# Testing Microservice Integration with Consumer-Driven Contract Tests

MASTERARBEIT

**Felix Quast**

Eingereicht am 7. Januar 2022


Friedrich-Alexander-Universität Erlangen-Nürnberg
Technische Fakultät, Department Informatik
Professur für Open-Source-Software

Betreuer:
Georg Schwarz, M.Sc.
Prof. Dr. Dirk Riehle, M.B.A.

# Versicherung

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

---

Erlangen, 7. Januar 2022

# License



---

Erlangen, 7. Januar 2022

# Abstract

Microservice-based systems consist of self-contained, distributed services that communicate via network connections. Testing service integrations of such systems can be challenging because it involves running multiple services at the same time and there are many potential sources for false-negative test results.

Consumer-Driven Contract Testing (CDCT) is an approach that can be used for testing both sides of a service integration independently. This is achieved by decoupling the two services using a contract as an intermediary. It is driven by the consuming service and expresses its expectations of the API that it uses from the providing service.

This thesis investigates, how CDCT can contribute to test the integration of microservices by identifying benefits, drawbacks, challenges and guidelines that are associated with CDCT for microservice-based systems. For the initial theory building, a structured literature review was conducted. After that, action research was conducted where Consumer-Driven Contract tests were developed for an open-source microservice-based system. Finally, after the concluding evaluation of the action research, the findings of the structured literature review were compared to the experiences gained from the action research.

# Zusammenfassung

Microservice-Systeme bestehen aus eigenständigen, verteilten Services, die über Netzwerkverbindungen miteinander kommunizieren. Das Testen von Service-Integrationen kann bei derartigen Systemen eine Herausforderung darstellen, da hierzu mehrere Services zur selben Zeit ausgeführt werden müssen und es viele potenzielle Quellen für falsch-negative Testergebnisse gibt.

Consumer-Driven Contract Testing (CDCT) ist ein Ansatz, der dazu verwendet werden kann, beide Seiten einer Service-Integration unabhängig voneinander zu testen. Dies wird erreicht, indem die beiden Seiten der Integration mithilfe eines Vertrags (engl. *contract*) voneinander entkoppelt werden, wobei der Contract als Vermittler fungiert. Dieser wird durch den Service vorgegeben, welcher die Schnittstelle eines anderen Services beansprucht, und drückt dessen Erwartungen an die verwendete Schnittstelle aus.

Diese Arbeit erforscht, inwieweit CDCT zur Testung von Microservice-Integrationen beitragen kann, indem Vorteile, Nachteile, Herausforderungen und Richtlinien erfasst werden, die im Zusammenhang zu CDCT für Microservice-Systeme stehen. Für die initiale Theoriebildung wurde zunächst eine strukturierte Literaturrecherche durchgeführt. Im Anschluss wurde Aktionsforschung betrieben, bei der Consumer-Driven Contract Tests für ein Open-Source Microservice-System entwickelt wurden. Zuletzt, nach der abschließenden Evaluation der Aktionsforschung, wurden die Inhalte, die im Rahmen der strukturierten Literaturrecherche erhoben wurden, mit den Erfahrungen aus der Aktionsforschung abgeglichen.

# Inhaltsverzeichnis

# Abbildungsverzeichnis

# Tabellenverzeichnis

# Akronyme

AMQP: Advanced Message Queuing Protocol

API: Application Programming Interface

CDCT: Consumer-Driven Contract Testing

CDC: Consumer-Driven Contract

CD: Continuous Delivery

CI: Continuous Integration

DDD: Domain-Driven Design

E2E: End-to-End

HTTP: Hypertext Transfer Protocol

JSON: JavaScript Object Notation

ODS: JValue Open Data Service

REST: Representational State Transfer

UI: User Interface

URL: Uniform Resource Locator

YAML: YAML Ain't Markup Language

# 1 Aufbau der Arbeit

Diese Arbeit ist nachfolgend in zwei Teile gliedert: Zum einen in eine kurzgefasste Ausarbeitung, welche die Aufmachung eines wissenschaftlichen Artikels (engl. *paper*) hat, und zum anderen in weitergehende Erläuterungen, in denen vertiefende und ergänzende Inhalte thematisiert werden, die über den Rahmen des kurzgefassten Teils hinausgehen. An geeigneten Stellen existieren Querverweise zwischen diesen beiden Teilen, um auf ergänzende Inhalte aufmerksam zu machen.

Kapitel 2 umfasst die kurzgefasste Ausarbeitung. Diese beginnt mit Sektion 2.1, die eine Einführung in die Thematik bietet und die Forschungsfragen der Arbeit benennt. In Sektion 2.2 wird eine themenverwandte Arbeit vorgestellt, wonach in Sektion 2.3 die Vorgehensweisen der strukturierten Literaturrecherche und der Aktionsforschung beschrieben werden. Im Anschluss werden in Sektion 2.4 die Resultate der strukturierten Literaturrecherche präsentiert, woraufhin die Ergebnisse der Aktionsforschung in Sektion 2.5 dargelegt werden. In Sektion 2.6 findet ein Abgleich zwischen den erhobenen Inhalten der strukturierten Literaturrecherche und der Aktionsforschung statt und zuletzt werden in Sektion 2.7 die Ergebnisse der Arbeit zusammengefasst und ein Ausblick für zukünftige Forschungsarbeiten gegeben.

Das nachfolgende Kapitel 3 umfasst die weitergehenden Erläuterungen. Zu Beginn werden in Sektion 3.1 Grundlagen zu den Themen Microservice Architekturen und Testverfahren für Microservices vermittelt. Insbesondere wird in dieser Sektion auch Consumer-Driven Contract Testing thematisiert. In Sektion 3.2 werden einige Werkzeuge vorgestellt, die im Rahmen der Aktionsforschung eingesetzt wurden. Danach folgt Sektion 3.3, in welcher ergänzende Inhalte zur Entwicklung der Consumer-Driven Contract Tests thematisiert werden, wonach in der abschließenden Sektion 3.4 auf weitergehende Details des Defect Seedings[1] eingegangen wird.

[1] Hierbei handelt es sich um einen Evaluationsansatz, welcher gegen Ende der Sektion 2.3.2 der kurzgefassten Ausarbeitung genauer erläutert wird.

# 2 Kurzgefasste Ausarbeitung

## 2.1 Einleitung

Microservice-Systeme bestehen aus eigenständigen, lose gekoppelten Services, die in separaten Prozessen und gegebenenfalls auf unterschiedlichen Rechnern ausgeführt werden und über Netzwerkverbindungen miteinander kommunizieren (Wolff, 2017). Die Aufteilung des Systems in Microservices folgt üblicherweise dem Ansatz des Domain-Driven Design (DDD), wonach die Servicegrenzen anhand der Geschäftsdomäne festgelegt werden.
Mit Microservice-Architekturen gehen diverse positive Eigenschaften einher, darunter Skalierbarkeit, Robustheit, unabhängiges Deployment einzelner Services und der Möglichkeit, unterschiedliche Technologien je Service einzusetzen (Newman, 2021).

Eine Herausforderung stellt hingegen das Testen von Microservices über Servicegrenzen hinweg dar. Insbesondere End-to-End (E2E) Tests, bei denen eine große Anzahl von Services involviert sind, benötigen viel Zeit für die Durchführung und liefern potenziell falsch-negative Testergebnisse (Newman, 2021).

Consumer-Driven Contract Testing (CDCT) verfolgt hingegen einen anderen Ansatz, um Service-Integrationen zu testen. Das Grundprinzip ist, dass wenn ein konsumierender Service (sog. Consumer) die Schnittstelle eines anderen Services (sog. Provider) in Anspruch nimmt, ein Vertrag (engl. *contract*) zwischen diesen beiden Services geschlossen wird. Bei den Inhalten des Contracts handelt es sich um die Erwartungen, welche der Consumer an den Provider stellt. Mithilfe des Contracts kann ein Provider schließlich verifizieren, ob er den geforderten Ansprüchen des Consumers genügt (Richardson, 2018).
Ausführlichere Hintergrundinformationen zu Microservice Architekturen und Testverfahren für Microservices sind in Sektion 3.1 der weitergehenden Erläuterungen aufzufinden.

Für das Testen von Microservice-Integrationen wird CDCT in der Literatur oftmals als geeignetere Alternative zu herkömmlichen Testverfahren, wie Integrationstests oder E2E Tests, angeführt. Ob dies zutreffend ist, soll im Rahmen die-

ser Arbeit ergründet werden. Die zentrale Fragestellung lautet daher, inwieweit CDCT zur Testung von Microservice-Integrationen beitragen kann. Insbesondere folgende Teilaspekte werden betrachtet:

- Welche Vorteile bietet CDCT bei Microservice-Architekturen?
- Welche Nachteile bietet CDCT bei Microservice-Architekturen?
- Welche Herausforderungen existieren für CDCT bei Microservice-Architekturen?
- Welche Richtlinien existieren für CDCT bei Microservice-Architekturen?

Dabei sind Nachteile als inhärente negative Eigenschaften von CDCT zu verstehen, wohingegen Herausforderungen Problemstellungen beschreiben, die sich potenziell bewältigen lassen.

In einer strukturierten Literaturrecherche konnten 6 Vorteile, 1 Nachteil, 4 Herausforderungen und 7 Richtlinien aus insgesamt 4 Publikationen ermittelt werden. Mittels Aktionsforschung und einer abschließenden Evaluation wurden diese im Kontext des Open-Source Projekts JValue Open Data Service (ODS) untersucht. Dabei konnten 2 Vorteile, 1 Nachteil und 2 Herausforderungen bestätigt werden sowie zu 2 Richtlinien neue Erfahrungswerte eingeholt werden. Des Weiteren konnten zusätzlich ein Nachteil, eine Herausforderung sowie 2 Richtlinien identifiziert werden.

