkeiten gelingt dies bei UI-Tests auf Ebene eines einzelnen Microservices hingegen leichter. Allerdings können Wolffs zusätzliche Teststufen die Geschwindigkeit der Veröffentlichung eines neuen Services einschränken, was für Schaffer ein wichtiger Aspekt im Kontext von Microservice-Architekturen ist.

Für das Testen des Gesamtsystems schlägt Wolff gemeinsame Integrationstests als Teststufe vor. Er empfiehlt vor dem Deployment eines geänderten Services eine gemeinsame Teststufe für Integrationstests, in der alle Microservices gestartet und überprüft werden. Dies hat den Vorteil, dass Microservices mit nicht-simulierten Abhängigkeiten getestet werden können und die Testumgebung der Produktivumgebung ähnelt. Abhängig von der gewählten Teststrategie für gemeinsame Integrationstests kann es aufgrund der vielen verschiedenen Abhängigkeiten innerhalb einer Microservice-Architektur jedoch sehr aufwendig sein, Mocks oder Testtreiber für Services zu implementieren. Newman, Clemson und Schaffer verzichten auf Integrationstests, für die das Gesamtsystem in einer gemeinsamen Testumgebung gestartet werden muss und nutzen wie zuvor beschrieben Integrationstests für einzelne Microservices in Isolation. Newman und Clemson nutzen End-To-End Tests als Möglichkeit für Systemtests, die das Gesamtsystem in einer gemeinsamen Testumgebung auf funktionale Aspekte überprüfen. Newman und Clemson stimmen überein, dass diese Tests leicht fehlschlagen und ungenaue Testergebnisse liefern können. Dadurch entsteht die Forderung, die Anzahl an Testfällen für End-To-End Tests niedrig zu halten.
Clemson nutzt Exploratives Testen als Teststufe für das Gesamtsystems als Mittel zur Formulierung und Verbesserung von End-To-End Tests, während Wolff manuelles Testen als Möglichkeit für das Entdecken von Fehlern nutzt, die schwierig zu automatisieren sind. Beide Vorgehensweisen sind valide und können zusätzliche Sicherheit geben.
Schaffer verzichtet komplett das Testen des Gesamtsystems. Obwohl gemeinsame Integrationstests aufwendiger zu implementieren sind, liefern sie jedoch ein großes Maß an Vertrauen, dass die einzelnen Microservices auch mit nicht-simulierten Abhängigkeiten korrekt interagieren können. Ähnliches gilt für Systemtests, denn sie garantieren, dass das Gesamtsystem seine

funktionalen Anforderungen erfüllt.

Wolff und Newman definieren eine gemeinsame Endstufe innerhalb einer Deployment-Pipeline für alle Services. Diese Endstufe stellt einen kritischen Abschnitt dar, in dem nur ein Service zur Zeit getestet werden darf. Dieses Vorgehen hat den Vorteil, dass die benötigten Ressourcen für die Durchführung von gemeinsamen Integrationstests bei Wolff und gemeinsamen End-To-End Tests nach Newman begrenzt werden, da nicht mehrere Testumgebungen gleichzeitig mit dem Gesamtsystem gestartet werden müssen. Allerdings besitzt die gemeinsame Endstufe auch den Nachteil, dass die Geschwindigkeit der Veröffentlichung von neuen Microservices deutlich eingeschränkt wird und zusätzliche Mechanismen in Form von Mutex-Verfahren geschaffen werden müssen, um das Problem des kritischen Abschnittes zu lösen. Newmans Testkonzept hat den Vorteil, dass es das Testen eines Microservices im Betrieb miteinbezieht. Durch die Beachtung von verschiedenen Deployment-Strategien, kann die Fehlerwirkung von unentdeckten Fehlerzuständen innerhalb der Deployment-Pipeline in der Produktivumgebung begrenzt werden.

Zusammenfassend lässt sich sagen, dass alle Testkonzepte auf ihre eigene Art versuchen, mit den Herausforderungen beim Testen von Microservices umzugehen. Alle Testkonzepte enthalten ähnliche Vorgehensweisen für das interne Testen eines Microservices, den Umgang mit dessen Abhängigkeiten und das Testen des Gesamtsystem. Schaffers Ansatz unterscheidet sich in besonderer Weise von den Anderen, denn sein Testkonzept fokussiert sich fast ausschließlich auf Integrationstests, die das Zusammenspiel und die Interaktion der Services untereinander überprüfen und einen Microservice als Black-Box betrachten. Wolff, Newman und Clemson hingegen legen einen großen Wert auf das interne Testen der Implementation eines Microservices und das Testen des Gesamtsystems mit seinen funktionalen Anforderungen.

4 Entwicklung und Implementierung eines Testkonzeptes

In diesem Kapitel wird ein Testkonzept basierend auf den vorgestellten Testkonzepten des vorherigen Kapitel 3 entwickelt und anhand eines Versuchssystems evaluiert.

4.1 Das Versuchssystem

In diesem Abschnitt wird das Versuchssystem vorgestellt. Dazu wird in einer Kurzbeschreibung das Anwendungssystem mit seinen Anforderungen beschrieben. Anschließend wird die umgesetzte Microservice-Architektur dargestellt und auf die einzelnen Microservices als Komponenten im Detail eingegangen.

Die Anwendungsdomäne für das Versuchssystem muss dabei ein abgrenzbares Problemfeld für die Definition von Systemanforderungen und den technischen Entwurf eines geeignetes Anwendungssystems bieten.

4.1.1 Kurzbeschreibung

Als Versuchssystem für die Entwicklung und Implementierung eines Testkonzeptes wird ein Microservice-basiertes Eventmanagement System verwendet. Das System deckt die folgenden drei Aufgabenbereiche ab:

1. *Organisation von Veranstaltungen*
2. *Verwaltung von Tickets und Vouchern*
3. *Erfassung von Zutritten*

Das System ermöglicht es, Veranstaltungen unterschiedlicher Größe abzubilden. Eine Veranstaltung kann Unterveranstaltungen wie zum Beispiel Workshops oder Vorträge beinhalten,

die das System ebenfalls darstellen kann. Zu den Veranstaltungen können Tickets und Voucher erworben werden. Ein Voucher ist ein Gutschein, der sich gegen ein entsprechendes Ticket für eine Veranstaltung eintauschen lässt. Während einer Veranstaltung können Zutritte registriert und überprüft werden. Ein Zutritt zu einer Veranstaltung ist nur mit einem gültigen Ticket möglich, das innerhalb des Veranstaltungszeitraums liegt.

Aus den genannten Aufgabenbereichen des Eventmanagement-Systems lassen sich fachliche Zusammenhänge ableiten, die für einen geeigneten Komponentenschnitt in einer Microservice-Architektur genutzt werden können.

4.1.2 Architektur

Das Versuchssystem besteht aus drei Microservices, die miteinander interagieren. Jeder einzelne Microservice kapselt einen eigenständigen Teil der Geschäftslogik des Gesamtsystems und wird in einem Docker-Container in einem Kubernetes-Cluster orchestriert. Als Persistenzlösung wird PostgreSQL als freies, objektrelationales Datenbankmanagementsystem verwendet. Die Datenbank wird in dem Versuchssystem von allen Microservices gemeinsam für die Persistierung von Daten verwendet, mit dem Bewusstsein, dass der Architekturstil andere Varianten in dieser Hinsicht zulässt.

Client Ein Client ist ein Endbenutzer, der über einen Schnittstellen-Aufruf eine gewünschte Funktionalität anfordert. Ein Schnittstellen-Aufruf kann sowohl über REST, als auch als Remote Procedure Call ausgeführt werden. Als Datenformat wird bei REST die JavaScript Object Notation via HTTP(S), beziehungsweise Protocol Buffers zur Serialisierung bei RPC verwendet.

API-Gateway Zwischen dem Client und den Microservices wird ein API-Gateway geschaltet, das in Form einer Fassade dem Client eine vereinfachte Schnittstelle zu der Menge von Schnittstellen der einzelnen Microservices bereitstellt. Das API-Gateway trägt zu einer losen Kopplung zwischen Systemaufrufern und den dahinter versteckten Subsystemen in Form von Microservices bei. Gleichzeitig senkt es die Komplexität des Anwendungssystems, indem die Schnittstellen der Services zu einer Gemeinsamen zusammengefasst werden. Als Proxy zwischen Client und Microservices erlaubt das API-Gateway auch das Implementieren von Middleware für beispielsweise Authentifizierung oder Logging.

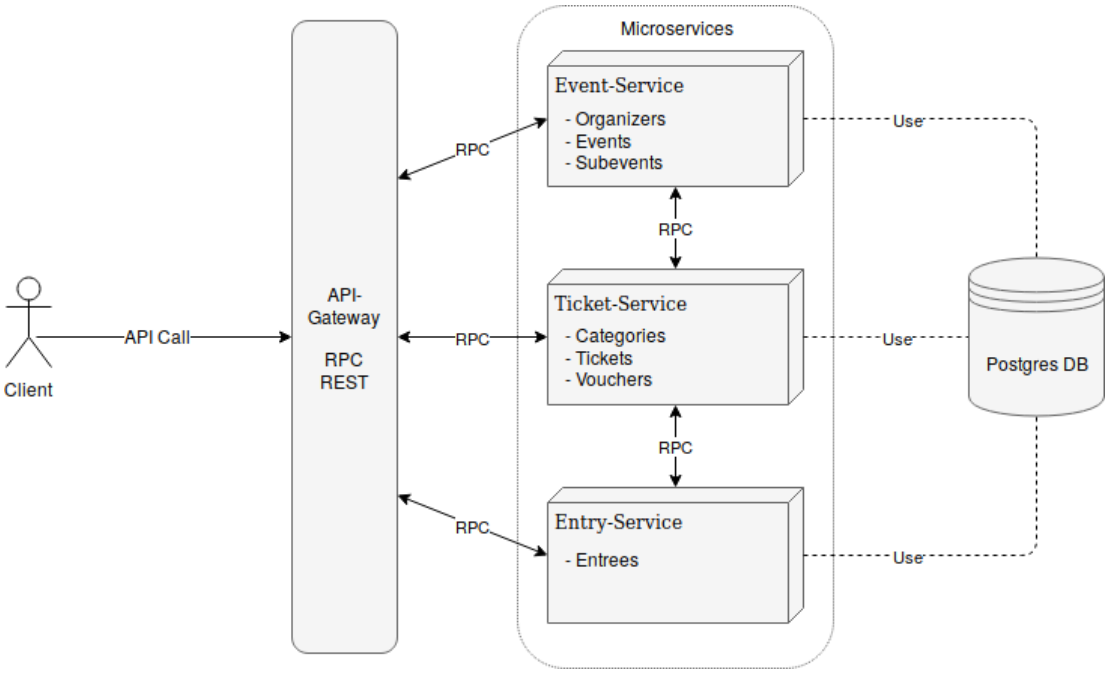

Abbildung 4.1: Überblick über die Architektur des Versuchssystems. Die Microservices und das API-Gateway wurden in der Programmiersprache Go implementiert und nutzen „micro/go-micro“ als RPC-Framework für die Entwicklung verteilter Systeme.

Event-Service Der Event-Service verwaltet die Ressourcen Organisatoren, Events und Subevents. Der Service bietet bietet simple CRUD-Operationen für die Verwaltung der Ressourcen an.

Ticket-Service Der Ticket-Service verwaltet die Ressourcen Ticketkategorien, Tickets und Vouchers. Der Ticket-Service ist direkt abhängig von dem Event-Service, denn um ein Ticket zu erstellen muss die entsprechende Veranstaltung existieren.

Entry-Service Der Entry-Service verwaltet Zutritte als Ressourcen. Der Entry-Service ist direkt abhängig von dem Ticket-Service und transitiv abhängig vom Event-Service. Um einen Zutritt zu registrieren muss geprüft werden, ob zu einer gegebenen Ticketnummer ein Ticket existiert und ob es eine Veranstaltung mit einem Veranstaltungszeitraum, die das aktuelle Datum enthält, gibt.

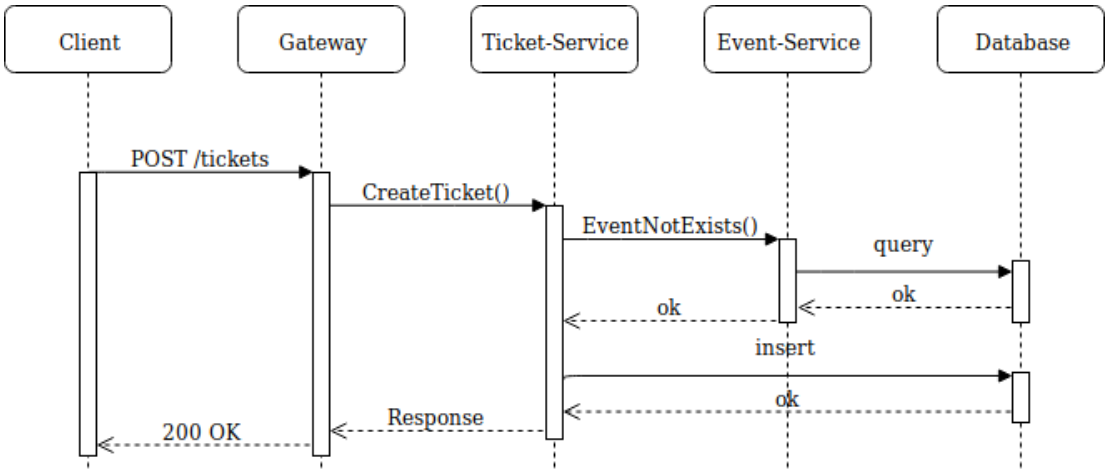

Abbildung 4.2: Sequenzdiagramm für das erfolgreiche Erstellen einer neuen Ticket-Ressource im Versuchssystem. Eine Ticket-Ressource wird durch die Zusammenarbeit des Ticket- und Event-Services realisiert.

4.2 Testkonzept

In diesem Abschnitt wird ein Testkonzept für das Microservice-basierte Versuchssystem vorgestellt. Dazu werden verschiedene Anforderungen an das Testkonzept gestellt, die es zu erfüllen gilt. Daraufhin wird ein Modell vorgestellt, das die einzelnen Teststufen und die Verteilung von Testfällen beschreibt. Anschließend folgen Ergänzungen zum Testkonzept, die zusätzliche Aspekte für das Validieren der Microservices miteinbeziehen.

4.2.1 Anforderungen

In dem vorherigen Kapitel 3 in dem Abschnitt 3.1.2 wurde bereits auf Herausforderungen für das Testen für Microservices eingegangen. Das Testkonzept muss mit den diskutierten Aspekten umgehen können:

- *Dokumentation von Abhängigkeiten und Schnittstellen*
- *Kommunikation und Interaktion zwischen Microservices*
- *Unabhängige Entwicklung von Microservices*
- *Unabhängige Veröffentlichung von Microservices*

Eine wichtige Anforderung an das Testkonzept ist das Ermöglichen der schnellen und unabhängigen Durchführung von Änderungen an einem Microservice. Das Testkonzept muss folgende Änderungen unterstützen:

1. *Integrieren* eines neuen Microservices in das Gesamtsystem
2. *Entfernen* eines bestehenden Microservices aus dem Gesamtsystem
3. *Ersetzen / Anpassen* eines bestehenden Microservices
   a) bei Anforderungsänderung
   b) bei Erweiterung um neue Features
   c) zur Behebung von gefundenen Fehlerzuständen

Das Testkonzept muss in angemessener Zeit ausgeführt werden können und Feedback an die Verantwortlichen geben. Das Testkonzept darf den Entwicklungsprozess nicht verlangsamen und kurze Releasezyklen von neuen Versionen der einzelnen Services nicht behindern.

Es gibt *Rahmenbedingungen* des Versuchssystems, die Auswirkungen auf die Anforderungen an das Testkonzept haben:

- Das Versuchssystem stellt keine grafische Benutzeroberfläche bereit. Stattdessen wird eine öffentliche API des Gesamtsystems angeboten, um unterschiedliche Clients zu bedienen.
- Die Microservices des Versuchssystem führen keine komplexen Berechnungen durch.
- Ein Großteil der Logik wird für das Persistieren von Ressourcendaten eingesetzt.

4.2.2 Eisberg-Modell

Für das Testen des Versuchssystems wurde ein Modell entwickelt, dass verschiedene Aspekte bestehender Testkonzepte miteinbezieht. In Anlehnung an Wolff (siehe Kap. 3, Abschn. 3.2.1) existiert sowohl ein Testkonzept für das Überprüfen eines einzelnen Microservices und seinen Abhängigkeiten als auch ein Weiteres für das Gesamtsystem. Aufgrund seines Aussehens wurde es „Eisberg-Modell“ getauft. Ähnlich zu einem Eisberg gibt es den Teil oberhalb und den nicht sichtbaren Teil unterhalb der Wasseroberfläche. Um den Eisberg als Ganzes zu erfassen, müssen beide Teile betrachtet werden. In einem Microservice-basiertem System verhält es

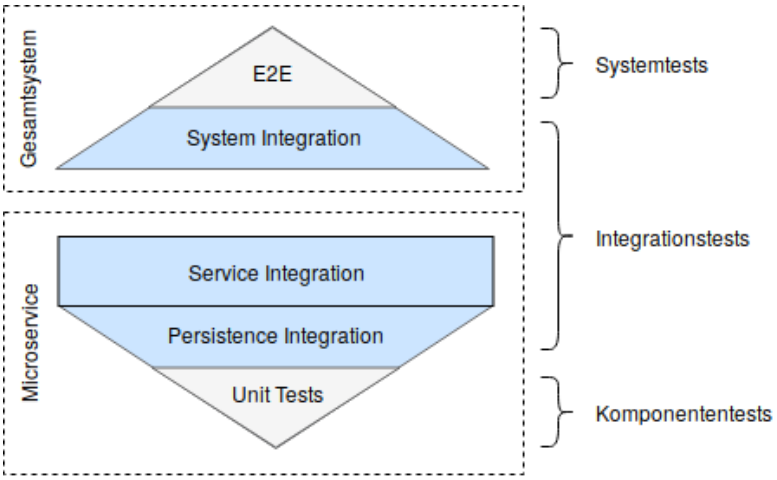

Abbildung 4.3: „Eisberg-Modell“ des Testkonzeptes für einen Microservice und das Gesamtsystem

sich ähnlich, denn um das System zu erfassen muss sowohl das Gesamtsystem, als auch der einzelne Microservice betrachtet werden.
Obwohl eine Unterteilung vorgenommen wird, bauen die beiden Testkonzepte aufeinander auf. Wird ein Fehler auf Ebene eines Microservices entdeckt, dann erfüllt der Microservice nicht seine Anforderungen und das Gesamtsystem muss somit nicht getestet werden. Besteht ein Microservice alle seine Tests und erfüllt seine Anforderungen, kann das Gesamtsystem getestet werden.

4.2.3 Testkonzept für einen Microservice

Das Testkonzept für einen Microservice unterscheidet die Teststufen Unit Tests, Persistence Integration und Service Integration. Auf die einzelnen Teststufen wird nachfolgend im Detail eingegangen.

Unit Tests Auf unterster Ebene des Testkonzeptes für einen Microservice befinden sich die Unit-Tests. Sie testen einzelne Methoden oder Funktionsaufrufe, welche die kleinsten Einheiten innerhalb eines Microservices darstellen, in Isolation. Unit-Tests stellen sicher, dass das jeweilige Testobjekt die laut seiner Spezifikation geforderte Funktionlität korrekt und vollständig umsetzt. Die Testbasis für den Unit Test ist deshalb die im Design festgelegte Spezifikation der jeweiligen Komponente und der Programmcode für die Implementierung.
Die Unit-Tests bilden wie in allen untersuchten Testkonzepten auch die Basis für das „Eisberg-Modell“. Dadurch, dass im Versuchssystem der Zugriff auf den Sourcecode möglich ist, können

die Unit-Tests als White-Box Tests durchgeführt werden. Durch die eingesetzten funktionalen Unit-Test auf der untersten Ebene des Testkonzeptes können Fehlerwirkungen, wie falsche Berechnungen oder inkorrekte Programmpfade nachgewiesen werden. Auch die Robustheit einer Komponente kann durch Unit-Tests überprüft werden, denn wenn eine Komponente mit ungültigen Parametern aufgerufen wird, sollte sich diese nicht beenden und den Microservice zum Absturz bringen. Stattdessen sollte die Komponenten auf Fehlersituationen robust reagieren. Um dies zu überprüfen enthält das „Eisberg-Modell“ auch Negativtests.

Auf höheren Teststufen kann es schwierig werden die Ressourcennutzung einzelner Komponenten innerhalb eines Microservices festzustellen oder ihre Wartbarkeit zu beurteilen. Deshalb wird im „Eisberg-Modell“ auf Ebene der Unit-Tests neben Funktionalität und Robustheit auch die Qualität einer Komponente getestet. Weil sich die Wartbarkeit nicht durch dynamisches Testen überprüfen lässt, wurde auf unterster Ebene des Testkonzeptes der Einsatz von Statischen Analysen (siehe 4.2.5.1) gewählt. Code-Reviews eignen sich aufgrund der vergleichsweise langen Vorbereitungsphasen nur bedingt im Rahmen der Unit-Tests bei Microservices und sind deshalb nicht Teil des Konzeptes.

Unit-Tests sind sehr entwicklungsnah und werden deshalb von den Entwicklern selbst geschrieben. Sie stellen sicher, das eine von dem Entwickler geschriebene Komponenten so arbeitet, wie diese es beabsichtigt. Das muss auch im Rahmen des „Eisberg-Modells“ sichergestellt sein.

Unit-Tests sollten nach Möglichkeit vollständig automatisiert werden. Dies entspricht dem Vorgehen bei Unit-Tests in den Testkonzepten von Wolff, Newman und Clemson. Es empfiehlt sich der Einsatz von Testframeworks für Unit-Tests, die ein einfaches Erstellen von automatisierten Testfällen für Komponenten ermöglichen. Die Wahl des Testframeworks ist dabei stark abhängig von der verwendeten Technologie des Microservices. Im Falle des Versuchssystems wurden die Microservices in der Programmiersprache Go entwickelt, weshalb als Testframeworks das Go-eigene „Testing“-Packet für Unit-Tests und das Behavior Driven Development Framework „Goblin“ genutzt wurde. Beide haben sich im Rahmen der Tests bewährt.
Für die Laufzeit eines Unit-Tests ist es sinnvoll, eine Begrenzung vorzunehmen. Das Testkonzept für einen Microservice orientiert sich dabei an den Aussagen Newmans, der wenige Sekunden Laufzeit für einen Unit-Test empfiehlt. (vgl. Newman, 2016, S.134). Überschreitet ein Unit-Test die Laufzeitbeschränkungen müssen Maßnahmen ergriffen werden, in denen entweder die Implementation des Testobjektes oder der Testfall selbst angepasst werden. Dies ist im Rahmen der durchgeführten Tests nicht aufgetreten. Bei dem spezifizierten Versuchssystem war das

aber auch nicht zu erwarten, da die Microservices keine komplexen Berechnungen durchführen.

Die Anzahl der Testfälle orientiert sich an Schaffers Idee der „Implementation Detail Tests“ (siehe Kap. 3, 3.2.4). Unit-Tests sollen für Funktionen und Methoden eingesetzt werden, die eine gewisse Komplexität besitzen. Weil die Microservices im Versuchssystem im wesentlichen nur klassische Methoden zur Manipulation der Ressourcen anbieten, gibt es kaum komplexe Berechnungen oder Verzweigungen die überprüft werden müssen. Diese Funktionen haben den Charakter eines Integrationstests, weshalb ausführlicher auf höheren Teststufen getestet werden sollte. Für vergleichsweise triviale Funktionen oder Methoden können Unit-Tests in diesem Testkonzept entfallen um die Entwicklungsgeschwindigkeit zu beschleunigen. Bei dem Entry-Service des Versuchssystems, welcher mit Zeitangaben kalkuliert, bot es sich beispielsweise an, einen Unit-Test für eine Überprüfung der korrekten Berechnung von Zeiträumen zu implementieren.

```
1  func TestIsDateBetween(t *testing.T) {
2      start, end := getTestPeriod()
3
4      toTest := time.Date(2018, 10,1, 12, 0, 0,0, time.UTC)
5      // Positive: test if date is in range, should return true
6      if got := isDateBetween(start, end, toTest); got == false {
7          t.Errorf("got %v, want %v", got, true)
8      }
9
10     toTest = time.Date(2017, 10,1, 12, 0, 0,0, time.UTC)
11     // Negative: test if date is in range, should return false
12     if got := isDateBetween(start, end, toTest); got == true {
13         t.Errorf("got %v, want %v", got, false)
14     }
15 }
```

Listing 4.1: Beispiel für einen Unit-Test einer Methode eines Microservices

Für die Teststufe der Unit-Tests gibt das „Eisberg-Modell“ bewusst keine Empfehlung über den Einsatz von Mocks oder Stubs zur Isolierung von Komponenten. Weil die Unit-Tests in der Verantwortung des jeweiligen Entwicklers liegen, soll dieser abhängig von dem konkreten Testfall über den Einsatz von einem Stub oder einem Mock entscheiden. Bei den Tests im Rahmen dieser Arbeit wurde auf Mocks oder Stubs auf Ebene der Unit-Tests verzichtet. Auch wird keine weitere Unterteilung der Unit-Tests wie beispielsweise bei Clemson in Sociable

und Solitary Unit Tests (siehe Kap. 3, 3.2.3) vorgenommen. Clemson geht in der Theorie von einer ähnlichen Struktur bei Microservices aus, die allerdings in der Praxis nicht immer zutrifft. Aufgrund der Entscheidung für eine strikte Isolierung der Testobjekte entfällt die Unterteilung.