Der weitere Aufbau der kurzgefassten Ausarbeitung ist Kapitel 1 zu entnehmen.

## 2.2 Themenverwandte Arbeiten

Als einzige themenverwandte Arbeit konnte (Lehvä et al., 2019) identifiziert werden. Dort wurden im Rahmen einer Fallstudie die Tests eines bestehenden Microservice-Systems mit Consumer-Driven Contract (CDC) Tests ergänzt. Die Arbeit untersuchte dabei, wie Microservice-Integrationen durch CDCT effektiver getestet werden können und wie sich die Einführung von CDC Tests auf die Verantwortlichkeiten anderer Testarten auswirkt.
Im Gegensatz zu einer Fallstudie wurde in dieser Arbeit sowohl eine strukturierte Literaturrecherche als auch Aktionsforschung betrieben, um Vorteile, Nachteile, Herausforderungen und Richtlinien von CDCT zu ermitteln. Ein weiterer Unterschied besteht in der Kommunikation der getesteten Microservices: In (Lehvä et al., 2019) kommunizieren die Microservices ausschließlich mittels Hypertext Transfer Protocol (HTTP), während die in dieser Arbeit involvierten Microservices nach außen zwar eine Representational State Transfer (REST) Schnittstelle anbieten, die Services untereinander jedoch ausschließlich asynchron über einen Message Broker mittels Advanced Message Queuing Protocol (AMQP)[1] kommunizieren. Im Rahmen der Evaluation führten Lehvä et al. (2019) ein sog. De-

[1] https://www.amqp.org/

fect Seeding durch. Dabei wurden einzelne Integrationsfehler in das Microservice-System eingebracht, um das Fehleraufdeckungspotenzial unterschiedlicher Testarten miteinander zu vergleichen. Ein solches Defect Seeding wurde in dieser Arbeit ebenfalls durchgeführt, jedoch wurden darüber hinaus auch Entwickler interviewt, die im Rahmen der Aktionsforschung eigenständig CDC Tests entwickelt haben.

## 2.3 Forschungsansatz

Im Rahmen dieser Arbeit wurde sowohl eine strukturierte Literaturrecherche als auch Aktionsforschung betrieben, um Erkenntnisse zu den in Sektion 2.1 genannten Forschungsfragen zu erlangen. Die jeweiligen Vorgehensweisen werden in den folgenden beiden Sektionen 2.3.1 und 2.3.2 beschrieben.

### 2.3.1 Strukturierte Literaturrecherche

Bei der Durchführung der strukturierten Literaturrecherche wurde sich an den Richtlinien von Kitchenham und Charters (2007) orientiert, jedoch wurden sie nicht umfassend befolgt, um den nötigen Aufwand zu begrenzen. Dies ist auch der Grund, weshalb diese Recherche lediglich als strukturiert und nicht als systematisch bezeichnet wird.

#### Suchprozess

Im Rahmen des Suchprozesses wurde ausschließlich die Suchmaschine Google Scholar[2] verwendet. Folgender Suchbegriff kam zum Einsatz:

```
("test" OR "tests" OR "testing")
      AND ("microservice" OR "microservices" OR "micro service" OR
           "micro services" OR "micro-service" OR "micro-services")
```

Dieser wurde nur auf die Titel der Publikationen bezogen, um die Anzahl der Suchergebnisse zu begrenzen und deren Relevanz sicherzustellen.

#### Ein- und Ausschlusskriterien

Folgende Einschlusskriterien wurden verwendet:

- Publikationen, die in englischer Sprache verfasst sind
- Publikationen, auf die durch den Autor dieser Arbeit frei zugegriffen werden kann
- Publikationen, bei denen es um das Testen von Microservices geht

[2] https://scholar.google.de/

Folgendes Ausschlusskriterium kam zum Einsatz:

- Publikationen, in denen Vorteile, Nachteile, Herausforderungen und Richtlinien von CDCT allesamt nicht thematisiert werden

Zur Überprüfung der Kriterien kam ein Beurteilungsprozess aus drei Schritten zum Einsatz:

1. Zusammenfassung der Publikation erfassen
2. Einleitungs-, Diskussions- und Schlussfolgerungssektion der Publikation erfassen
3. Vollständige Publikation erfassen

Für jedes aufgefundene literarische Werk wurde gemäß dieses Beurteilungsprozesses die Eignung überprüft. Dabei wurde iterativ vorgegangen: Sofern nach der Durchführung eines Schrittes die Eignung der Publikation anhand der Ein- und Ausschlusskriterien nicht eindeutig festgestellt werden konnte, wurde der darauf folgende Schritt durchgeführt.

### Qualitätsanforderungen

Als Qualitätsanforderung wurde festgelegt, dass ausschließlich wissenschaftliche Publikationen, die einem Peer-Review unterzogen wurden, in der Recherche berücksichtigt werden.

### Backward und Forward Snowballing

Auf Grundlage der als geeignet eingestuften Literatur wurde jeweils eine Iteration Backward und Forward Snowballing durchgeführt.

Bei Backward Snowballing werden Werke aus den Literaturverzeichnissen der ausgewählten Literatur gesammelt. Beim Forward Snowballing hingegen kommt jene Literatur in Betracht, die Werke aus der ausgewählten Literatur zitiert (Wohlin, 2014).

Die auf diese Weise aufgefundenen literarischen Werke wurden anschließend auf Grundlage des bereits beschriebenen Beurteilungsprozesses anhand der Ein- und Ausschlusskriterien sowie Qualitätsanforderungen als geeignet oder ungeeignet eingestuft.

### Datensammlung und -synthese

Für jedes infrage kommende literarische Werk wurden die Autoren, der Titel und die Quelle in Form einer URL festgehalten. Im Rahmen des Beurteilungsprozesses wurden jeweils Begründungen für die Kategorisierung in geeignet oder ungeeignet verfasst.

Vorteile, Nachteile, Herausforderungen und Richtlinien von CDCT, die in den als geeignet eingestuften literarischen Werken thematisiert wurden, sind durch den Autor dieser Arbeit erhoben worden. Dabei wurden jeweils die Kernaussagen zu derartigen Aspekten gesammelt und inhaltliche Überschneidungen unterschiedlicher Werke identifiziert und als solche gekennzeichnet.

Die Resultate der Datensynthese sind Sektion 2.4.2 zu entnehmen.

### 2.3.2 Aktionsforschung

Mittels Aktionsforschung (engl. *action research*) wurde untersucht, inwieweit CDCT dazu eingesetzt werden kann, Service-Integrationen eines existierenden Microservice-Systems zu testen.

Die Methodik umfasst eine iterative, kollaborative Vorgehensweise, bei der bestehende Problemstellungen analysiert werden und mit geeigneten Maßnahmen vermindert oder behoben werden sollen. Auswirkungen von durchgeführten Maßnahmen werden dabei evaluiert und daraus resultierende Lehren festgehalten (Baskerville, 1999).

#### Vorstellung des JValue Open Data Service

Das Microservice-System, für welches im Rahmen der Aktionsforschung CDC Tests entwickelt wurden, ist der JValue Open Data Service[3]. Dieser kann dazu verwendet werden, Daten aus unterschiedlichen Quellen zu sammeln, weiterzuverarbeiten und Clients zur Verfügung zu stellen.

Abbildung 2.1 zeigt eine vereinfachte Übersicht über die involvierten Microservices und deren Kommunikationsbeziehungen. Eine umfassendere Darstellung ist in Abbildung A1 des Anhangs zu finden.

Die Kommunikation zwischen dem User Interface (UI) und den Microservices ist mittels REST realisiert, wobei die Nutzdaten in JavaScript Object Notation (JSON) übermittelt werden. Derartige Kommunikationspfeile in der Abbildung 2.1 zeigen dabei vom Consumer zum Provider. Als Datenbankmanagementsystem wird stets PostgreSQL[4] eingesetzt. Beim Storage Service kommt außerdem ein PostgREST[5] Service zum Einsatz, um eine REST-Schnittstelle für das UI bereitzustellen.
Die Microservices auf Server-Seite kommunizieren ausschließlich asynchron durch einen AMQP Message Broker untereinander. In der Abbildung 2.1 zeigen derartige Kommunikationspfeile vom Provider zu einem oder mehreren Consumern.

[3] https://github.com/jvalue/ods
[4] https://www.postgresql.org/
[5] https://github.com/PostgREST/postgrest

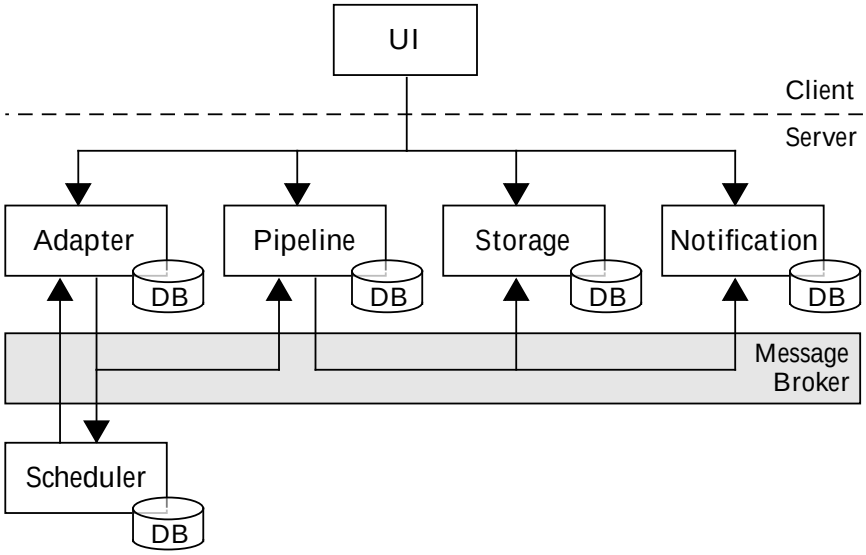


Abbildung 2.1: Vereinfachter Überblick über die Microservices des ODS und deren Kommunikationsbeziehungen. Zylinder mit der Aufschrift „DB“ symbolisieren, dass der zugehörige Microservice über ein Datenbanksystem verfügt.