Persistence Integration Tests Persistence Integration Tests spielen im Versuchssystem eine große Rolle, da ein Großteil der Logik für das Persistieren von Ressourcendaten eingesetzt wird. Sie testen das Zusammenspiel zwischen einem Microservice und einem externen Datenspeicher. Dabei haben sie das Ziel Mängel in der Abbildung der Ressourcen zwischen dem Microservice und der Datenbank aufzudecken. Die Persistence Integration Tests als zweite Ebene des Testkonzeptes für einen Microservice orientieren sich an den Integrationstests nach Clemson (siehe Kap. 3, Abschn. 3.2.3).

Testobjekte der Persistence Integration Tests sind die internen Komponenten eines Microservices, die für die Kommunikation mit der Datenbank zuständig sind. Durch diese Teststufe wird sichergestellt, dass die verwalteten Ressourcen eines Services korrekt persistiert und Fehlerzustände vor dem Testen der öffentlichen Service-Schnittstellen frühzeitig entdeckt werden können. Auch nicht funktionale Eigenschaften wie die Performanz im Zusammenhang mit der Persistierung von Daten werden explizit auf dieser Teststufe überprüft.

Clemson geht in seinem Testkonzept bei Persistence Integration Tests davon aus, dass Microservices dem Programmierparadigma der Objektorientierung folgen. Weil sich die Repräsentation der Daten innerhalb der Anwendung von dem Schema der Datenbank unterscheiden kann, wird diese Abweichung bei objektorientierten Anwendungen getestet. Oft werden OR-Mapping Frameworks verwendet, mit denen ein Microservice seine Objekte in einer relationalen Datenbank ablegen kann. Gleichzeitig wird versucht das Problem des „Impedance Mismatch“ zu bewältigen.
Durch Persistence Integration Tests als Teststufe des „Eisberg-Modells“ kann überprüft werden, ob das verwendete Framework zur objektrelationalen Abbildung korrekt und vollständig innerhalb des Microservices konfiguriert worden ist. Die Implementierung des Versuchssystems in Go bedingt nicht objektorientierte Microservices. Die Microservices im Versuchssystem verwenden auch kein Framework für die Persistierung der Daten, sondern kapseln die Statements innerhalb der einzelnen Komponente. Durch Persistence Integration Tests konnte hier die Korrektheit der Statements nachgewiesen und bei Fehlschlagen der Tests ein robuster Umgang mit der Fehlersituation überprüft werden.

Zuständig für die Implementation von Testfällen ist auch hier das jeweilige Entwicklerteam, das den Microservice entwickelt. Die Testfälle wurden in der Deployment-Pipeline des Versuchssystems vollständig automatisiert. Für die Durchführung wurde der zu überprüfende Microservice mit der Testdatenbank in einer gemeinsamen Testumgebung gestartet. Dabei wurde die Datenbank innerhalb eines Docker-Containers ausgeführt, um einheitliche Testbedingungen für die wiederholte Testausführung zu erzielen. Die Verbindung zur Testdatenbank wurde in der zu testenden Komponente konfiguriert und anschließend wurden die einzelnen Schnittstellen zur Persistierung von Ressourcen getestet.

```
1  func TestCreateOrganizer(t *testing.T) {
2      s := newTestClient()
3      defer func() {
4          s.Exec("delete from organizers")
5          s.Close()
6      }()
7      organizer := &events.Organizer{
8          OrganizerID:     1,
9          Name:            "HAW Hamburg",
10         Description:     "Hamburg University of Applied Sciences",
11         Website:         "https://haw-hamburg.de",
12     }
13     if err := s.CreateOrganizer(organizer); err != nil {
14         t.Error(err)
15     }
16     organizer := s.GetOrganizerByID(1); err != nil{
17         t.Errorf(err)
18     }
19     if got, want := organizer.ID, int64(1); got != want {
20         t.Errorf("Want organizer id %d, got %d", want, got)
21     }
22     ...
23 }
```

Listing 4.2: Beispiel für einen Persistence Integration Test eines Microservices zum Erstellen einer Organizer-Ressource

Die Kommunikation eines Microservices mit dem externen Datenspeicher sollte möglichst umfangreich mit Testfällen abgedeckt werden. Die Testfälle können sowohl im White-Box, als auch im Black-Box Verfahren von dem jeweiligen Entwicklerteam implementiert werden. Durch Persistence Integration Tests und deren Betrachtung der internen Komponente eines Services und des Datenspeichers als Teilsystem, können Mängel unabhängig von den öffentli-

chen Schnittstellen des Microservices lokalisiert werden. Für Microservices, die keine eigenen Ressourcen verwalten oder Persistierung benötigen, kann diese Teststufe der Persistence Integration Tests entfallen.

Clemson erwähnt als eine weitere Art von Integrationstest die Gateway Integration Tests (siehe Kap. 3, Abschn. 3.2.3), welche Fehler auf Protokollebene identifizieren. Diese wurden in das Versuchssystem nicht aufgenommen, da die zur Implementierung genutzten Frameworks Fehler dieser Art abfangen. Bei Bedarf können diese Tests auf der Teststufe der Service Integration Tests dieses Testkonzeptes implementiert werden. Dies kann z.B. mit manchen CDCT-Frameworks wie „Pact“ erfolgen, die das Kontrollieren von Protokollinformationen unterstützen.

Service Integration Tests Service Integration Tests sind im „Eisberg-Modells“ die oberste Teststufe auf Ebene eines Microservices. Sie betrachten einen einzelnen Microservice in Isolation zu seinen Abhängigkeiten und testen diesen in einem Black-Box Verfahren. Testobjekte sind dabei die Schnittstellen eines Microservices, über die der Service mit anderen Microservices kommuniziert. Ziel ist das Aufdecken von Fehlerzuständen in den Schnittstellen und dem Zusammenspiel zwischen dem zu testenden Microservice und seinen Abhängigkeiten.

Voraussetzung für die Durchführung von Service Integration Tests ist das erfolgreiche Bestehen von Unit- und Persistence Integration Tests, durch die eine korrekte Anwendungslogik innerhalb des Services und eine fehlerfreie Persistierung von Daten gewährleistet wird. Durch Service Integration Tests sollen verschiedene Typen von Fehlerzuständen sichtbar gemacht werden, wie zum Beispiel keine oder syntaktisch falsch übermittelte Daten, eine fehlerhafte Interpretation der übergebenen Daten oder die Identifikation von Timing-, Durchsatz- oder Lastproblemen.

Für das Testen der Schnittstellen eines Microservices wird dieser von seinen Abhängigkeiten isoliert. Dabei werden Mocks oder Stubs für die Simulation von Abhängigkeiten verwendet. Ähnlich zu dem Testkonzept von Newman wird eine Empfehlung für den Einsatz von Stubs gegeben, aufgrund ihrer Einfachheit (siehe Kap. 3, Abschn. 3.2.2). Um beispielsweise den Ticket-Service des Versuchssystems isoliert zu testen, muss der Event-Service als direkte Abhängigkeit durch einen Stub ersetzt werden. Gleiches gilt für den Entry-Service, dessen direkte Abhängigkeit der Ticket-Service darstellt.

Um geeignete Testfälle zu entwerfen zu können, müssen die Anforderungen an den Service dokumentiert sein. Eine öffentliche Schnittstellenbeschreibung ist daher wichtig und kann zum Beispiel mithilfe der OpenAPI Specification (OAS) durch Werkzeuge wie „Swagger“ generiert werden. Eine bei der Implementierung des Versuchssystems erstellte Schnittstellenbeschreibung zeigt Abbildung 4.4.
Des Weiteren muss definiert werden, welche Versionen der abhängigen Komponenten ein Microservice voraussetzt, um seine Funktionalitäten anzubieten. Das Wissen um die Abhängigkeiten liefert die Grundlage für das Erstellen geeigneter Stubs und Mocks. Zuständig für die Implementierung von Service Integration Tests ist das jeweilige Entwicklerteam des zu überprüfenden Microservices, da für die Erstellung von geeigneten Testfällen Hintergrundwissen über die Anforderungen an den Microservice notwendig ist und das jeweilige Entwicklerteam mit diesen am besten vertraut ist.

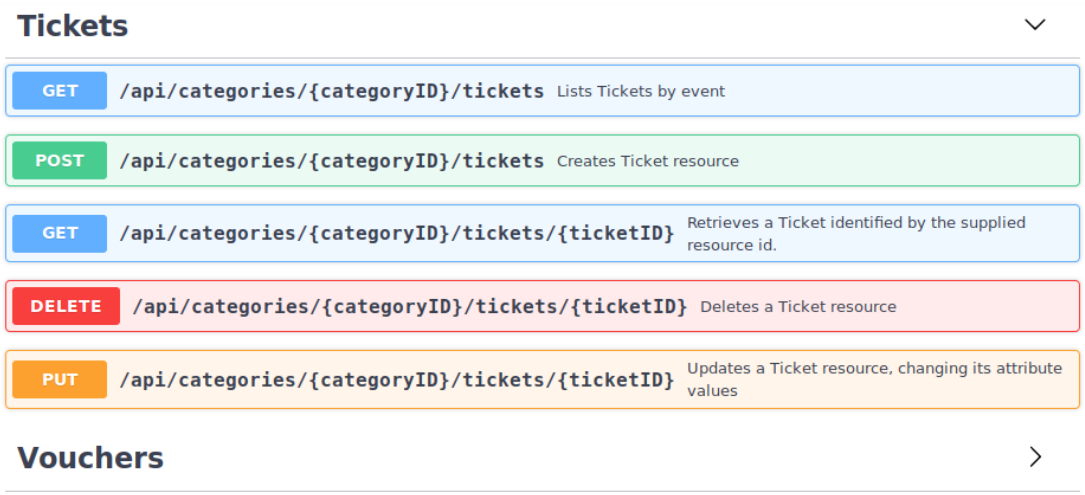

Abbildung 4.4: Ausschnitt aus der Dokumentation der öffentlichen Schnittstellen des Ticket-Services. Die Dokumentation wurde mit Swagger generiert.

Die Service Integration Tests basieren auf den getroffenen Annahmen der Teststufen bei den vorgestellten Testkonzepte von Wolff, Newman, Clemson und Schaffer. Alle Autoren empfehlen in einer ausgewählten Teststufe in ihren jeweiligen Testkonzepten das Isolieren von Abhängigkeiten und das Testen von funktionalen Anforderungen der Schnittstellen ohne dabei ihre Implementation zu betrachten. Vor allem Schaffers konsequenter Ansatz, einen Microservice in Isolation zu seinen Abhängigkeiten als Black-Box zu betrachten und diesen mit einer möglichst hohen Anzahl an Testfällen abzudecken, dient als Inspiration für die Einführung von Service

Integration Tests in diesem Testkonzept.

Durch das Simulieren der Abhängigkeiten wird sichergestellt, dass ein Service Integration Test für einen Microservice nicht aufgrund von Fehlern eines externen Systems fehlschlägt. Außerdem können die Testfälle auf dieser Teststufe schneller ausgeführt werden, weil nicht zusätzlich weitere reale Microservices gestartet werden müssen. Für die Testlaufzeit wird keine zeitliche Beschränkung angegeben, weil die Testlaufzeit abhängig von dem zu testenden Microservice variieren kann. Allgemein besitzen Service Integration Testfälle eine längere Ausführungsdauer als Unit-Tests und auch als Persistence Integration Tests. Allerdings liefern sie ein hohes Maß an Vertrauen, dass die Schnittstellen des Microservices ihrer Spezifikation entsprechen.

Zusätzlich können Consumer Driven Contract Tests verwendet werden, um die Verträge zwischen einem Microservice und seinen Abhängigkeiten zu überprüfen. Sie bieten außerdem die Möglichkeit für Regressionstests, wenn sich die Implementation der Schnittstellen ändert. Empfehlenswert ist auch hier der Einsatz eines Testframeworks für CDCT wie beispielsweise „Pact“. Eine ausführliche, öffentliche Dokumentation der Schnittstellen des zu testenden Microservices erleichtert zudem die Erstellung von Verträgen zwischen Diesem und seinen Abhängigkeiten.

4.2.4 Testkonzept für das Gesamtsystem

Das Testkonzept für das Gesamtsystem des „Eisberg-Modells“ ist unterteilt in die beiden Teststufen „System Integration“ und „End-To-End“. Auf die beiden Teststufen wird in den folgenden Abschnitten eingegangen.