Nachrichten, die zum Message Broker gelangen, werden ausschließlich durch Outboxer-Services[6] erzeugt und versendet. Dabei überwacht der Outboxer-Service die Datenbank des zugehörigen Microservice und sendet im Zuge von Einbringungen in eine spezifische `OUTBOX`-Tabelle die eingebrachten Inhalte als Nachricht an den Message Broker (Richardson, 2018).

Im ODS kommen folgende Testverfahren zum Einsatz: Unit Tests, die kritische Geschäftslogik isoliert testen; Integrationstests innerhalb einzelner Services, die das Zusammenspiel von Modulen testen; Integrationstests für Microservices, die jedoch keine Microservice-Integrationen, sondern ausschließlich Integrationen von Microservices mit deren Datenbanksystem und ggf. deren Outboxer-Service sowie dem Message Broker testen und zuletzt E2E Tests, die alle Microservices gemeinsam szenariobasiert testen. In Letzteren ist das UI allerdings ausgeschlossen.

Eine Besonderheit stellt die Organisation der Versionskontrolle dar: Der Quellcode aller Microservices des ODS wird in einem einzigen Git-Repository verwaltet. Dies impliziert, dass ein unabhängiges Deployment einzelner Microservices nicht unterstützt wird. Des Weiteren existiert keine separate Versionierung der Services. Die Continuous Integration (CI) Pipeline des ODS wurde mittels GitHub Actions[7] umgesetzt. Diese Technologie wird in Sektion 3.2.3 der weitergehenden

[6] https://github.com/jvalue/outboxer-postgres2rabbitmq
[7] https://docs.github.com/en/actions

Erläuterungen vorgestellt.

### Eignung des JValue Open Data Service

Die Eignung des ODS für die Aktionsforschung wird mithilfe der Anforderungen von Aderaldo et al. (2017) festgestellt. Diese sollten gemäß den Autoren durch ein Microservice-System erfüllen werden, sofern es als Referenzsystem für Forschungszwecke dienen soll.

| | | Anforderung | Begründung |
|---|---|---|---|
| Architektur | 1. | Explizite topologische Übersicht | Eine entsprechende Abbildung befindet sich auf der Projektseite des ODS. |
| | 2. | Pattern-basierte Architektur | Diverse Patterns kommen zum Einsatz, darunter Messaging, Database per Service, Transactional Outbox und API Gateway. |
| DevOps | 3. | Einfacher Zugri über Versionskontrollsystem | Es handelt sich um Open-Source Software, der Zugri kann über GitHub erfolgen. |
| | 4. | Unterstützung von Continuous Integration | Wird durch die Verwendung von GitHub Actions erfüllt. |
| | 5. | Unterstützung von automatisiertem Testen | Wird durch die Verwendung von Jest und JUnit erfüllt. |
| | 6. | Unterstützung von Abhängigkeitsmanagement | Wird durch die Verwendung von npm und Gradle erfüllt. |
| | 7. | Unterstützung von wiederverwendbaren Container Images | Alle Microservices des ODS werden als Docker-Images verö entlicht. |
| | 9. | Unterstützung von Container Orchestrierung | Es werden sämtliche Dateien bereitgestellt, die für das Deployment des ODS in einen Kubernetes Cluster benötigt werden. |

Tabelle 2.1: Auflistung derjenigen Anforderungen an Microservice-Systeme von Aderaldo et al. (2017), die durch das ODS-System erfüllt werden.

Die Tabelle 2.1 zeigt auf, welche der insgesamt 12 Anforderungen durch das ODS-System erfüllt werden.

Die Anforderungen 8. sowie 10. bis 12. werden durch das ODS-System nicht erfüllt, jedoch kann dies im Rahmen dieser Arbeit toleriert werden. Die Tabelle 2.2 listet diese Anforderungen daher mit entsprechenden Begründungen für die Tolerierung auf.

### Entwicklung der Consumer-Driven Contract Tests

Ziel der Aktionsforschung war die Realisierung von CDC Tests für Microservice-Integrationen des ODS. Dabei sollten sowohl HTTP- als auch AMQP-basierte

| | Anforderung | Begründung |
|---|---|---|
| DevOps | 8. Unterstützung von automatisiertem Deployment | CDC Tests können zwar die Entscheidung beeinflussen, ob ein Deployment stattfinden kann, jedoch wird diese Entscheidung auf Grundlage des Ausgangs der CI Pipeline getro□en. In diese können die CDC Tests trotz mangelndem automatisierten Deployment eingegliedert werden. |
| Generell | 10. Unabhängigkeit der Automatisierungstechnologie | Ein Vergleich zwischen unterschiedlichen Automatisierungstechnologien wird nicht gezogen. Die Existenz einer CI Pipeline genügt für die Realisierung der CDC Tests. |
| | 11. Alternative Versionen | Für die Realisierung der CDC Tests besteht kein Bedarf an alternativen Implementierungen desselben Microservices, da kein Vergleich zwischen alternativen Implementierungen gezogen wird. |
| | 12. Gemeinschaftliche Verwendung und Interesse | Das globale Interesse der Microservice-Forschungsgemeinschaft am ODS ist für die Realisierung der CDC Tests unerheblich. |

Tabelle 2.2: Auflistung derjenigen Anforderungen an Microservice-Systeme von Aderaldo et al. (2017), die durch das ODS-System nicht erfüllt werden. Die Begründungen geben jeweils an, weshalb dies toleriert werden kann.

Integrationen umgesetzt werden. Als Werkzeug kam hierbei Pact zum Einsatz, welches in Sektion 3.2.1 der weitergehenden Erläuterungen vorgestellt wird.

Die Entwicklung wurde zunächst durch den Autor dieser Arbeit und später auch durch Entwickler des ODS durchgeführt. Der dabei angewandte iterative Prozess folgt den in Aktionsforschung etablierten Zyklen[8].
Eine Iteration verläuft hierbei wie folgt:
Zu Beginn werden noch ungetestete Service-Integrationen oder Probleme des aktuellen Implementierungsansatzes in Erfahrung gebracht. Im Anschluss wird eine Integration, die innerhalb dieses Zyklus getestet werden soll, ausgewählt oder zwischen Lösungsansätzen für das bestehende Problem abgewägt. Danach erfolgt die Umsetzung der CDC Tests oder der ausgewählten Problemlösung. Im Rahmen eines Pull Requests werden die Änderungen vor deren Einbringung schließlich durch Entwickler des ODS begutachtet.

### Evaluationsansätze

Für die Durchführung einer abschließenden Evaluation wurden zwei unterschiedliche Ansätze verfolgt.

Zum Einen wurden im Rahmen des letzten Zyklus der Aktionsforschung weitere CDC Tests von 3 Entwicklern des ODS-Projekts umgesetzt. Diese umfassten die

[8] Einzelheiten hierzu sind der Arbeit von Baskerville (1999) zu entnehmen.

HTTP-Integration zwischen dem UI und dem Notification Service. Dazu wurde die Application Programming Interface (API) des Notification Service anhand ihrer Schnittstellenoperationen unter den Entwicklern aufgeteilt. Zur Einarbeitung in CDCT wurden den Entwicklern praxisorientierte Materialien zur Verfügung gestellt. Nach Fertigstellung der CDC Tests wurden semistrukturierte Einzelinterviews mit den Entwicklern geführt, um deren gesammelte Erfahrungen einzuholen. Der Leitfaden, an welchem sich bei der Durchführung der Interviews orientiert wurde, befindet sich in Sektion C des Anhangs.

Als zweiter Ansatz wurde ein sog. Defect Seeding[9] durchgeführt. Dabei wurden diverse Integrationsfehler jeweils einzeln an einer willkürlichen, geeigneten Stelle in das ODS-System eingebracht. Beispiele für derartige Fehler sind Hinzufügungen, Umbenennungen, Entfernungen und Wertebereichsänderungen von Attributen und Parametern versendeter Nachrichten sowie, bezüglich HTTP, auch Änderungen an Status Codes, versendeten Headern und Schnittstellenoperationen. Im Anschluss wurde ermittelt, welche Integrationsfehler durch welche Testverfahren aufgedeckt werden konnten.
Weitere Details zu der Herleitung der einzubringenden Integrationsfehler und zur Durchführung des Defect Seedings sind Sektion 3.4 der weitergehenden Erläuterungen zu entnehmen.

## 2.4 Theorie

Im Rahmen der initialen Recherche, die auf Grundlage des in Sektion 2.3.1 angeführten Suchbegriffs durchgeführt wurde, konnten insgesamt 46 Publikationen gefunden werden. Nach Überprüfung der Ein- und Ausschlusskriterien sowie der Qualitätsanforderungen, erwiesen sich 4 der literarischen Werke als geeignet.

Auf Grundlage dieser 4 ausgewählten Publikationen wurde anschließend Backward und Forward Snowballing durchgeführt. Dabei ergab das Backward Snowballing 48 weitere Werke und das Forward Snowballing ein weiteres Werk. Durch die anschließende Prüfung der Kriterien wurden jedoch alle diese Werke als ungeeignet eingestuft.

### 2.4.1 Ausgewählte Literatur

Diese Sektion führt die 4 Werke an, die im Rahmen der Literaturrecherche ausgewählt wurden, stellt diese inhaltlich kurz vor und benennt Gründe für deren Auswahl.

In dem Conference Paper (Koschel et al., 2021) wird über die Analyse, den Entwurf, die Implementierung und die Testung eines Microservice Systems berich-

[9] Diese Begri□ichkeit wurde aus (Lehvä et al., 2019) übernommen.

tet. Zur Testung dieses Microservice Systems wurde neben anderen Testverfahren auch CDCT eingesetzt. Ein dedizierter Abschnitt der Arbeit beschreibt CDCs, zählt die Werkzeuge Pact und Spring Cloud Contract auf, mit denen CDCT realisiert werden kann, und benennt einen Vorteil dieser Testart.

Im Conference Paper (Lehvä et al., 2019) werden im Rahmen einer Fallstudie Tests eines bereits bestehenden Microservice Systems mit CDC Tests komplementiert. Im Rahmen einer Evaluation wird der Vergleich zwischen CDCT und anderen Testarten auf Grundlage eines Defect Seedings und der Erfahrungen, die während der Entwicklung gemacht wurden, gezogen.

In dem Conference Paper (Li et al., 2020) wird ein speziell für Microservices ausgerichteter Testprozess vorgeschlagen, der das Energieunternehmen „State Grid Corporation of China“ dabei unterstützen soll, microservicebasierte Cloud-Anwendungen zu entwickeln. Es werden die sechs Testphasen beschrieben, aus denen der Testprozess besteht, und geeignete Testwerkzeuge zu jeder Phase benannt. Darüber hinaus wird eine Automatisierungsstrategie vorgestellt, womit sich die Durchführung der Testphasen in einer Pipeline automatisieren lässt.
Eine Phase des vorgestellten Testprozesses wird als „Contract Test“ bezeichnet, worunter insbesondere CDCT fällt. Dabei wird die Funktionsweise von CDCT erläutert, darunter auch die Nennung eines Vorteils von CDCT, Hinweise zur Implementierung unter Zuhilfenahme des Werkzeugs Pact gegeben und die „Contract Test“-Phase in die automatisierte Test-Pipeline eingegliedert.

In der Journal Publication (Raychev, 2020) werden Testverfahren für eine bestehende, zunächst monolithische Anwendung vorgestellt. Des Weiteren werden Änderungen an diesen Testverfahren beschrieben, die im Zuge des Wandels der Applikation zu einer Microservice Architektur durchgeführt wurden.
Dabei wird insbesondere auf CDCT eingegangen: Das Verfahren wird vorgestellt und es wird ein Ansatz beschrieben, mit dem eine Automatisierung von CDCT, unter Verwendung des Werkzeugs Pact, realisiert werden kann. Auch Erfahrungen bezüglich CDCT, die im Rahmen der Entwicklung gemacht wurden, werden geäußert.

### 2.4.2 Erkenntnisse aus der Literatur

Eine Übersicht der Vorteile, Nachteile, Herausforderungen und Richtlinien von CDCT, die in den insgesamt 4 ausgewählten literarischen Werken thematisiert werden, findet sich in Tabelle 2.3. Die diesbezüglich in der Literatur getroffenen Aussagen sind in den nachfolgenden Sektionen zusammengetragen.

| | Aspekt | | Literatur |
|---|---|---|---|
| Vorteile | **V1** | Erkennung von Inkompatibilitäten zwischen Services | (Koschel et al., 2021), (Lehvä et al., 2019), (Li et al., 2020) |
| | **V2** | Explizitmachung der consumerseitigen Erwartungen an Provider | (Lehvä et al., 2019), (Raychev, 2020) |
| | **V3** | Deterministische und stabile Testausführung | (Lehvä et al., 2019) |
| | **V4** | Ersetzbarkeit von Integrationstests durch CDC Tests | (Lehvä et al., 2019) |
| | **V5** | Geringe Testlaufzeit | (Lehvä et al., 2019) |
| | **V6** | Verbesserung der Teamkommunikation durch CDCs | (Lehvä et al., 2019) |
| Nachteile | **N1** | Keine Testung des funktionalen Verhaltens von Consumern | (Lehvä et al., 2019) |
| Herausforderungen | **H1** | Absprache zwischen unterschiedlichen Entwicklungsteams | (Lehvä et al., 2019), (Raychev, 2020) |
| | **H2** | Mocking externer Services | (Raychev, 2020) |
| | **H3** | Notwendigkeit zur Übermittlung von CDCs | (Lehvä et al., 2019) |
| | **H4** | Ungleicher consumer- und providerseitiger Implementierungsaufwand | (Lehvä et al., 2019) |
| Richtlinien | **R1** | Einbettung in den Testprozess | (Lehvä et al., 2019), (Li et al., 2020) |
| | **R2** | Aufbereitung von Datenbankinhalten für Testdurchführungen | (Raychev, 2020) |
| | **R3** | Einsatz einer containerisierten und virtualisierten Testausführungsumgebung | (Raychev, 2020) |
| | **R4** | Erprobung durch Pilotprojekt vor umfassender Einführung | (Lehvä et al., 2019) |
| | **R5** | Identifizierung der zu testenden Service-Interaktionen | (Lehvä et al., 2019) |
| | **R6** | Vorgehen bei bewusster consumerseitiger Contract-Änderung | (Lehvä et al., 2019) |
| | **R7** | Vorgehen bei bewusster providerseitiger Contract-Missachtung | (Lehvä et al., 2019) |

Tabelle 2.3: Auflistung der in der ausgewählten Literatur thematisierten Vorteile, Nachteile, Herausforderungen und Richtlinien von CDCT.

### Vorteile

V1: Durch CDCT kann geprüft werden, ob Inhalte, die von Providern zur Verfügung gestellt werden, den Erwartungen von Consumern gerecht werden (Li et al., 2020) und ob Änderungen an Providern die Erwartungen von Consumern verletzen. Probleme aufgrund von inkompatiblen Schnittstellen werden dadurch offenbart (Koschel et al., 2021).

Nach erfolgreicher Erzeugung eines Contracts durch den Consumer und anschließender Verifikation des Contracts durch den Provider, kann die Kompatibilität zwischen Consumer und Provider festgestellt werden. Der Contract enthält dabei die Details zu allen Interaktionen, die der Consumer vom Provider voraussetzt (Lehvä et al., 2019).

V2: CDCT kann Providern Aufschluss darüber geben, welche Services dessen Consumer sind und auf welche Weise dessen Schnittstellen durch die jeweiligen Consumer verwendet wird. Dies ermöglicht es dem Provider, sich anhand von tatsächlichen Geschäftsbedürfnissen weiterzuentwickeln, welche durch die Consumer vorgegeben werden (Lehvä et al., 2019).

Consumer erstellen gemäß ihrer Anforderungen an einen Provider Contracts, die durch den Provider zu erfüllen sind. Vonseiten des Providers ist es einfacher, die Anforderungen aus allen, an ihn gerichteten Contracts umzusetzen, als darüber mutmaßen zu müssen, welche API bereit gestellt werden muss, um die Erwartungen aller Consumer gleichermaßen zu erfüllen (Raychev, 2020).

V3: CDC Tests weisen einen hohen Grad an Determinismus auf. Die isolierte Testung beider Integrationsseiten resultiert in stabilen Testausführungen (Lehvä et al., 2019).

V4: Integrationstests können durch CDC Tests ersetzt werden (Lehvä et al., 2019).

V5: CDC Tests verfügen sie über eine geringe Laufzeit, da sie in Isolation zu anderen Services ausgeführt werden (Lehvä et al., 2019).

V6: CDCs können als Kommunikationsmedium eingesetzt werden, um die Kommunikation zwischen unterschiedlichen Entwicklungsteams zu verbessern (Lehvä et al., 2019).

Die Herausforderung H1 thematisiert weitere Aspekte der Teamkommunikation und -absprache.

### Nachteile

N1: Durch CDC Tests kann im Gegensatz zu Komponententests[10] kein funktionales Verhalten von Consumern geprüft wird. Es werden lediglich jene Teile des Quellcodes der Consumer getestet, die externe Aufrufe an Provider durchführen. Der Fokus liegt somit ausschließlich auf der Testung von Integrationen (Lehvä et al., 2019).

Ein möglicher Lösungsansatz wird in Richtlinie R1 thematisiert.

[10] Mit dem Begri Komponententests sind in diesem Zusammenhang Service Tests gemeint, wie sie in Sektion 3.1.2.2 der weitergehenden Erläuterungen vorgestellt werden.

### Herausforderungen

H1: Für eine Umsetzung von CDCT für Services, die von unterschiedlichen Entwicklungsteams implementiert werden, ist Kommunikation zwischen diesen Teams nötig. Dabei sollten Entwicklungsteams, die bisher kein CDCT einsetzen, motiviert werden, dies umzusetzen. Andernfalls könnte beispielsweise aufgrund einer zu geringen Priorisierung seitens des verantwortlichen Teams die Implementierung der CDC Tests verzögert werden (Lehvä et al., 2019).

Entwicklungsteams von Consumern können CDC Tests uneinheitlich umsetzen, wobei nicht auszuschließen ist, dass Testfälle von unterschiedlichen Teams dasselbe testen. Dies resultiert potenziell in instabilen Tests (Raychev, 2020), was im Widerspruch zu Vorteil V3 steht.

Der Vorteil V6 thematisiert ebenfalls Aspekte von Teamkommunikation.

H2: Eine Problematik besteht darin, das Verhalten externer Services durch Mocks langfristig abzubilden. Zur Lösung dieses Problems könnte eine Erhöhung der Ressourcenzuteilung speziell für diese Problematik oder eine Limitierung des Testumfangs dienen (Raychev, 2020).