System Integration Tests Bei System Integration Tests wird das gesamte System in einer gemeinsamen Testumgebung gestartet. Ähnlich wie bei den Service Integration Tests auf Ebene eines Microservices, werden die Abhängigkeiten eines Services nicht simuliert, sondern es wird mit realen Abhängigkeiten getestet. Ziel ist das Identifizieren von Fehlern im Zusammenspiel eines geänderten Microservices, bevor der Service in die Produktivumgebung gegeben wird.

Aufgrund der vergleichsweise langen Ausführungszeit von Testfällen, die aus dem Starten des Gesamtsystems resultiert, sollen nicht alle Testfälle aus den Service Integration Tests wiederholt abgebildet werden. Vielmehr sollen kritische Anforderungen unter realen Bedingungen explizit getestet werden. Es wird getestet, ob das Gesamtsystem sich mit dem verändertem

Service wie erwartet starten und ausführen lässt.

Die Teststufe der System Integration Tests orientiert sich an Wolffs Integrationstests auf Ebene des Gesamtsystems (siehe Kap. 3 , Abschn. 3.2.1.1). Innerhalb der Deployment-Pipeline muss eine gemeinsame Teststufe für Integrationstests geschaffen werden. Dabei darf nur ein geänderter Service zur Zeit in der gemeinsamen Testumgebung gestartet und mit den aktuellen Versionen anderer getestet werden. Wenn für mehrere veränderte Versionen unterschiedlicher Services gleichzeitig System Integration Tests durchgeführt werden, kann das Analysieren von fehlerhaften Verhalten und die Fehlerlokalisierung komplex sein. Durch eine lange Testausführung von System Integration Tests auf Ebene des Gesamtsystems und die Beschränkung für veränderte Versionen von Services in der gemeinsamen Testumgebung, kann das schnelle Veröffentlichen von neuen Versionen in die Produktivumgebung verlangsamt werden. Aus diesem Grund wird empfohlen, die Testfälle von System Integration Tests auf wenige, kritische Anforderungen an die zu überprüfenden Microservices zu beschränken.

System Integration Tests liegen in der Verantwortung mehrerer Entwicklerteams und müssen in gegenseitiger Abstimmung implementiert werden. Durch unabhängige Testteams innerhalb der Organisation können Anforderungen an den Testprozess aufgestellt und Regelungen festgelegt werden. Unabhängige Tester können Annahmen überprüfen, die von den Entwicklern der jeweiligen Microservices während der Spezifikation und Implementierung des Systems gemacht worden sind. Sie können möglicherweise auch neue Fehlermöglichkeiten entdecken und testen, die ein Entwickler eines Services durch seine Voreingenommenheit nicht wahrnehmen konnte. Da es für das Versuchssystem wichtig ist, möglichst viele Fehler vor dem Deployment und der Veröffentlichung zu finden, wurden im „Eisberg-Modell“ System Integration Test als Teststufe eingeführt.

End-To-End Tests End-To-End Tests überprüfen den Kommunikationsfluss über mehrere Microservices hinweg. Das Gesamtsystem wird dabei als eine Black-Box betrachtet. Anhand eines konkreten Anwendungsszenarios wird das gesamte System von „einem Ende bis zum Anderen “ durchlaufen. Ziel der Durchführung von End-to-End-Tests ist es, Abhängigkeiten zu erkennen und sicherzustellen, dass die richtigen Daten zwischen verschiedenen Microservices und externen Systemen ausgetauscht werden. Dabei werden weite Teile des Gesamtsystems integriert und ein großer Teil der Codebasis verwendet, weshalb ein bestandener End-To-End Test ein hohes Maß an Vertrauen in das Gesamtsystem liefert.

Ähnlich zu Newmans Testkonzept muss das Gesamtsystem für die Durchführung von End-To-End Tests in einer Testumgebung gestartet werden (siehe Kap. 3, Abschn. 3.2.2.1). Dies wird durch eine gemeinsame Endstufe für End-To-End Tests innerhalb der Deployment-Pipeline ermöglicht. Voraussetzung für das Ausführen von End-To-End Tests ist das erfolgreiche Bestehen der zuvor durchlaufenden System Integration Tests.

Der Einstiegspunkt für End-To-End Tests kann dabei je nach System variieren. Das Versuchssystem besitzt keine grafische Benutzeroberfläche, weshalb End-To-End Tests an den öffentlichen Schnittstellen des API-Gateways angesetzt werden. Als Orienterung für das Entwerfen von geeigneten Testfällen dienen Sequenzdiagramme (siehe Abb. 4.2), die den Datenfluss über mehrere Komponenten des Microservice-basierten Systems darstellen. Besitzt das Gesamtsystem eine grafische Benutzeroberfläche, dann können mithilfe von GUI-Testwerkzeugen wie z.B. mit „Selenium“ automatisierte Testfälle erstellt werden, die Systemfunktionen über diese Oberfläche überprüfen. Wie bei den End-To-End Tests des Testkonzeptes nach Clemson (siehe Kap. 3, Abschn. 3.2.3) kann es sinnvoll sein, externe Systeme durch Mocks zu ersetzen. Dies kann zur Stabilität eines End-To-End Tests beitragen. Weil der Versuchssystem auf keine externen Systeme angewiesen ist, entfällt dieser Aspekt.

Es besteht das Risiko, dass End-To-End Tests aufgrund ihrer hohen Integration leicht fehlschlagen können. Aus diesem Grund sollten wenige Testfälle für grundlegende Anforderungen an das Gesamtsystem implementiert werden. Die Verantwortung für Entwicklung von geeigneten Testfällen liegt ähnlich wie bei den „System Integration Tests“ nicht nur bei einem Entwicklerteam. Auch hier bietet sich ein unabhängiges Testteam an, welches nicht nur an der Entwicklung eines Microservices beteiligt ist. Gegebenenfalls können auch aus der Gruppe der Endbenutzer des System unabhängige Tester abgestellt werden um zu testende Anforderungen an das System zu priorisieren, ergänzt um Testspezialisten, die das Testobjekt gegen Vorschriften wie zum Beispiel Usability oder Sicherheitsstandards überprüfen.

4.2.5 Ergänzungen

In diesem Abschnitt werden ergänzende Möglichkeiten für das Testkonzept des „Eisberg-Modells“ vorgestellt, um weitere Mängel in einzelnen Microservices und dem Gesamtsystem zu entdecken und zu behandeln.

4.2.5.1 Statische Analyse

Als statische Tests können werkzeug-gestützte, statische Analysen das Testkonzept für einen Microservice ergänzen. Durch statische Analysen können Fehler nachgewiesen werden, in denen ein zu testendes Dokument von seiner vorgegebenen, formalen Struktur abweicht. Für das Versuchssystem wurden folgende statische Analysen verwendet, die vor dem Durchlaufen der Teststufen des Testkonzeptes für einen Microservice automatisch ausgeführt werden.

- *Statische Analyse durch den Compiler* — Vor dem eigentlichen Kompilieren des Quellcodes eines Microservices wird durch den Compiler selbst eine Reihe an statischen Analysen durchgeführt, welche zum Beispiel Syntaxfehler, Datenflussanomalien oder Kontrollflussanomalien feststellen können.
- *Verwendung externer Bibliotheken* — Werden in einem Microservice externe Bibliotheken verwendet, bietet es sich an die verwendeten Versionen zu fixieren. Durch Werkzeuge kann in Form einer statischen Analyse überprüft werden, ob die verwendeten Bibliotheken im Programmcode eines geänderten Microservices in den vereinbarten Versionen vorliegen.
- *Einhaltung des Coding-Styles* — Durch statische Analysen können Abweichungen vom vereinbarten Coding-Style festgestellt werden. Beispiele hierfür sind falsch benannte Funktionen oder Methoden, Datenstrukturen oder auch fehlende Kommentare. Abweichungen vom Coding-Style wirken sich nicht auf die Funktionalität des Microservices aus. Dennoch sollte es das Ziel eines jeden Entwicklers im Team sein, qualitativ hochwertigen Code auszuliefern, welcher den vereinbarten Standards entspricht.
- *Verwendung von Lizenzen* — Wird fremder Code im Projekt verwendet, kann es sinnvoll sein die Nutzungslizenzen zu überprüfen. Durch eine statische Analyse kann festgestellt werden, ob die verwendeten Abhängigkeiten jeweils eine Lizenz besitzen. Sie kann auch auf rechtliche Probleme im Zusammenhang mit der Nutzung von bestimmten Lizenzarten im Projekt hindeuten.

4.2.5.2 Exploratives Testen

Exploratives Testen kann das Testkonzept für das Gesamtsystem als weitere Teststufe ergänzen. Weil exploratives Testen sich nicht automatisieren lässt, sollte diese Teststufe je nach Entscheidung für den Einsatz von Continuous Delivery oder Continuous Deployment verwendet

werden (siehe Abb. 4.8).

Durch das Durchlaufen der Teststufen des Microservices und des Gesamtsystems lassen sich bereits viele Fehlerzustände identifizieren. Allerdings kann es manchmal schwierig sein, bestimmte Testfälle zu automatisieren. Auch kann es Qualitätsprobleme geben, die durch automatisierte Testfälle nicht sichtbar werden. Exploratives Testen kann deshalb als zusätzliche Ergänzung zu den bestehenden Teststufen genutzt werden, um Qualitätsprobleme zu entdecken, welche die Deployment-Pipeline nicht erkannt hat.

Ähnlich zu der Teststufe des „Exploratory Testing“ in dem Testkonzept nach Clemson (siehe Kap. 3, Abschn. 3.2.3) können die gewonnenen Erfahrungen beim Explorativen Testen zur Weiterentwicklung der End-To-End Tests verwendet werden. Der Tester sollte sich dabei auf Testaspekte beziehen, die noch nicht durch Testfälle abgedeckt worden sind, wie zum Beispiel langsame Reaktionszeiten oder irreführende Fehlermeldungen.

4.2.5.3 Testen von nicht-funktionalen Anforderungen

Das Hauptaugenmerk dieses Testkonzeptes liegt auf dem Testen von Funktionalen Anforderungen an einen Microservice, sowie an das Gesamtsystem. Allerdings sollte auch das Testen von nicht-funktionalen Anforderungen im Testkonzept berücksichtigt werden.

Das Testkonzept stimmt Wolff und Newman zu und empfiehlt das Nutzen von Lasttests für das Testen von Performance als nicht-funktionale Anforderung. Ähnlich zu Wolff kann zwischen Performanz- und Kapazitätstests unterschieden werden. Performanztestes haben das Ziel, die Ausführungsdauer von Systemfunktionen zu überprüfen, während Kapazitätstests die parallele Bearbeitung von Systemfunktionen testen soll.

Lasttests können auf Ebene des Gesamtsystems des Testkonzeptes angesetzt werden. Sie sollten allerdings aufgrund ihrer langen Ausführungsdauer nicht bei jedem Durchlaufen der Deployment-Pipeline ausgeführt werden, sondern können innerhalb der Entwicklungsumgebung in regelmäßig Abständen verwendet werden, um Mängel vor der Veröffentlichung in die Produktivumgebung zu entdecken.

Das Testen von Benutzbarkeit ist in einem Entwicklungssystem schwierig. Eine Möglichkeit für das Testen dieser nicht-funktionalen Anforderung ist das zuvor vorgestellte „Explorative

Testen“ als zusätzliche manuelle Teststufe auf Ebene des Gesamtsystems (siehe 4.2.5.2).

Auf Ebene des Gesamtsystems können außerdem Audits als unabhängige Prüfung der Einhaltung von Sicherheitsstandards angesetzt werden. Das Testen von Sicherheitsaspekten auf Ebene eines Microservices ist schwierig, weil bei den durchgeführten Tests Abhängigkeiten wie zum Beispiel eine Middleware für das Autorisieren eines Schnittstellen-Aufrufes durch einen Mock simuliert oder per „Feature-Toggle“ deaktiviert werden. Auch werden in der Regel benötigte TLS-Zertifikate oder Anmeldedaten an Datenbanken durch Testdaten ersetzt. Das Testen von Sicherheitsaspekten sollte daher in einer Laufzeitumgebung durchgeführt werden, die möglichst der Produktivumgebung ähnelt.

4.3 Deployment und Veröffentlichung

In diesem Abschnitt wird das Deployment und die Veröffentlichung eines geänderten Microservices im Versuchssystem beschrieben. Dazu wird eine Empfehlung für die Umsetzung einer Deployment-Pipeline für das Gesamtsystem gegeben.

Die Deployment-Pipeline dieses Testkonzeptes basiert auf den Vorschlägen Wolffs und Newmans. Für die einzelnen Microservices verwenden beide Autoren jeweils eine eigene Deployment-Pipeline, in denen der Microservice ohne Starten des Gesamtsystems überprüft wird. Wolff definiert im Anschluss an die einzelnen Pipelines der Services eine gemeinsame Endstufe für Integrationstests aller Microservices (siehe Kap. 3, Abschn. 3.2.1.2). Newman verfolgt einen ähnlichen Ansatz, denn er definiert eine gemeinsame Endstufe für alle Services, in denen End-To-End Tests ausgeführt werden (siehe Kap. 3, Abb. 3.6).

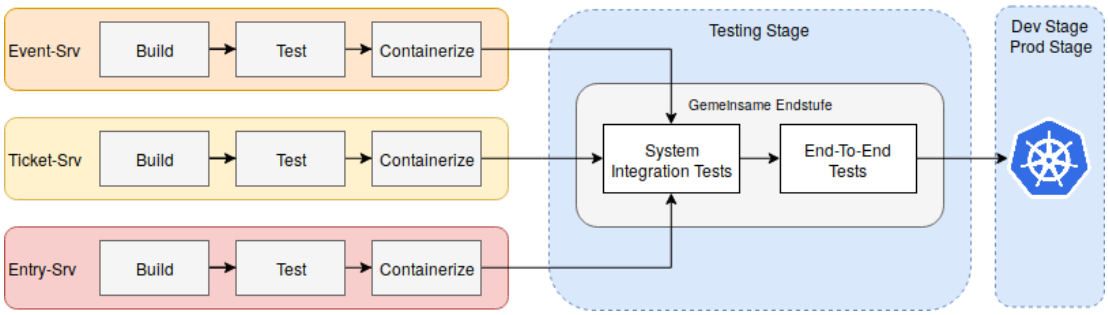

Abbildung 4.5: Die Deployment-Pipeline für das Versuchssystem

Dieses Testkonzept bedient sich beiden Ansätzen und definiert zwei Endstufen für die Deployment-Pipeline für alle Microservices, in denen sowohl gemeinsame Integrationstests ähnlich zu Wolff in Form von „System Integration Tests “, als auch End-To-End Tests nach Newman durchgeführt werden. Auf die Nachteile, die sich aufgrund dieser Entscheidung ergeben, wurde bereits in der zusammenfassenden Bewertung zum Abschluss des dritten Kapitels eingegangen (siehe Kap. 3 , Abschn. 3.4).

4.3.1 Anwendung von Continuous Delivery / Deployment

Die Entscheidung für Continuous Delivery oder Continuous Deployment hat Auswirkungen auf die Gestaltung der Deployment-Pipeline des Gesamtsystems.

Im Versuchssystem wird Continuous Delivery für den Softwareauslieferungsprozess angewendet. Aufgrund dieser Entscheidung wird ein erfolgreich getesteter Microservice nicht sofort in die Produktivumgebung deployed. Als Werkzeug für Testautomatisierung wird Gitlab CI verwendet, das ein automatisches Deployment aus der Pipeline heraus in ein Kubernetes Cluster unterstützt.

Abbildung 4.6: Übersicht der Repositories mit Sourcecode für die Testausführung. Zu jedem Repository existiert ein zugehöriger Namespace in Kubernetes

Insgesamt wurden drei verschiedene Laufzeitumgebungen als Namespaces innerhalb des Kubernetes Clusters definiert:

1. *Namespace-Testing* — Testing ist ein Namespace des Kubernetes Clusters, in dem System Integration und End-To-End Tests durchgeführt werden können. In Kubernetes lassen sich Deployments als YAML-Dateien definieren, wodurch die Testumgebung in dem Kubernetes-Namespace reproduzierbar aufgebaut werden kann. Das zugehörige Repository beinhaltet diese Deployment-Dateien.

2. *Namespace-Staging* — Staging ist ein Namespace des Kubernetes Clusters mit zugehörigem Repository, der den letzten stabilen Entwicklungsstand des Gesamtsystems

bereitstellt. Das zugehörige Repository beinhaltet die Deployment-Dateien zum Bereitstellen und Aktualisieren des Entwicklungssystems.

3. *Namespace-Productive* — Productive ist ein Namespace des Kubernetes Clusters, in dem das Produktivsystem ausgeführt wird. Das zugehörige Repository kann Deployment-Dateien für das Bereitstellen und Aktualisieren des Produktivsystems beinhalten.

Die Codebasis der einzelnen Microservices ist jeweils in einem eigenen Repository organisiert. Für jedes Repository existiert eine eigene Deployment-Pipeline, die sich in der Ausführung der Schritte nicht voneinander unterscheiden. Auf den Ablauf der Pipeline für einen Microservice wird in dem folgenden Abschnitt eingegangen (siehe Abschn. 4.3.2). Wird eine Änderung an der Codebasis eines Microservices in das entsprechende Repository eingecheckt, dann wird automatisch die Deployment-Pipeline aktiviert. Der kürzlich veränderte Code wird von Gitlab CI ausgecheckt und die einzelnen Schritte innerhalb von Docker-Containern in der Deployment-Pipeline werden durchlaufen.

Ist die Deployment-Pipeline des veränderten Microservices erfolgreich bestanden worden, wird über einen Webhook die Deployment-Pipeline des Testing-Namespaces ausgelöst, woraufhin wiederum einzelne Schritte innerhalb dieser Pipeline durchgeführt werden. Ist die Deployment-Pipeline des Testing-Namespaces erfolgreich durchlaufen worden, wird erneut per Webhook die Pipeline des Staging-Namespaces aktiviert, die im laufenden Entwicklungssystem des zugehörigen Kubernetes-Namespaces den geänderten Microservice aktualisiert.

Anschließend kann Exploratives Testen am Entwicklungssystem durchgeführt werden, als zusätzliche Teststufe für die Qualitätssicherung. Erfüllt das System fehlerfrei seine Anforderungen, kann zu einem geeigneten Zeitpunkt die Produktivumgebung unter Verwendung der genannten Deployment-Strategien (siehe Abschn. 4.3.4) aktualisiert werden.

Soll Continuous Deployment für den Softwareauslieferungsprozess genutzt werden, kann auf den Staging-Namespace verzichtet werden. Eine Umstellung der Deployment-Pipeline ist ohne großen Aufwand möglich. Lediglich der Webhook zum Ende des erfolgreichen Bestehens der Deployment-Pipeline des Testing-Namespaces muss auf die Deployment-Pipeline des Productive-Namespaces umgestellt werden. Das Repository des Productive-Namespaces kann Deployment-Dateien für Kubernetes enthalten, die automatisch den veränderten Microservice unter Beachtung der ausgewählten Deployment-Strategie aktualisiert.

4.3.2 Deployment-Pipeline für einen Microservice

Die Deployment-Pipeline für einen Microservice automatisiert das Durchlaufen der Teststufen gemäß des Testkonzeptes.

In einem ersten Schritt “Build„ wird der Quellcode des Microservices mithilfe von werkzeuggestützten, statischen Analysen überprüft. Dabei werden die in 4.2.5.1 genannten Aspekte untersucht. Gleichzeitig wird überprüft, ob sich der Quellcode mithilfe des Compilers zu einer ausführbaren Datei kompiliert lässt. Das gebaute Artefakt wird in einem späteren Schritt in der Deployment-Pipeline verwendet. Sind die statischen Analysen und das Kompilieren des Quellcodes erfolgreich verlaufen, werden in einem nächsten Schritt “Test„ die Teststufen des Testkonzeptes für einen Microservice durchlaufen.

In der ersten Teststufe werden die Unit-Tests ausgeführt, welche die kleinsten Einheiten innerhalb eines Microservices in Isolation überprüfen.
Nach erfolgreichem Bestehen der Unit-Tests folgen die Persistence Integration Tests. Hierfür wird in einem Schritt eine Testdatenbank innerhalb eines Docker-Containers gestartet und der Microservice auf die Verwendung der Testdatenbank konfiguriert. Die entsprechenden Methoden werden durch ein Testframework aufgerufen, wodurch Ressourcen in der Datenbank persistiert werden. Im Anschluss wird kontrolliert, ob das Persistieren der Ressourcen erfolgreich war oder nicht.

Ist auch dieser Schritt erfolgreich verlaufen, werden die Service Integration Tests ausgeführt. Dazu werden die Abhängigkeiten des Microservices simuliert. Der Service selbst wird gestartet und die Schnittstellen des Microservices im Black-Box Verfahren durch RPCs aufgerufen. Anschließend wird kontrolliert, ob die Schnittstelle wie erwartet funktioniert. Durch CDCTs wird zusätzlich das Einhalten der Verträge überprüft.

Ist kein Service Integration Test fehlgeschlagen, wird das zuvor gebaute Artefakt in einem letzten Schritt “Containerize„ in einen Docker-Container verpackt und mit einem Hash gekennzeichnet. Der verpackte Microservice wird in eine Container-Registry hochgeladen und später im Testkonzept für das Gesamtsystem verwendet. Zum Schluss der Deployment-Pipeline wird über einen Webhook die Deployment-Pipeline des Gesamtsystems aktiviert.

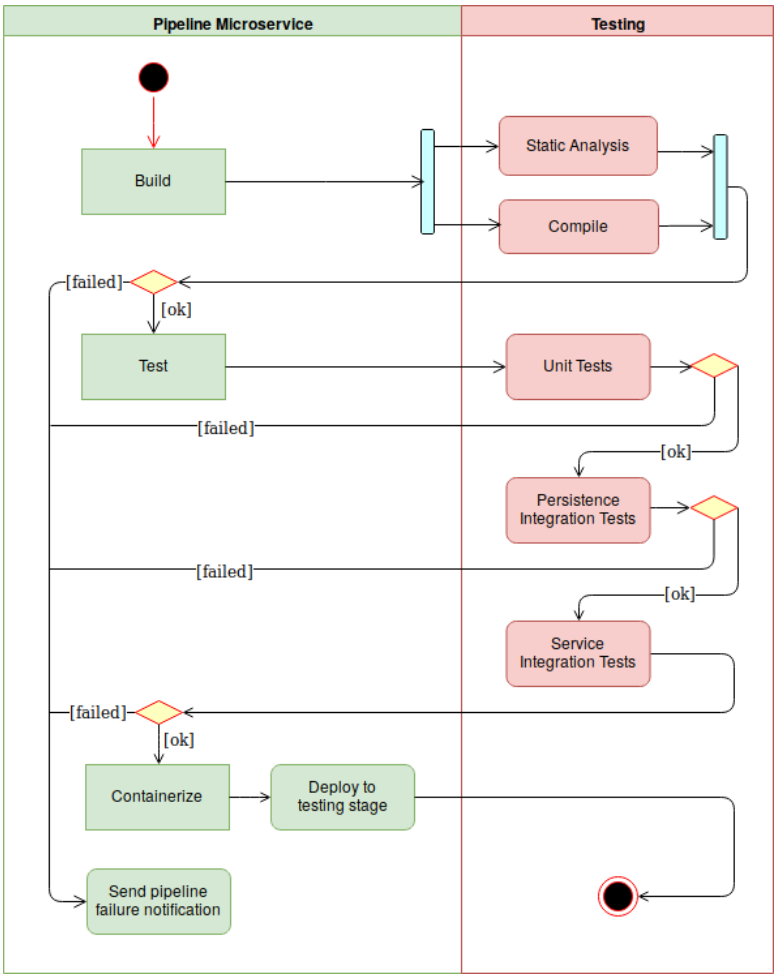

Abbildung 4.7: Aktivitätsdiagramm für den Ablauf einer Deployment-Pipeline für einen Microservice

4.3.3 Deployment-Pipeline für das Gesamtsystem

Der Startpunkt der Deployment-Pipeline des Gesamtsystems ist der Webhook der zuvor erfolgreich durchlaufenden Deployment-Pipeline eines Microservices. In einem ersten Schritt "Test„ werden die einzelnen Teststufen des Testkonzeptes für das Gesamtsystem durchlaufen.

Die Deployment-Pipeline des Gesamtsystem checkt die im Repository des Testing-Namespaces enthaltenen Dateien aus. Das Repository enthält alle notwendigen Deployment-Dateien und den Code für die Durchführung von System Integration und End-To-End Tests.