H3: Im Rahmen von CDCT besteht die Notwendigkeit einer Contract-Übermittlung, die zwischen der consumerseitigen und providerseitigen Testung stattzufinden hat. Mithilfe der Werkzeuge Pact und Pact Broker kann die Übermittlung jedoch leicht umgesetzt werden (Lehvä et al., 2019).

H4: Die consumerseitige Implementierung der CDC Tests ist aufwendiger als die providerseitige Implementierung. Consumerseitig müssen erwartete Anfragen und Antworten definiert werden, die bei der Testdurchführung mit den tatsächlichen Anfragen und Antworten verglichen werden. Providerseitig muss hingegen sichergestellt werden, dass der Provider verfügbar ist und dass dessen Datenbank mit passenden Inhalten gefüllt ist (Lehvä et al., 2019).

### Richtlinien

R1: CDCT leistet einen durchaus spezifischen Beitrag im Entwicklungsprozess. Es wird daher empfohlen, CDCT stets mit herkömmlichen Testverfahren zu komplementieren (Lehvä et al., 2019).

In einer Testpyramide können CDC Tests zu anderen Testverfahren ins Verhältnis gesetzt werden. Je weiter unten sich ein Testverfahren in der Pyramide befindet, desto mehr Testfälle dieses Verfahrens sind empfohlen und desto geringer ist dessen Entwicklungsaufwand (Lehvä et al., 2019) bzw. dessen Kosten (Li et al., 2020). Von unten nach oben sind folgende Testverfahren vertreten: Unit Test, Service Test, CDC Test, E2E Test (Lehvä et al., 2019; Li et al., 2020). Der Integrationstest ist dabei entweder zwischen Unit Test und Service

Test (Li et al., 2020) oder zwischen CDC Test und E2E Test (Lehvä et al., 2019) einzuordnen.
Für die Bildung einer automatisierten Test-Pipeline können die Testverfahren gemäß ihrer Reihenfolge aus der Testpyramide sequenziell aneinandergereiht und in dieser Reihenfolge ausgeführt werden (Li et al., 2020).

R2: Bestehende Datenbankinhalte sollten in eine leichtgewichtige Form überführt werden. Das Ziel ist es sicherzustellen, dass Datenbanken all jene Daten enthalten, die zur Testdurchführung nötig sind (Raychev, 2020).

R3: Containerisierung und Virtualisierung sollte bei der Entwicklung von Microservices und auch für die Ausführung der CDC Tests verwendet werden (Raychev, 2020).

R4: CDCT könnte zunächst nur für einige ausgewählte Funktionalitäten des Gesamtsystems umgesetzt werden. Dies ermöglicht es Entwicklungsteams, in einem kleineren Rahmen mit CDCT zu experimentieren und die Eignung von CDCT zu bewerten. Ebenso kann festgestellt werden, ob es Schwierigkeiten bei der Kommunikation innerhalb der Organisation oder zwischen Teams gibt (Lehvä et al., 2019).

Auf die Teamkommunikation wird auch in Vorteil V6 und in Herausforderung H1 eingegangen.

R5: HTTP Status Codes können für die Identifizierung von zu testenden Service-Interaktionen berücksichtigt werden. Sofern consumerseitig explizit zwischen unterschiedlichen Status Codes unterschieden wird, können daraus separate Interaktionen abgeleitet werden.

R6: Änderungen an einem Consumer können sich auf dessen Contract auswirken und diesen abändern. Sofern dies intendiert war, kann es als ein Vorschlag für eine neue Contract Version aufgefasst werden. Nach Anpassung der consumerseitigen Tests hat eine Verifikation durch den Provider zu erfolgen, um dessen Kompatibilität sicherzustellen (Lehvä et al., 2019).

R7: In manchen Situationen ist es unumgänglich, den Provider derart abzuändern, dass dieser inkompatibel zu einem oder mehreren Consumern wird. In solchen Fällen sollte die Änderung mit den Consumern kommuniziert werden, sodass die Consumer neue Versionen ihrer Contracts etablieren können, welche an die Änderungen des Providers angepasst sind (Lehvä et al., 2019).

## 2.5 Evaluation

Im Rahmen der Aktionsforschung wurden insgesamt 5 Integrationen mittels CDCT getestet. Darunter befinden sich die 3 HTTP-Integrationen zwischen UI

und Pipeline Service, UI und Storage Service sowie UI und Notification Service. Bei den anderen beiden handelt es sich um nachrichtenbasierte Integrationen zwischen Notification Service und Pipeline Service sowie Storage Service und Pipeline Service. Als Programmiersprache kam durchgängig TypeScript zum Einsatz.

Bezüglich des Testaufbaus wurde sich bei den nachrichtenbasierten Integrationen[11] jeweils dazu entschieden, zusätzlich zum eigentlichen Provider Service sowohl das zugehörige Datenbanksystem mit Outboxer-Service als auch den AMQP Message Broker in den Umfang der CDC Tests mit einzubeziehen. Dies ermöglichte eine Erfassung des gesamten Prozesses der Nachrichtenentstehung und -sendung. Des Weiteren war es im Rahmen der Testung von UI und Storage Service notwendig, den PostgREST Service und das Datenbanksystem des Storage Services mit einzubeziehen, da der PostgREST Service die vom UI verwendete REST-API anhand des Schemas der zugehörigen Datenbank bereitstellt. In Sektion B des Anhangs befinden sich hierzu noch Abbildungen, welche die Testumfänge der umgesetzten CDC Tests verbildlichen.

Für die Ausführung der CDC Tests kamen Docker und Docker Compose zum Einsatz. Dies erleichterte die Umsetzung der zuvor beschriebenen, komplexeren Testaufbauten. Hintergrundinformationen zu Docker und Docker Compose werden in Sektion 3.2.2 der weitergehenden Erläuterungen vermittelt.

Der Austausch von Contract-Dateien zwischen Consumern und Providern wurde letztendlich im Rahmen von GitHub Actions durch sog. Artifacts realisiert. Insbesondere wurde sich gegen den Einsatz des Pact Brokers entschieden, da alle Microservices des ODS bezüglich deren Versionskontrolle innerhalb eines einzigen Repositories verwaltet werden.

Weitere Informationen zum Testaufbau und der Einbindung der CDC Tests in die bestehende CI Pipeline des ODS werden in Sektion 3.3 der weitergehenden Erläuterungen thematisiert.

Durch die Umsetzung der CDC Tests konnten insgesamt 4 Softwarefehler aufgedeckt werden, die allesamt in HTTP-Integrationen angetroffen wurden. Die Ursache für zwei dieser Fehler war, dass Consumer und Provider abweichende Wertebereiche für selbige JSON-Attribute von übermittelten Nutzdaten festgelegt hatten. Die anderen beiden Fehler resultierten daraus, dass Consumer und Provider abweichende Festlegungen darüber hatten, ob selbige JSON-Attribute von übermittelten Nutzdaten als optional oder obligatorisch anzusehen sind.

Im letzten Aktionsforschungszyklus wurden weitere CDC Tests durch 3 Entwickler des ODS für die HTTP-Integration zwischen dem UI und dem Notification Service realisiert. Bei der Umsetzung wurde sich insgesamt stark an den bereits

[11] Hiermit sind jene Integrationen gemeint, bei denen asynchrone Nachrichten über den AMQP Message Broker versendet werden.

umgesetzten CDC Tests orientiert, die in vorherigen Zyklen entwickelt wurden. Auf die Erkenntnisse, welche aus den anschließenden Interviews mit den Entwicklern hervorgegangen sind, wird in der nachfolgenden Sektion eingegangen.

### 2.5.1 Interview mit Entwicklern des JValue Open Data Service

Aus den Interviews ergab sich, dass es bei der Identifizierung der zu testenden Interaktionen Unterschiede zwischen den 3 Entwicklern gab. Einer der 3 Entwickler prüfte etwa im Rahmen einer Interaktion den Grenzwert „Not a Number“, indem dieser anstatt einer gültigen ID versendet wurde. Im Interview gab dieser Entwickler an, sowohl den Quellcode des Consumers als auch den des Providers für die Identifikation der zu testenden Interaktionen betrachtet zu haben, während die anderen beiden Entwickler, die diesen Grenzwert nicht explizit geprüft haben, ausschließlich den Quellcode des Consumers betrachtet haben.

Im weiteren Verlauf der Interviews wurde der Vergleich zwischen CDCT und anderen Testarten, die im ODS zum Einsatz kommen, gezogen. Dabei gaben 2 der 3 Befragten an, dass für CDCT meist ein zusätzlicher Einarbeitungsaufwand nötig ist, da es die üblichen Kenntnisse eines Entwicklers übersteigt. Ebenfalls 2 der 3 Entwicklern gaben an, dass der Entwicklungsaufwand der CDC Tests größer als jener der Unit Tests, jedoch geringer als jener der Integrationstests ist. Alle 3 Entwickler gaben an, dass die CDC Tests eine geringere Laufzeit als die Integrationstests aufweisen.

Gegen Ende der Interviews wurden die Entwickler gefragt, welche Vorteile, Nachteile, Herausforderungen und Richtlinien sie mit CDCT assoziieren. Eine Übersicht über die genannten Inhalte ist Tabelle 2.4 zu entnehmen.

### 2.5.2 Defect Seeding

Bei der Durchführung des Defect Seedings wurden insgesamt 53 unterschiedliche Integrationsfehler, die potenziell durch die CDC Tests aufgedeckt werden konnten, in das ODS-System eingebracht. Von diesen 53 Fehlern konnten jedoch potenziell nur 12 durch den E2E Test aufgedeckt werden, da in diesem das UI nicht involviert ist und dort somit keinerlei HTTP-Integrationen getestet werden. Anderweitige Testarten, wie etwa der Integrationstest, fallen gänzlich aus der Betrachtung, da hierbei keine Service-Integrationen getestet werden.