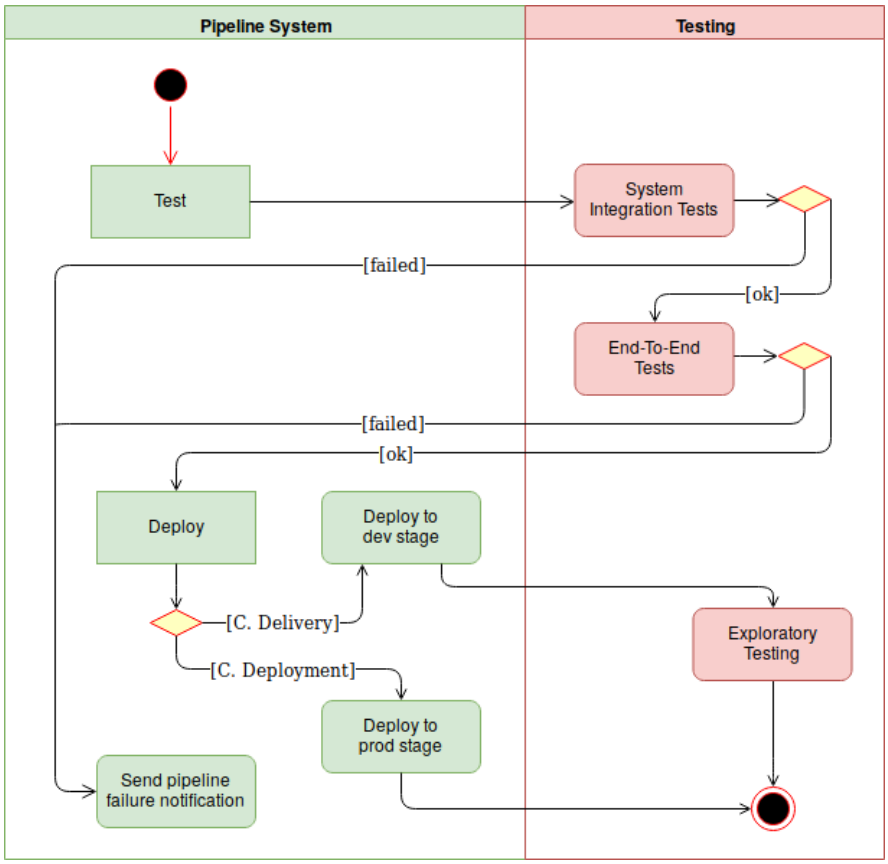

Abbildung 4.8: Aktivitätsdiagramm für den Ablauf einer Deployment-Pipeline für das Gesamtsystem

Für die Durchführung von System Integration Tests wird das gesamte System in den Kubernetes-Namespace deployed. Anschließend werden ähnlich wie bei den Service Integration Tests die Schnittstellen des Microservices direkt mit RPCs aufgerufen. Die Abhängigkeiten des Services werden nicht simuliert, stattdessen wird mit realen Microservices in deren aktueller Version der Produktivumgebung getestet und kontrolliert, ob die Schnittstellen sich wie erwartet verhalten.

Nach erfolgreichem Bestehen der System Integration Tests, werden End-To-End Tests durchgeführt. Das gesamte System wird dabei als Black-Box betrachtet, die öffentlichen Systemschnittstellen des System aufgerufen und dadurch mehrere Microservices anhand von konkreten Anwendungsszenarios durchlaufen.

In einem zweiten Schritt “Deploy„ wird per Webhook die Deployment-Pipeline des Staging-Namespaces, beziehungsweise des Productive-Namespaces aktiviert und der geänderte Service in der jeweiligen Laufzeitumgebung aktualisiert. Gleichzeitig wird das Gesamtsystem im Testing-Namespace heruntergefahren und die Ressourcen werden freigegeben.

4.3.4 Deployment-Strategien

Wie bei Newmans Testkonzept kann der Testprozess durch die Anwendung von Deployment-Strategien ergänzt werden (siehe Kap. 3, Abschn. 3.2.2.2). Dadurch können Fehler aufgefangen werden, die erst im laufenden Betrieb auftreten. Als Deployment-Strategie kann Blue / Green Deployment oder „Canary Releasing“ angewendet werden.

Das Versuchssystem nutzt Kubernetes als Tool für Container-Orchestrierung und verwendet für einen neuen Release eines Microservices das „Canary Releasing“. In Kubernetes wird dies über Labels und Selektoren umgesetzt. Durch Labels können Eigenschaften eines Deployments gekennzeichnet werden. Diese Eigenschaften können für den Benutzer relevant sein, werden allerdings nicht für die Kernfunktionalitäten von Kubernetes selbst benötigt. Mithilfe von Labels können die von dem API-Server in Kubernetes zur Verfügung gestellten Abstraktionen organisiert und in Teilmengen zerlegt werden.

```
1  # Service definition for event-service
2  apiVersion: v1
3  kind: Service
4  metadata:
5      name: eventsrv
6      ...
7  spec:
8      selector:
9          app: eventsrv
10         tier: backend
11         track: stable
12 ---
13 # deployment definition for the event-service
14 apiVersion: extensions/v1beta1
15 kind: Deployment
16 metadata:
17     name: eventsrv
18     ...
19     template:
20         metadata:
```

```
21              labels:
22                  app: eventsrv
23                  tier: backend
24                  track: stable
25          spec:
26              containers:
27                  - name: eventsrv
28                  image: docker-hub.informatik.haw-hamburg.de/ticketee/event-srv:latest
29  ...
```

Listing 4.3: Ausschnitt aus dem Kubernetes Deployment des Event-Services des Versuchssystems

Wie in Abbildung 4.4 dargestellt, wird über das Label „Track“ mit dem Wert „Stable“ das letzte stabile Deployment eines Microservices im Gesamtsystem gekennzeichnet. Soll ein aktualisierter Microservice in die Produktivumgebung deployed werden, wird der Wert des Labels auf den Wert „Canary“ geändert. Der Selektor der Service-Abstraktion, welche den Zugriff auf die definierte, logische Menge an Pods im Deployment steuert, wird aktualisiert, in dem das Label „Track“ für die Selektion von Pods entfernt wird. Dadurch wird eingehender Netzwerkverkehr sowohl an den Canary-Release, als auch an den letzten stabilen Release weitergeleitet.

```
1  # Service definition update canary-release
2  apiVersion: v1
3  kind: Service
4  metadata:
5      name: eventsrv
6      ...
7  spec:
8      selector:
9          app: eventsrv
10         tier: backend
11         # Omitting track label
12 ---
13 # deployment definition for event-service canary release
14 apiVersion: extensions/v1beta1
15 kind: Deployment
16 metadata:
17     name: eventsrv-canary
18     ...
19     template:
```

```
20          metadata:
21              labels:
22                  app: eventsrv
23                  tier: backend
24                  track: canary
25          spec:
26              containers:
27              - name: eventsrv
28              image: docker-hub.informatik.haw-hamburg.de/ticketee/event-srv:test
29  ...
```

Listing 4.4: Ausschnitt aus einem Canary-Deployment für einen aktualisierten Event-Service

Werden Fehler in der Produktivumgebung durch die Veröffentlichung des neuen Microservices sichtbar, kann schnell reagiert werden. Durch das Aktualisieren der Service-Ressource kann der Netzwerkverkehr wieder auf die letzte stabile Version umgeleitet und das Canary-Deployment sicher entfernt werden.

4.4 Evaluation

In diesem Abschnitt soll das vorgestellte Testkonzept im Bezug auf die in genannten Herausforderungen für das Testen in einer Microservice-Architektur (siehe Kap. 3, Abschn. 3.1.2) und den gestellten Anforderungen im Umgang mit Änderungen evaluiert werden (siehe Abschn. 4.2.1).

4.4.1 Umgang mit den Herausforderungen von Microservices

Die Entscheidung für eine Microservice-Architektur beeinflusst die Art und Weise, wie die Software getestet werden muss in einem großen Maße. Nachfolgend wird erläutert, ob das vorgestellte Testkonzepten diesen Herausforderungen gerecht werden kann:

- *Dokumentation* — Um eine Microservice-Architektur umfassend testen zu können, ist eine Dokumentation der Schnittstellen und das Wissen um gegenseitige Abhängigkeiten zwischen den Microservices notwendig. Das Testkonzept selbst hat nicht die Aufgabe eine angemessene Dokumentationsgrundlage zur Verfügung zu stellen, vielmehr ist die Dokumentation Voraussetzung für das sinnvolle Testen der Microservices und des Gesamtsystems. Trotzdem kann das Testkonzept unterstützen, in dem es auf undokumentierte Systembestandteile und auf Abweichung von vereinbarten Konventionen hinweist. Auch kann durch das Durchlaufen des Testprozesses identifiziert werden,

wo eine Dokumentation erstellt werden muss und ob das Erstellen von bestimmten Dokumentationsteilen höher priorisiert werden sollte als das von Anderen. In dem vorgestellten Testkonzept wird auf den Einsatz von statischen Analysen verwiesen, um formale Abweichungen der Testdokumente zu entdecken. Zusätzlich kann beim Einsatz von Continuous Delivery das Explorative Testen als Teststufe genutzt werden, wenn wenig Vorwissen über das System vorhanden ist. Allerdings ersetzt es in keinem Fall das Anfertigen einer Dokumentation. Für die Dokumentation der Versionen benötigter Abhängigkeiten eines Microservices konnte in diesem Testkonzept keine geeignete Lösung gefunden werden.

- *Kommunikation und Interaktion* — Als ein verteiltes System rufen Microservices sich gegenseitig über standardisierte Schnittstellen auf Netzwerkebene auf. Gegenüber monolithischen Anwendung verlagert sich deshalb ein großer Teil der Komplexität aus der Anwendung hinaus in die Kommunikation der Services untereinander. Das Testkonzept ist sich dessen bewusst und fokussiert sich auf die Schnittstellen als wichtige Testobjekte. Auf Ebene eines Microservices werden durch eine große Anzahl an Testfällen die Schnittstellen eines Microservices in Isolation zu dessen Abhängigkeiten überprüft. Durch ergänzende Consumer-Driven-Contract Tests wird das Einhalten der Verträge zwischen zwei Microservices getestet. Weil selbst kleine Änderungen in dynamischen Systemen wie Microservices große Auswirkungen haben können und deshalb das isolierte Betrachten eines Services nicht ausreichend ist, werden System Integration Tests auf Ebene des Gesamtsystems als zusätzliche Teststufe in diesem Konzept vorgestellt, um Fehlerwirkungen vor der Veröffentlichung zu identifizieren. Durch sie werden die Interaktionen zwischen Microservices mit echtem Netzverkehr getestet und damit geprüft, ob der aktualisierte Microservice mit Netzwerkfehlern wie Latenzen oder Timeouts umgehen kann.

- *Unabhängige Entwicklung* — Microservices als kleine, isolierte und selbständige Einheiten innerhalb des Gesamtsystems müssen sich unabhängig voneinander entwickeln lassen. Die Unterteilung des Testkonzeptes in zwei unterschiedliche Ebenen unterstützt dies. Durch Unit-Tests, Persistence Integration und Service Integration Tests kann ein einzelner Service unabhängig von dem Gesamtsystem überprüft werden. Das Implementieren der Teststufen und das Bereitstellen von Mocks bzw. Stubs für einen Microservice liegt in der Verantwortung des jeweiligen Entwicklerteams. Zusätzlich besitzt jeder Microservice eine eigene, unabhängige Deployment-Pipeline, wodurch die einzelnen Teststufen automatisiert durchlaufen werden. Die Deployment-Pipeline kann dahin-

gehend konfiguriert werden, dass nur bei Änderungen auf dem Master-Branch oder explizitem Kennzeichnen einer neuen Version die Deployment-Pipeline für das Gesamtsystem aktiviert wird. Dadurch wird eine unabhängige Entwicklung eines Microservices und schnelles Feedback an die Entwickler bei Änderungen möglich.

- *Unabhängige Veröffentlichung* — Microservices als Komponenten des Gesamtsystems müssen sich unabhängig voneinander veröffentlichen lassen. Das Testkonzept schlägt eine gemeinsame Deployment-Pipeline für alle Microservices und die Berücksichtigung von Deployment-Strategien für die Veröffentlichung vor. Eine unabhängige Veröffentlichung der einzelnen Microservices ist möglich, allerdings hat die Entscheidung für eine gemeinsame Deployment-Pipeline Auswirkungen auf die Geschwindigkeit in der sich neue Versionen unterschiedlicher Microservices in der Entwicklungs- bzw. Produktivumgebung veröffentlichen lassen. Durch die gemeinsame Deployment-Pipeline wird die Ressourcennutzung des Testprozesses beschränkt, allerdings hat dies negative Auswirkungen auf die Geschwindigkeit für Neuveröffentlichungen.

4.4.2 Umgang mit Änderungen in einer Microservice-Architektur

In einer Microservice-Architektur ist die schnelle und unabhängige Entwicklung und Veröffentlichung einzelner Microservices wichtig. Deshalb muss das Testkonzept einen geeigneten Umgang mit Änderungen vorschlagen. In diesem Abschnitt sollen die an das Testkonzept gestellten Anforderungen aus 4.2.1 evaluiert werden. Dazu wird zunächst für die unterschiedlichen Arte von Änderungen beschrieben, welche Tätigkeiten dafür bei Nutzung des „Eisberg-Modells“ notwendig sind.