Die CDC Tests konnten schließlich 41 der 53 Fehler aufdecken, darunter 9 der 12 Fehler, die potenziell durch den E2E Test aufgedeckt werden konnten. Der E2E Test konnte hingegen 4 der 12 Fehler aufdecken, wobei diese 4 Fehler ebenfalls durch die CDC Tests aufgedeckt wurden. Eine Übersicht der eingebrachten Fehler und der zugehörigen Testergebnisse ist Sektion D des Anhangs zu entnehmen.

| | Aspekt | Beschreibung |
|---|---|---|
| Nachteile | Keine Testung funktionaler Eigenschaften | Durch CDCT wird ausschließlich Interoperabilität und keine Funktionalität geprüft. |
| Herausforderungen | Wissensaufbau für Entwickler*innen | Entwickler*innen müssen sich meist zusätzliches Wissen aneignen, bevor sie dazu imstande sind, CDC Tests zu entwickeln. |
| | Notwendigkeit zur Übermittlung von Contracts | Contracts, die durch consumerseitige Tests erzeugt werden, müssen den providerseitigen Tests zur Verfügung gestellt werden. |
| Richtlinien | Softwareentwurfsanpassungen für erleichterte Umsetzung | Die Umsetzung von Microservices gemäß hexagonaler Architektur, die Verwendung von Dependency Injection und API-Anpassungen können die Entwicklung von CDC Tests vereinfachen. |
| | Befolgung des Robustness Principle (Postel, 1981) | Consumer sollten in ihren Contracts nur jene Inhalte von Providern fordern, die sie tatsächlich von ihnen erwarten und unerwartete Inhalte tolerieren. |

Tabelle 2.4: Nachteile, Herausforderungen und Richtlinien, die von mindestens 2 der 3 befragten Entwickler mit CDCT assoziiert wurden. Assoziierte Vorteile sind nicht vertreten, da kein Vorteil von mehr als einem Entwickler genannt wurde.

Gelegentlich kam es vor, dass aufgrund einer Fehlereinbringung CDC Tests angepasst werden mussten, sodass der TypeScript-Quellcode im Anschluss wieder fehlerfrei kompiliert werden konnte. In 5 Fällen wurde nach einer consumerseitigen Fehlereinbringung eine diesbezügliche Interaktion in den CDC Tests ergänzt, ohne die der Fehler nicht aufgedeckt worden wäre. Dies wurde jedoch nur dann durchgeführt, wenn auch zuvor im Rahmen der regulären Entwicklung der CDC Tests eine derartige Interaktion abgedeckt worden wäre.

In einem Fall wurde aufgrund eines zu losen Matchings eine eingebrachte Inkompatibilität durch die CDC Tests nicht aufgedeckt. In den 11 weiteren Fällen, die unaufgedeckt blieben, handelte es sich um Änderungen an Wertebereichen von JSON-Attributen und Query- bzw. Pfad-Parametern. Die Tabelle 2.5 gibt einen Überblick, welche dieser Fehlerarten eine Inkompatibilität hervorrufen und wie diese potenziell durch CDCT hätten aufgedeckt werden können. Die Grundannahme ist dabei, dass die consumer- und providerseitigen Wertebereiche vor Einbringung der Änderung gleich gewesen sind.

W1 kann durch die Ergänzung einer Interaktion aufgedeckt werden, in welcher der Consumer neue Werte seines vergrößerten Wertebereichs sendet. Die Aufdeckungsvoraussetzungen von W2, W3 und W4 könnten möglicherweise in bereits getesteten Interaktionen erfüllt werden. Falls dies für W2 oder W3 nicht der Fall ist, hat der Consumer jedoch keinen Anhaltspunkt dafür, dass neue Interaktionen zur Fehleraufdeckung ergänzt werden müssten, da die Änderungen ausschließlich im Provider vorgenommen wurden. Um hingegen W4 anderweitig gezielt aufzu-

| | Service | Medium | Wertebereichs-änderung | Aufdeckungsvoraussetzung |
|---|---|---|---|---|
| **W1** | Consumer | HTTP-Anfrage | Vergrößerung des Bereichs der gesendeten Werte | Der Consumer testet eine Interaktion, in der er neue Werte seines vergrößerten Wertebereichs sendet. |
| **W2** | Provider | HTTP-Anfrage | Verkleinerung des Bereichs der akzeptierten Werte | Der Consumer testet eine Interaktion, in der er Werte außerhalb des verkleinerten Wertebereichs sendet. |
| **W3** | Provider | HTTP-Antwort, asynchrone Nachricht | Vergrößerung des Bereichs der gesendeten Werte | Der Provider sendet im Rahmen einer getesteten Interaktion neue Werte seines vergrößerten Wertebereichs. |
| **W4** | Consumer | HTTP-Antwort, asynchrone Nachricht | Verkleinerung des Bereichs der akzeptierten Werte | Der Provider sendet im Rahmen einer getesteten Interaktion Werte außerhalb des verkleinerten Wertebereichs. |

Tabelle 2.5: Übersicht über Integrationsfehler, die durch Wertebereichsänderungen hervorgerufen werden können. Die Aufdeckungsvoraussetzung gibt ein notwendiges Kriterium für die Aufdeckung des Fehlers durch CDCT mittels Pact an.

decken, müsste der Consumer den Provider dazu anweisen, in einer Interaktion Werte außerhalb des akzeptierten Wertebereichs zu senden, um so einen fehlschlagenden CDC Test herbeizuführen. Dies würde jedoch den Prinzipien von Pact widersprechen, da der Contract eines Consumers ausschließlich dessen Erwartungen an den Provider beinhalten sollte (Pact Foundation, 2021).

## 2.6 Diskussion

Beim Defect Seeding stellte sich heraus, dass die entwickelten CDC Tests die eingebrachten Integrationsfehler in den meisten, jedoch nicht in allen Fällen aufdecken konnten. Der Vorteil V1 aus der Literaturrecherche konnte daher nicht gänzlich bestätigt werden.
Im Rahmen des Interviews wurde durch einen befragten Entwickler der Vorteil benannt, dass die Erwartungen von Consumern explizit in Contracts ausgedrückt werden, wodurch Einfluss auf die Evolution des Providers genommen werden kann. Dies steht im Einklang zu Vorteil V2.
Des Weiteren führte ein befragter Entwickler den schnellen Feedback-Zyklus an, den CDC Tests während der Entwicklung bieten. Auch im Vergleich zu anderen Testarten konnten alle 3 Entwickler bestätigen, dass die Laufzeit von CDC Tests verhältnismäßig gering ist. Vorteil V5 konnte daher bestätigt werden.

Zwei Entwickler nannten, dass durch CDCT die Funktionalität von Services nicht

ausgiebig getestet wird, was im Einklang zu den Erfahrungen aus der Aktionsforschung und zu Nachteil N1 steht.
Das Defect Seeding zeigte auf, dass CDCT nur eine limitierte Fähigkeit zur Aufdeckung von Integrationsfehlern aufweist, wenn diese aus Änderungen an Wertebereichen hervorgehen. Insbesondere bei derartigen providerseitigen Änderungen sollte sich daher über die CDC Tests hinaus mit dem Entwicklungsteam des Consumers abgestimmt werden. Ansonsten hätte das Team des Consumers keinen Anhaltspunkt dafür, dass der Provider eine Änderung durchgeführt hat, die potenziell eine Inkompatibilität hervorruft. Des Weiteren konnte bei der Umsetzung von CDC Tests durch den Consumer nicht ausgedrückt werden, welche Werte durch den Provider künftig nicht mehr versendet werden dürfen, womit manche eingebrachte Integrationsfehler gezielt hätten aufgedeckt werden können.

Die Herausforderung H3, welche besagt, dass durch Consumer erzeugte Contracts den Providern für deren Testdurchführung zur Verfügung gestellt werden muss, wurde auch durch zwei Entwickler benannt. In der Aktionsforschung stellte sich heraus, dass dies beispielsweise durch Versionskontrollsysteme, Artifacts im Rahmen von GitHub Actions oder den Pact Broker bewältigt werden könnte.
Auf Nachfrage hin bestätigten alle 3 Entwickler die Herausforderung H4. Der Aufwand, um die consumer- und providerseitigen Strukturen umzusetzen, die grundsätzlich zur Testdurchführung nötig sind, ist zwar vergleichbar, jedoch stellt das anschließende Ergänzen von Interaktionen üblicherweise einen größeren Aufwand auf Consumer- als auf Provider-Seite dar. In der Aktionsforschung wurde zuvor dieselbe Erfahrung gemacht.

Ein Entwickler assoziierte die Verwendung von Docker als Richtlinie für CDCT, was im Einklang zu Richtlinie R3 steht. Im Rahmen der Aktionsforschung wurde jedoch die Erfahrung gemacht, dass die Verwendung von Docker ausschließlich die Umsetzung komplexerer Testaufbauten[12] vereinfachen konnte.
Bezüglich Richtlinie R5 legten Erkenntnisse aus den Interviews mit den Entwicklern nahe, dass Interaktionen potenziell übersehen werden, sofern der Quellcode des Providers nicht zur Identifikation von zu testenden Interaktionen mit einbezogen wird. Hierzu hat es sich zuvor in der Aktionsforschung bewährt, zunächst Interaktionen nur anhand des Quellcodes des Consumers zu identifizieren, um die Erwartungen des Consumers an den Provider möglichst unvoreingenommen zu erfassen. Anschließend wurde jedoch zusätzlich noch der Quellcode des Providers erfasst, um ggf. weitere Interaktionen zu identifizieren, die bislang nicht identifiziert wurden.