Integrieren eines neuen Microservices Wird ein neuer Microservice in das Gesamtsystem integriert, müssen verschiedene Teststufen angepasst werden. Es wird davon ausgegangen, dass der neue Microservices bereits mit Unit-Tests zum Testen der kleinsten Einheiten innerhalb des Services, Persistence Integration Tests für das Überprüfen der korrekten Persistierung von Ressourcen und Service Integration Tests zum Testen der Schnittstellen mit simulierten Abhängigkeiten abgedeckt ist. Zunächst muss das Team des neuen Microservices einen Mock bzw. Stub des Services für die anderen Entwicklerteams bereitstellen. Bei den Verbrauchern des neuen Microservices müssen die Consumer Driven Contract Tests angepasst werden, sodass sie die vom neuen Microservice geforderten Verträge erfüllen. Auf Ebene des Gesamtsystems können System Integration Tests ergänzt werden, die sich an den Testfällen der Service Integration Tests des neuen Services mit simulierten Abhängigkeiten orientieren. Bietet der

zu integrierende Microservice eine neue Systemfunktionalität an, kann über das Erstellen eines End-To-End Tests entschieden werden. Allerdings werden End-To-End Tests in dem vorgestellten Testkonzept weniger stark gewichtet.

Entfernen eines bestehenden Microservices Das Entfernen eines Microservices aus dem Gesamtsystems hat Auswirkung auf die Teststufen des Gesamtsystems. Durch den Wegfall eines Services werden die Consumer Driven Contract Tests der Verbraucher des zu entfernenden Service fehlschlagen, weshalb die Testfälle angepasst werden müssen. Gleichzeitig müssen andere Entwicklerteams informiert werden, dass die Service Integration Tests, welche die Mocks bzw. Stubs des zu entfernenden Services verwenden, gelöscht werden können. Auf Ebene der System Integration Tests müssen die Integrationstests des wegfallenden Microservices entfernt werden. Existieren End-To-End Tests, die Kommunikationspfade entlang des zu entfernenden Microservice enthalten, müssen diese ebenfalls entfernt werden.

Ersetzen / Anpassen eines bestehenden Microservices Wenn ein Microservice einen Bestehenden ersetzt, muss zwischen zwei verschiedenen Szenarios unterschieden werden. Erfüllt der zu ersetzende Microservice die Spezifikation der Schnittstelle des zu ersetzenden Microservices, müssen die Teststufen des Gesamtsystems nicht angepasst werden. Erfüllt der zu ersetzende Microservice die Spezifikation der Schnittstelle des zu ersetzenden Microservice nicht, müssen die anderen Entwicklerteams über das Ersetzen in Kenntnis gesetzt werden, sodass sie die Änderung der Abhängigkeit ihres eigenen Microservices dokumentieren und sie die Mocks bzw. Stubs des ersetzenden Microservices in ihren Service Integration Tests nutzen können. Außerdem müssen die Consumer Driven Contract Tests der Verbraucher auf Ebene eines Microservices und die System Integration Tests sowie End-To-End Tests auf Ebene des Gesamtsystems angepasst werden.

Wird ein bestehender Microservice angepasst, ist entscheidend, auf welcher Ebene die Anpassung vorgenommen wird. Ändert sich die Spezifikation der Schnittstelle des Microservices, wird ähnlich wie bei dem beschriebenen Szenario des Ersetzens eines bestehenden Microservices vorgegangen. Handelt es sich um eine interne Anpassung des Microservices, durch die sich die Schnittstellen des Services nicht verändern, müssen nur die Unit-Tests bzw. Persistence Integration Tests angepasst werden. Ansonsten lassen sich folgende Fälle unterscheiden:

- *Anforderungsänderung* — Abhängig von der Art der Anforderungsänderung müssen die Teststufen angepasst werden. Handelt es sich um eine Änderung einer nicht-funktionalen

Anforderung an einen bestehenden Microservice, müssen Unit-Tests, Persistence Integration Tests oder Service Integration Tests angepasst werden. Beispielsweise können sich die Anforderungen an die Perfomanz eines Services ändern, weshalb eine bestimmte Methode eine neue Laufzeitbeschränkung erhält. Hierfür kann ein Unit-Test angepasst werden, der die Laufzeit gemäß der Anforderung kontrolliert. Handelt es sich um die Änderung einer Funktionalen Anforderung, sind in der Regel auch die Schnittstellen des Microservices betroffen. Für die Anpassung von Teststufen ist auch hier wieder entscheidend, auf welchen Ebenen des Microservices oder des Gesamtsystems die Änderungen umgesetzt worden sind, um die Anforderung gemäß der Spezifikation zu erfüllen und die Testfälle der Teststufen anzupassen.

- *Erweiterung um neue Features* — Wird ein bestehender Microservice um neue Features erweitert, ist für die Anpassung von Testfällen auch hier wieder relevant, auf welcher Ebene das Feature hinzugefügt wird. Wird innerhalb eines Microservices eine neue Funktion oder Methode, die eine gewisse Komplexität aufweist, hinzugefügt oder verändert, dann muss ein neuer Unit-Test implementiert oder angepasst werden. Ändern sich die Eigenschaften einer vom Microservice verwalteten Ressource, müssen die Persistence Integration Tests angepasst oder erweitert werden. Aufwendiger wird es, wenn sich die Schnittstelle eines Microservices verändert. In diesem Fall müssen die Service Integration Tests, vor allem aber die Consumer Driven Contract Tests, angepasst werden. Zusätzlich müssen die Mocks bzw. Stubs des Microservices gemäß der geänderten Version angepasst werden und die Verbraucher des Microservices müssen ihrerseits die Service Integration Tests ändern. Darüber hinaus müssen in diesem Fall auch die System Integration Tests und ggf. End-To-End Tests abgeändert werden. Das Anpassen der Schnittstelle eines Microservices sollte daher gründlich überlegt und in Absprache mit den betroffenen Entwicklerteams geplant werden.

- *Behebung von gefundenen Fehlerzuständen* — Abhängig davon, wie ein Fehlerzustand lokalisiert wurde, muss mit der Situation unterschiedlich umgegangen werden. Wird ein Fehlerzustand durch einen fehlgeschlagenen Test innerhalb der Deployment-Pipeline eines Services entdeckt, muss dieser auf Ebene eines Microservices behoben und die Pipeline erneut ausgeführt werden. Besteht der Microservice alle Tests, gilt der Fehlerzustand als behoben. Gleiches gilt für das Fehlschlagen eines Testfalls in der gemeinsamen Deployment-Pipeline, mit dem Unterschied, dass womöglich mehrere Microservices angepasst werden müssen. Zusätzlich kann der Fall eintreten, dass ein Fehlerzustand in Produktion oder durch den Einsatz von Exploratory Testing entdeckt wird. Hier muss

ein geeigneter Testfall auf der entsprechenden Ebene ergänzt werden, was durch die hohe Integration und die breite verwendete Codebasis mit großen Aufwand verbunden sein kann. Allerdings kann nur durch das Hinzufügen eines Testfalles und anschließende Fehlerbehebung geprüft werden, ob der Fehlerzustand im Testprozess abgedeckt ist.

Wie gezeigt lassen sich alle Arten der Änderungen im „Eisberg-Modell“ mit einem klar definierten Änderungsprozess abarbeiten, der sich auf die absolut notwendigen Anpassungen der Tests beschränkt. Insgesamt zeigt sich, dass Änderungen, die Auswirkungen auf Abhängigkeiten zwischen Microservices haben, wesentlich aufwändiger einzupflegen sind als lokale Änderungen. Dies gilt für das „Eisberg-Modell“ ebenso wie für alle in Kapitel 3 vorgestellten Testkonzepte.

4.4.3 Bewertung

Für das Versuchssystem wurde ein Testkonzept entwickelt, das den Anforderungen eines Microservice-basierten System gerecht wird. Das Testkonzept muss mit den Herausforderungen eines verteilten Systems umgehen können, darf den Entwicklungsprozess nicht verlangsamen und muss eine geeignete Vorgehensweise für Änderungen implementieren.

Hierbei wirkte sich die Gliederung des Testkonzeptes in zwei unterschiedliche Teile positiv auf die unabhängige Entwicklung und Veröffentlichung einzelner Microservices als Teil eines Gesamtsystems aus.
Durch eine eigene Deployment-Pipeline für die individuellen Microservices wurde der Entwicklungsprozess einzelner Services unterstützt, in dem sich Microservices unabhängig vom Gesamtsystem automatisiert testen lassen. Durch die gemeinsamen Pipeline-Schritte für System Integration und End-To-End Tests wurde das Gesamtsystem vor dem Deployment und der Veröffentlichung eines neuen Microservices überprüft. Dadurch können Fehler, welche sich aus der Kommunikation und Interaktion der Microservices mit dem neuen Release ergeben, vor dem Deployment oder der Veröffentlichung in eine Entwicklungs- bzw. Produktivumgebung entdeckt und behoben werden. In der praktischen Umsetzung hat sich die Deployment-Pipeline bewährt. Die nicht-parallelisierbaren, gemeinsamen Pipeline-Schritte können allerdings zu einem Flaschenhals in der Softwarebereitstellung führen.

Kommunikation und Interaktion mit externen Abhängigkeiten ist eine der größten Herausforderung für einen Microservice. Dadurch, dass das Testkonzept einen Schwerpunkt auf das Durchführen von Integrationstests legt, um die Schnittstellen eines Microservices gemäß ihrer

Spezifikation zu überprüfen, wird es dieser Herausforderung gerecht. Es ist in der Lage, mit den gestellten Anforderungen bezüglich des Umgangs mit Änderungen umzugehen. Ändert sich die Implementation eines Microservices, kann das jeweilige Entwicklerteam zu einem großen Teil selbst die entsprechenden Testfälle für Unit-, Persistence Integration und Service Integration Tests auf den Teststufen anpassen. Ändert sich die Schnittstelle eines Microservices, hat dies zusätzlich Auswirkungen auf System Integration- und End-To-End Tests, weshalb das Testkonzept dazu rät, besonders fehleranfällige Schnittstellen zwischen den Microservices mit geeigneten Testfällen in diesen Teststufen abzudecken.

Die prototypische Umsetzung von Teilen des Konzeptes anhand des Versuchssystems hat gezeigt, dass sich das Testkonzept auf vergleichbare Microservice-Basierte Systeme anwenden lässt. Das entwickelte Konzept wurde den selbst gestellten Anforderungen bezüglich des Umgangs mit Änderungen, der Unterstützung des Entwicklungsprozesses und den Herausforderungen an das Testen von Microservices gerecht.

5 Fazit

In diesem Kapitel werden die Inhalte der Bachelorarbeit zusammengefasst. In dem Abschnitt „Ausblick“ werden Aspekte genannt, die bei der Entwicklung und Implementierung des Testkonzeptes offen geblieben sind und mit denen das Ergebnis dieser Arbeit fortgeführt werden kann.

5.1 Zusammenfassung

Das Ziel dieser Arbeit war das Untersuchen, Zusammenfassen und Bewerten aktueller Testkonzepte für Microservice-Architektur (3) und darauf basierend die Entwicklung eines individuellen Testkonzeptes für ein Versuchssystem (4). Zunächst wurde auf das Modell der klassischen Testpyramide nach Cohn eingegangen (3.1) und erläutert, weshalb deren Anwendung auf Microservice-basierte Systeme eine Herausforderung darstellt (3.1.2). Daraufhin wurden aktuelle Testkonzepte für Microservices der Autoren Eberhard Wolff, Sam Newman, Toby Clemson und André Schaffer vorgestellt und zusammengefasst (3.2).

In einem ausführlichen Vergleich (3.3) wurden die Testkonzepte anhand der Einteilung von Teststufen gegenübergestellt und anschließend bewertet (3.4).

Auf Grundlage des Vergleiches und der Bewertung wurde für ein Versuchssystem das „Eisberg-Modell“ als Darstellung für ein individuelles Testkonzept entwickelt (4.2.2). Das zweiteilige Testkonzept für einen Microservice (4.2.3) und das Gesamtsystem (4.2.4) wurde dargestellt und es wurde auf die Unterteilung und Gewichtung der einzelnen Teststufen eingegangen. Weiterhin wurden Vorschläge für die Ergänzung des Testkonzeptes gemacht (4.2.5). Für das Testkonzept wurde eine Deployment-Pipeline vorgestellt, welche die einzelnen Teststufen durchläuft und einen aktualisierten Microservice unter der Berücksichtigung von Deployment-Strategien veröffentlichen kann (4.3).

Abschließend wurde evaluiert, inwiefern das entwickelte Testkonzept den Herausforderungen im Zusammenhang mit dem Testen von Microservices und den an das Testkonzept gestellten

Anforderungen für den Umgang mit Änderungen gerecht wird (4.4).

Das Ergebnis dieser Arbeit ist ein State-Of-The-Art Vergleich aktueller Testkonzepte für Microservice-Architekturen anhand der Einteilung und Gewichtung von Komponenten-, Integrations- und Systemtests sowie ein darauf basierendes Testkonzept. Das entwickelte Konzept orientiert sich an den Charakteristiken der vorgestellten Testkonzepte und lässt sich auf Microservice-basierte Systeme anwenden.