Mit dem Ziel zu vermeiden, dass Consumer und Provider unterschiedliche Auffassungen darüber haben, ob ein Attribut als optional oder obligatorisch anzusehen ist, wurden im Rahmen der Aktionsforschung in getesteten Interaktionen pro

[12] Damit sind Testaufbauten gemeint, in denen es nötig war, neben dem zu testenden Microservice auch anderweitige Services, Datenbanksysteme oder den Message Broker hochzufahren.

optionalem Attribut stets zwei Fälle abgedeckt: Zum einen, dass das optionale Attribut in der Anfrage, Antwort oder Nachricht präsent ist und zum anderen, dass es nicht präsent ist. Im Defect Seeding scheint sich dieser Ansatz bewährt zu haben, da die dabei eingebrachten Integrationsfehler, die mit optionalen Attributen in Verbindung standen, durch die CDC Tests allesamt aufgedeckt werden konnten.

Des Weiteren konnte das Robustness Principle (Postel, 1981) in den CDC Tests aufgrund einer technischen Limitation von Pact nicht gänzlich umgesetzt werden. Der Grund hierfür war, dass der Consumer in HTTP-Interaktionen, sofern er Erwartungen an Status Codes hat, stets einen konkreten Status Code zu fordern hat. Das UI des ODS akzeptiert jedoch meist gesamte Klassen von Status Codes[13], was durch Pact nicht ausgedrückt werden konnte. Dies führte dazu, dass die im Contract festgehaltenen Erwartungen unnötigerweise strikter als die tatsächlichen Erwartungen des Consumers formuliert werden mussten.

Insgesamt gaben die Entwickler des ODS jedoch an, dass sie die CDC Tests als gut befunden haben und diese auch in Zukunft beibehalten wollen.

## 2.7 Schlussfolgerungen

In dieser Arbeit konnten durch eine strukturierte Literaturrecherche Vorteile, Nachteile, Herausforderungen und Richtlinien von CDCT erhoben werden, die mit den Erfahrungen aus der Aktionsforschung sowie der abschließenden Evaluation abgeglichen wurden.
Demnach konnten die Vorteile V2 und V5, der Nachteil N1 und die Herausforderungen H3 und H4 bestätigt werden und es ergaben sich neue Erfahrungswerte zu den Richtlinien R3 und R5. V1 konnte hingegen nicht gänzlich bestätigt werden. Als weiterer Nachteil wurde ein limitiertes Aufdeckungspotenzial für manche Integrationsfehler festgestellt, die aufgrund von Wertebereichsänderungen entstehen können. Eine zusätzliche Herausforderung stellt der oftmals nötige Wissensaufbau für Entwickler*innen dar. Des Weiteren ergaben sich auch zwei neue Richtlinien: Zum einen sollten gewisse Anpassungen am Softwareentwurf vorgenommen werden, um die Entwicklung von CDC Tests zu erleichtern und zum anderen sollte das Robustness Principle (Postel, 1981) bei der Umsetzung von CDC Tests befolgt werden.
Aspekte von CDCT bezüglich Teamkommunikation und Schnittstellen-Evolution konnten im Rahmen der Aktionsforschung nicht untersucht werden, da es beim ODS keine separaten Entwicklungsteams pro Microservice gab und es nach Einführung der CDC Tests zu keinen Schnittstellenänderungen gekommen ist. Des

[13] Status Codes mit selbiger führender Zi□er gehören einer Klasse an. Beispielsweise gehören die beiden Status Codes `100` und `101` der Klasse „Informational `1xx`“ an (Fielding et al., 1999).

Weiteren ist unklar, ob neben den aufgefundenen Limitationen des Fehleraufdeckungspotenzials noch weitere Limitationen für gewisse Integrationsfehler existieren, die in dieser Arbeit nicht betrachtet wurden und ob diese Limitationen auch für andere Testverfahren gelten. Ein umfassenderes Defect Seeding oder Erfahrungen aus der Praxis könnten zukünftig Anhaltspunkte hierfür bieten.

# 3 Weitergehende Erläuterungen

## 3.1 Grundlagen

Diese Sektion dient der Vermittlung von Grundlagen zu Microservice Architekturen und etablierten Testverfahren für Microservice-Systeme.

### 3.1.1 Microservice Architekturen

Microservice-Systeme bestehen aus einzelnen Services, die isoliert voneinander in unterschiedlichen Prozessen oder virtuellen Maschinen ausgeführt werden. Dabei können die Services auch auf unterschiedlichen physischen Rechnern verteilt sein (Wolff, 2017).
Die Kommunikation zwischen Microservices erfolgt über Netzwerkverbindungen. Microservices stellen ihre Funktionalität durch eine Application Programming Interface (API) zur Verfügung und nehmen bei Bedarf die APIs anderer Microservices in Anspruch. Gemäß Newman (2021) sollte beim Entwurf derartiger APIs das Prinzip des Information Hiding befolgt werden, welches auf die Arbeit von Parnas (1971) zurückgeht. Bezogen auf Microservices besagt dieses, dass ein Service ein Maximum seiner Informationen in sich selbst verbergen und nur das nötige Minimum an Informationen über seine Schnittstelle nach außen offenbaren sollte. Verborgene Implementierungsdetails können dadurch frei geändert werden, solange die Kompatibilität zu jenen Services aufrecht erhalten wird, die die zugehörige Schnittstelle beanspruchen. Dies trägt zu einer losen Kopplung der Microservices bei (Newman, 2021).

Für die persistente Datenspeicherung wird angestrebt, dass jeder Microservice seine Daten selbst verwaltet. Demnach nutzen Microservices üblicherweise ein eigenes Datenbanksystem pro Service. Um als Service an Daten einer fremden Datenbank zu gelangen, sollte daher statt einer Datenbankanfrage eine Anfrage an den zugehörigen Microservice gesendet werden. Auch dieses Konzept trägt dazu bei, Information Hiding umzusetzen, indem der Mechanismus zur dauerhaften Speicherung von Daten gegenüber anderen Services verborgen wird (Newman, 2021).

Wie der Name Microservices bereits impliziert, stellt eine solche Architektur auch Beschränkungen an die Gröÿe der Services. Es existieren jedoch unterschiedliche Au assungen, woran die Gröÿe eines Microservices bemessen werden sollte. Newman (2021) nennt beispielsweise die Verständlichkeit oder auch den Schnittstellenumfang von Services als Kriterium. Newman (2021) und Wol (2017) führen des Weiteren die Softwaremetrik Lines of Code an, die zwar als Indikator für die Gröÿe herangezogen werden kann, allerdings auch stark von der eingesetzten Technologie des jeweiligen Services abhängt. Wol (2017) sowie Lewis und Fowler (2014) beziehen sich auÿerdem auch auf die Anzahl von Mitarbeitenden eines Entwicklungsteams und damit einhergehend auf die Anzahl von Microservices, für die ein solches Team zuständig sein kann.

Bei Microservices ist es üblich, dass pro Service ein Entwicklungsteam die Verantwortung trägt und dessen Entwicklung durchführt. Dies wird in Abbildung 3.1 veranschaulicht. Eine Teamaufteilung anhand der Servicegrenzen erfordert allerdings die Bildung von funktionsübergreifenden Teams (Wol , 2017). Im Bezug zur Abbildung würde dies beispielsweise bedeuten, dass jedes der drei Teams Entwickler*innen mit Expertisen in den Bereichen User Interface (UI), Backend und Datenbanken beinhalten sollte.


Abbildung 3.1: Beispiel für die Teamorganisation bei der Entwicklung von Microservices. Die drei gezeigten Microservices bestehen jeweils aus den Teilen UI, Backend und Datenbank. Die Verantwortlichkeiten der einzelnen Teams, die durch die gestrichelten Bereiche gekennzeichnet sind, verlaufen dabei anhand der Servicegrenzen. Diese Abbildung stammt ursprünglich aus (Wol , 2017) und wurde abgeändert übernommen.

In diesem Zusammenhang bietet Conway's Law eine wichtige Erkenntnis:

> *„Any organization that designs a system [...] will inevitably produce a design whose structure is a copy of the organization's communication structure."*
>
> Melvin E. Conway (1968)

Die Kommunikationsstrukturen einer Organisation bestimmen demnach maßgeblich die resultierende Softwarearchitektur. Indem bei der Teamorganisation wie zuvor beschrieben vorgegangen wird, kann Conway's Law bei der Entwicklung von Microservices dazu genutzt werden, den Koordinierungsaufwand unterschiedlicher Entwicklungsteams gering zu halten und möglichst unabhängige Services zu entwickeln. (Wolff, 2017).

Die Aufteilung des Gesamtsystems in einzelne Microservices wird üblicherweise anhand der Geschäftsdomäne vorgenommen, wodurch eine hohe Kohäsion der Services bezüglich der Geschäftsfunktionalitäten erzielt wird. Der Ursprung für diese Vorgehensweise liegt im Domain-Driven Design (DDD) (Newman, 2021).

In der nachfolgenden Sektion erfolgt zunächst ein kurzer Exkurs über monolithische Architekturen, wonach sowohl positive Eigenschaften als auch Herausforderungen von Microservices thematisiert werden.

#### 3.1.1.1 Exkurs über monolithische Architekturen

In der Literatur werden monolithische Systeme oftmals als gegensätzlicher Ansatz zu Microservice-Systemen genannt. Ein Monolith ist im Wesentlichen ein Softwaresystem, das im Rahmen des Deployments stets als Ganzes ausgeliefert werden muss. Der innere Aufbau kann allerdings durchaus modular und in manchen Fällen sogar verteilt sein. Oftmals kommen auch Datenbanksysteme für die dauerhafte Speicherung von Daten zum Einsatz (Newman, 2021).