5.2 Ausblick

Das in dieser Arbeit entwickelte, individuelle Testkonzept auf Grundlage der untersuchten Testkonzepte nach Wolff, Newman, Clemson und Schaffer, wurde in Teilen an einem Versuchssystem implementiert. Das Versuchssystem selbst bestand aus wenigen Microservices, die zugleich wenig Abhängigkeiten besaßen. Es bleibt offen, inwiefern sich das entwickelte Testkonzept auf ein größeres, auf Microservices basierendes System anwenden lässt und welche Schwierigkeiten sich im Zusammenhang mit dessen Anwendung ergeben.

Weiterhin wäre interessant zu untersuchen, inwiefern auf die gemeinsamen Pipeline-Schritte für Integrations- und Systemtests des Gesamtsystems verzichtet werden könnte, um den Bereitstellungsprozess von neuen Microservice-Releases nicht zu behindern, allerdings trotzdem eine Beschränkung von Ressourcen für den Testprozess vorgenommen werden kann. Ein alternativer, im Rahmen dieser Arbeit nicht verfolgter Ansatz ist auf eine gemeinsame Deployment-Pipeline für alle Microservices zu verzichten und System Integration, sowie End-To-End Tests innerhalb der Pipeline einen einzelnen Microservice auszuführen. Diese Entscheidung führt dazu, dass möglicherweise mehrere Instanzen des Gesamtsystems gestartet werden und für einzelne Services die Teststufen des Testkonzeptes des Gesamtsystems parallel durchlaufen werden, was wiederum deutlich mehr Ressourcen wie z.B. Rechenleistung beansprucht. Allerdings können dafür Microservices schneller veröffentlicht werden. Durch den Einsatz einer Computing Cloud im Testprozess, die Ressourcen nach Bedarf zur Verfügung stellt und bei Nicht-Nutzung freigibt, wäre dieser Ansatz denkbar um das schnelle, unabhängige Veröffentlichen von Microservices zu unterstützen.

Literaturverzeichnis

[Andreas Spillner 2012] Andreas Spillner, Tilo L.: *Basiswissen Softwaretest*. Dpunkt.Verlag GmbH, 2012. – URL https://www.ebook.de/de/product/19361935/andreas_spillner_tilo_linz_basiswissen_softwaretest.html. – ISBN 978-3-86490-024-2

[André Schaffer 2018] André Schaffer, Rickard D.: *Testing of Microservices*. 2018. – URL https://labs.spotify.com/2018/01/11/testing-of-microservices/. – Zugriffsdatum: 2018-06-13

[Clemson 2014] Clemson, Toby: *Testing Strategies in a Microservice Architecture*. 2014. – URL https://martinfowler.com/articles/microservice-testing/. – Zugriffsdatum: 2018-06-22

[Cohn 2009a] Cohn, Mike: *The Forgotten Layer of the Test Automation Pyramid*. 2009. – URL https://www.mountaingoatsoftware.com/blog/the-forgotten-layer-of-the-test-automation-pyramid. – Zugriffsdatum: 2018-06-11

[Cohn 2009b] Cohn, Mike: *Succeeding with Agile*. Addison Wesley, 2009. – ISBN 978-0321579362

[Farcic 2016] Farcic, Viktor: *The DevOps 2.0 Toolkit: Automating the Continuous Deployment Pipeline with Containerized Microservices*. 1. Auflage. CreateSpace Independent Publishing Platform, 2016. – ISBN 978-1523917440

[Fielding 2000] Fielding, Roy T.: *Architectural Styles and the Design of Network-based Software Architectures*, Dissertation, 2000. – URL https://www.ics.uci.edu/~fielding/pubs/dissertation/fielding_dissertation.pdf

[Fowler 2011] Fowler, Martin: *Contract Test*. 2011. – URL https://martinfowler.com/bliki/ContractTest.html. – Zugriffsdatum: 2018-06-05

[Fowler 2012] Fowler, Martin: *TestPyramid*. 2012. – URL https://martinfowler.com/bliki/TestPyramid.html. – Zugriffsdatum: 2018-06-11

[Fowler 2014] Fowler, Martin: *Microservices - Who Has Used Them?* 2014. – URL https://www.martinfowler.com/microservices/. – Zugriffsdatum: 2018-05-30

[Froslie 2016] Froslie, Daid: *Test Automation Pyramid*. 2016. – URL https://blogs.msdn.microsoft.com/dave_froslie/2016/03/09/test-automation-pyramid/. – Zugriffsdatum: 2018-06-13

[George T. Heinemann 2006] George T. Heinemann, Ivica Crnkovic Heinz W. S.: *Component-Based Software Engineering*. Springer Berlin Heidelberg, 2006. – URL https://www.ebook.de/de/product/25406079/component_based_software_engineering.html. – ISBN 978-3-540-35629-5

[Gitlab 2018a] Gitlab: *Gitlab*. 2018. – URL https://about.gitlab.com/. – Zugriffsdatum: 2018-06-03

[Gitlab 2018b] Gitlab: *Gitlab CI*. 2018. – URL https://docs.gitlab.com/ee/ci/. – Zugriffsdatum: 2018-06-03

[Hightower u. a. 2017] Hightower, Kelsey ; Burns, Brandon ; Beda, Joe: *Kubernetes: Up and Running*. O'Reilly UK Ltd., 2017. – URL https://www.ebook.de/de/product/24777509/kelsey_hightower_brandon_burns_joe_beda_kubernetes_up_and_running.html. – ISBN 9781491935675

[Hoffman 2016] Hoffman, Kevin ; Anderson, Brian (Hrsg.): *Beyond the Twelve-Factor App*. 1. Edition. O'Reilly Media, Inc., 2016. – ISBN 9781492042631

[Hohmann 2003] Hohmann, Luke: *Beyond Software Architecture: Creating and Sustaining Winning Solutions*. ADDISON WESLEY PUB CO INC, 2003. – URL https://www.ebook.de/de/product/3254976/luke_hohmann_beyond_software_architecture_creating_and_sustaining_winning_solutions.html. – ISBN 0-201-77594-8

[Homès 2013] Homès, Bernard: *Fundamentals of Software Testing*. 1. Auflage. ISTE LTD, jan 2013. – URL https://www.ebook.de/de/product/18587029/bernard_homes_fundamentals_of_software_testing.html. – ISBN 9781848213241

[Humble 2010] Humble, David Farley J.: *Continuous Delivery*. Pearson Technology Group, 2010. – URL https://www.ebook.de/de/product/9446498/jez_humble_david_farley_continuous_delivery.html. – ISBN 9780321601919

[IEEE 2000] IEEE: IEEE Recommended Practice for Architectural Description of Software-Intensive Systems. In: *IEEE Std 1471-2000* (2000), S. i–23

[Kerr 2018] Kerr, Dave: *The Death of Microservice Madness in 2018*. 2018. – URL https://www.dwmkerr.com/the-death-of-microservice-madness-in-2018/. – Zugriffsdatum: 2018-06-02

[Kolesnikov 2018] Kolesnikov, Dmitry: *Using Microservices to Power Fashion Search and Discovery*. 2018. – URL https://jobs.zalando.com/tech/blog/using-microservices-to-power-fashion-search-and-discovery/. – Zugriffsdatum: 2018-05-30

[Kruchten 1995] Kruchten, P. B.: The 4+1 View Model of architecture. In: *IEEE Software* 12 (1995), Nov, Nr. 6, S. 2. – ISSN 0740-7459

[Kubernetes.io 2018] Kubernetes.io: *Kubernetes Concepts*. 2018. – URL https://kubernetes.io/docs/concepts/. – Zugriffsdatum: 2018-06-04

[Martin 2005] Martin, Robert C.: *The Principles of OOD*. 2005. – URL http://www.butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod. – Zugriffsdatum: 2018-06-05

[Martin Fowler 2014] Martin Fowler, James L.: *Microservices - a definition of this new architectural term*. may 2014. – URL https://www.martinfowler.com/articles/microservices.html. – Zugriffsdatum: 2018-05-30

[Michael Wittig 2015] Michael Wittig, Andreas W.: *Amazon Web Services in Action*. Manning, 2015. – URL https://www.ebook.de/de/product/24270477/michael_wittig_andreas_wittig_amazon_web_services_in_action.html. – ISBN 978-1-61729-288-0

[Newman 2016] Newman, Sam: *Building Microservices*. O'Reilly UK Ltd., 2016. – URL https://www.ebook.de/de/product/22539693/sam_newmann_building_microservices.html. – ISBN 978-1491950357

[Nickoloff 2016] Nickoloff, Jeff: *Docker in Action*. Manning, 2016. – URL https://www.ebook.de/de/product/24270233/jeff_nickoloff_docker_in_action.html. – ISBN 9781633430235

[Nova 2018] Nova, Kris: *What is a monolithic application?* 2018. – URL https://blog.heptio.com/what-is-a-monolithic-application-e375f5ad5ecb?gi=a9a1f3e2ca08. – Zugriffsdatum: 2018-06-16

[Pact.io 2018] Pact.io: *Contract Testing*. 2018. – URL https://docs.pact.io/. – Zugriffsdatum: 2018-06-05

[Rainsberger 2010] Rainsberger, J. B.: *Integrated Tests Are A Scam*. 2010. – URL http://blog.thecodewhisperer.com/permalink/integrated-tests-are-a-scam. – Zugriffsdatum: 2018-06-13

[Raymond 2003] Raymond, Eric S.: *The Art of UNIX Programming*. Pearson Education (US), 2003. – URL https://www.ebook.de/de/product/3259137/eric_s_raymond_the_art_of_unix_programming.html. – ISBN 978-0131429017

[Robinson 2006] Robinson, Ian: *Consumer-Driven: A Service Evolution Pattern*. 2006. – URL https://www.martinfowler.com/articles/consumerDrivenContracts.html. – Zugriffsdatum: 2018-06-23

[Savchenko u. a. 2015] Savchenko, D. I. ; Radchenko, G. I. ; Taipale, O.: Microservices validation: Mjolnirr platform case study. In: *2015 38th International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*, May 2015, S. 235–240

[Scott 2013] Scott, Allister: *Testing Pyramids & Ice-Cream Cones*. 2013. – URL https://watirmelon.blog/testing-pyramids/. – Zugriffsdatum: 2018-06-13

[Sneed u. a. 2011] Sneed, Harry M. ; Baumgartner, Manfred ; Seidl, Richard: *Der Systemtest*. Hanser Fachbuchverlag, 2011. – URL https://www.ebook.de/de/product/14914017/harry_m_sneed_manfred_baumgartner_richard_seidl_der_systemtest.html. – ISBN 978-3446426924

[Society 2014] Society, IEEE C.: *Guide to the Software Engineering Body of Knowledge (SWEBOK(R)): Version 3.0*. IEEE Computer Society Press, 2014. – ISBN 978-0-7695-5166-1

[Thones 2015] Thones, J.: Microservices. In: *IEEE Software* 32 (2015), Jan.-Feb., Nr. 1, S. 116–117. – URL doi.ieeecomputersociety.org/10.1109/MS.2015.11. – ISSN 0740-7459

[Whittaker 2009] Whittaker, James A.: *Exploratory Software Testing: Tips, Tricks, Tours, and Techniques to Guide Test Design.* ADDISON WESLEY PUB CO INC, 2009. – ISBN 978-0321636416

[Winter u. a. 2016] Winter, Mario ; Rossner, Thomas ; Brandes, Christian ; Götz, Helmut: *Basiswissen modellbasierter Test.* Dpunkt.Verlag GmbH, 2016. – URL https://www.ebook.de/de/product/25914576/mario_winter_thomas_rossner_christian_brandes_helmut_goetz_basiswissen_modellbasierter_test.html. – ISBN 978-3-86490-297-0

[Wolff 2015] Wolff, Eberhard: *Microservices: Grundlagen flexibler Softwarearchitekturen.* 1. Auflage. dpunkt.verlag GmbH, 2015. – ISBN 978-3864903137

Abkürzungsverzeichnis

API Application Programming Interface

AMQP Advanced Message Queuing Protocol

CDCT Consumer Driven Contract Testing

GTB German Testing Board

HAW Hochschule für Angewandte Wissenschaften Hamburg

HAWAI HAW Laboratory for Architecture and IT Management

HTTP Hypertext Transfer Protocol

IP Internet Protocol

ISTQB International Software Testing Qualifications Board

JSON JavaScript Object Notation

REST Representational State Transfer

RPC Remote Procedure Call

SOAP Simple Object Access Protocol

TLS Transport Layer Security

XML Extensible Markup Language

YAML YAML Ain't Markup Language

Stichwortverzeichnis

Hiermit versichere ich, dass ich die vorliegende Arbeit ohne fremde Hilfe selbständig verfasst und nur die angegebenen Hilfsmittel benutzt habe.

Hamburg, 3. August 2018 Christian Bargmann