Die Skalierung von monolithischen Systemen erfolgt üblicherweise durch die Ausführung mehrerer Instanzen des Monolithen. Eingehende Anfragen können dann mithilfe eines vorgeschalteten Lastverteilungsmechanismus auf die einzelnen Instanzen aufgeteilt werden (Namiot & Sneps-Sneppe, 2014). Eine Gegenüberstellung der Skalierung von Microservices und Monolithen findet sich in der nachfolgenden Sektion 3.1.1.2.

Insbesondere bei großen, umfangreichen Monolithen kann es Entwickler*innen schwer fallen, das System zu verstehen und Änderungen vorzunehmen. Des Weiteren lassen sich die eingesetzten Technologien oftmals nur mit großem Aufwand ersetzen (Namiot & Sneps-Sneppe, 2014).

#### 3.1.1.2 Positive Eigenschaften

Aufgrund dessen, dass es sich bei Microservices um verteilte Systeme handelt und die Services gemäß DDD um die Geschäftsdomäne modelliert sind, können sie bedarfsgerecht skaliert werden. Damit ist gemeint, dass, je nach aktueller Systemlast, zusätzliche Service-Instanzen von stark ausgelasteten Services hinzugezogen werden können. Dies erlaubt eine Verteilung der gesamten Last unter den einzelnen Instanzen (Wolff, 2017). Abbildung 3.2 stellt in diesem Zusammenhang die unterschiedlichen Skalierungsansätze von Microservices und monolithischen Systemen gegenüber.

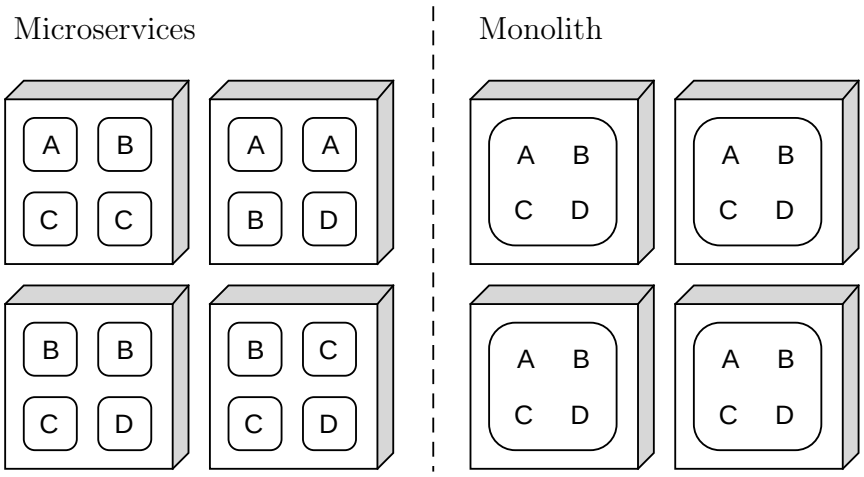


Abbildung 3.2: Gegenüberstellung einer beispielhaften Skalierung eines Microservice-Systems und eines Monolithen mit gleichem Funktionsumfang. Die Buchstaben A bis D stehen für Geschäftsfunktionalitäten des Gesamtsystems, die im Falle der Microservices durch einzelne Services realisiert sind. Die abgerundeten Quadrate in Quadern symbolisieren Prozesse, die auf entsprechenden Rechnern ausgeführt werden. Diese Abbildung wurde sinngemäß aus (Lewis & Fowler, 2014) übernommen.

Dabei wird deutlich, dass bei Microservices speziell jene Services skaliert werden können, deren Geschäftsfunktionalität zu einem gegebenen Zeitpunkt stark nachgefragt wird. Monolithen bieten diese Möglichkeit hingegen nicht: Eine Skalierung erfolgt lediglich horizontal, indem die Last auf weitere Instanzen desselben Monolithen verteilt wird.

Eine weitere Folge der Aufteilung von Microservices gemäß DDD ist es, dass Anpassungen meist nur einen einzelnen Service betreffen und sich nicht über mehrere Services hinweg erstrecken. In diesem Fall kann das für einen Microservice ver-

antwortliche Entwicklungsteam die Änderungen durchführen, ohne dass weitere Entwicklungsteams involviert sein müssen (Newman, 2021).

Des Weiteren können Microservice-Architekturen derart ausgelegt werden, dass Fehler oder auch Ausfälle einzelner Services toleriert werden können. Um dies zu erreichen, sollten sich Fehler eines Services nicht im restlichen System ausbreiten. Microservices sollten daher mit Fehlern oder Ausfällen anderer Services umgehen können, indem im Fehlerfall beispielsweise Standardwerte angenommen werden oder die eigene Funktionalität entsprechend eingeschränkt wird. (Wolff, 2017).

Ein weiterer Freiheitsgrad, den Microservice-Architekturen ermöglichen, ist die Technologiewahl der Services. Aufgrund dessen, dass Microservices lediglich über Netzwerkverbindungen miteinander kommunizieren, können pro Service unterschiedliche Technologien zum Einsatz kommen. Dadurch können gezielt jene Technologien eingesetzt werden, die besonders gut für die Umsetzung des jeweiligen Services geeignet sind (Newman, 2021).

Zudem kann das Deployment von Microservices, aufgrund deren loser Kopplung untereinander, unabhängig voneinander stattfinden. Dies bedeutet, dass Änderungen an einem Service ausschließlich das Deployment dieses Services nach sich ziehen und nicht des gesamten Systems. Dies beschleunigt nicht nur den Deployment-Prozess, sondern reduziert auch dessen Risiko, indem auftretende Fehler nach dem Deployment üblicherweise einem Microservice zugeordnet werden können und anschließend ein Rollback dieses Services durchgeführt werden kann (Newman, 2021).

#### 3.1.1.3 Herausforderungen

Die Freiheit bei der Technologiewahl kann dazu führen, dass viele verschiedene Technologien in unterschiedlichen Microservices eingesetzt werden. Dies kann die Komplexität des Gesamtsystems erhöhen, sodass Entwickler*innen die Funktionsweise des Gesamtsystems gegebenenfalls nicht mehr verstehen können. Außerdem wird im Rahmen des Refactorings die Verschiebung von Funktionalitäten zwischen Microservices erschwert, wenn die Services durch unterschiedliche Technologien realisiert sind. Zur Lösung des Problems kann dabei eine partielle Vereinheitlichung von eingesetzten Technologien beitragen (Wolff, 2017).

Aufgrund dessen, dass Microservices ausschließlich über Netzwerkverbindungen miteinander kommunizieren, erhöhen sich Latenzen des Systems. Diese entstehen durch die zusätzliche Notwendigkeit zur Serialisierung, Übertragung und Deserialisierung von Nachrichten, was bei der Kommunikation innerhalb eines Prozesses nicht nötig wäre (Newman, 2021).

Durch die Verteilung von Daten über viele Microservices und Datenbanksysteme hinweg, ist die Wahrung von Konsistenz schwer zu erreichen. Als Konsequenz

wird daher meist auf starke Konsistenz verzichtet[1].
Des Weiteren erschwert die Erstreckung der Daten auch das Reporting. Dazu müssen die Daten aus den unterschiedlichen Datenbanksystemen der Microservices zusammengeführt und verarbeitet werden. Dies kann jedoch über eine dedizierte Reporting-Datenbank geschehen, die durch die Microservices eigenständig befüllt wird (Newman, 2021).

Ansonsten sind für ein unabhängiges Deployment von Microservices separate Versionskontrolle, Versionierungen der Services und Continuous Delivery (CD) Pipelines pro Service umzusetzen, wodurch ein erhöhter Einrichtungs- und Wartungsaufwand entsteht.
Die Ausführung von Microservice-Systemen bedarf darüber hinaus einer verteilten Infrastruktur, die dazu imstande ist, eine Vielzahl virtueller Maschinen auszuführen (Wolff, 2017).

Daraus ergeben sich auch Problemstellungen bezüglich des Testens. Insbesondere End-to-End (E2E) Tests, bei denen das Zusammenwirken eines möglichst großen Teils der Microservices geprüft wird, können oftmals nicht auf gewöhnlichen Rechnern von Entwickler*innen durchgeführt werden, da diese zur selben Zeit nur begrenzt viele Services ausführen können (Newman, 2021).

Weitere Informationen zu verschiedenen Testverfahren, die für Microservice-Systeme eingesetzt werden, sind den nachfolgenden Sektionen zu entnehmen.

### 3.1.2 Testverfahren für Microservices

In dieser Sektion werden einige Testverfahren für Microservices vorgestellt. Dabei ist zu beachten, dass es sich ausschließlich um Verfahren handelt, die vor der Produktivsetzung durchgeführt werden (engl. *pre-production testing*) (Newman, 2021).

In der Literatur wird oftmals die Testpyramide angeführt, um die verschiedenen Testverfahren zueinander ins Verhältnis zu setzen. In Abbildung 3.3 wird diese dargestellt. Je weiter unten eine Testart in der Pyramide vertreten ist, desto geringer ist dessen Laufzeit, Entwicklungs- bzw. Durchführungsaufwand und Granularität und desto mehr Testfälle dieser Testart sollten umgesetzt werden (Lehvä et al., 2019).

Ebenso wird eine Unterteilung in isoliertes und integriertes Testen vorgenommen. Beim isolierten Testen werden einzelne Microservices, oder Teile von ihnen, losgelöst von ihren Abhängigkeiten, wie anderen Microservices oder Datenbanksystemen, getestet. Beim integrierten Testen wird hingegen das Zusammenspiel

[1] Angestrebt wird dann meist letztendliche Konsistenz (engl. *eventual consistency*) (Newman, 2021). Damit ist gemeint, dass sich das System in einem inkonsistenten Zustand befinden kann, der jedoch über die Zeit hinweg gegen einen konsistenten Zustand konvergiert.

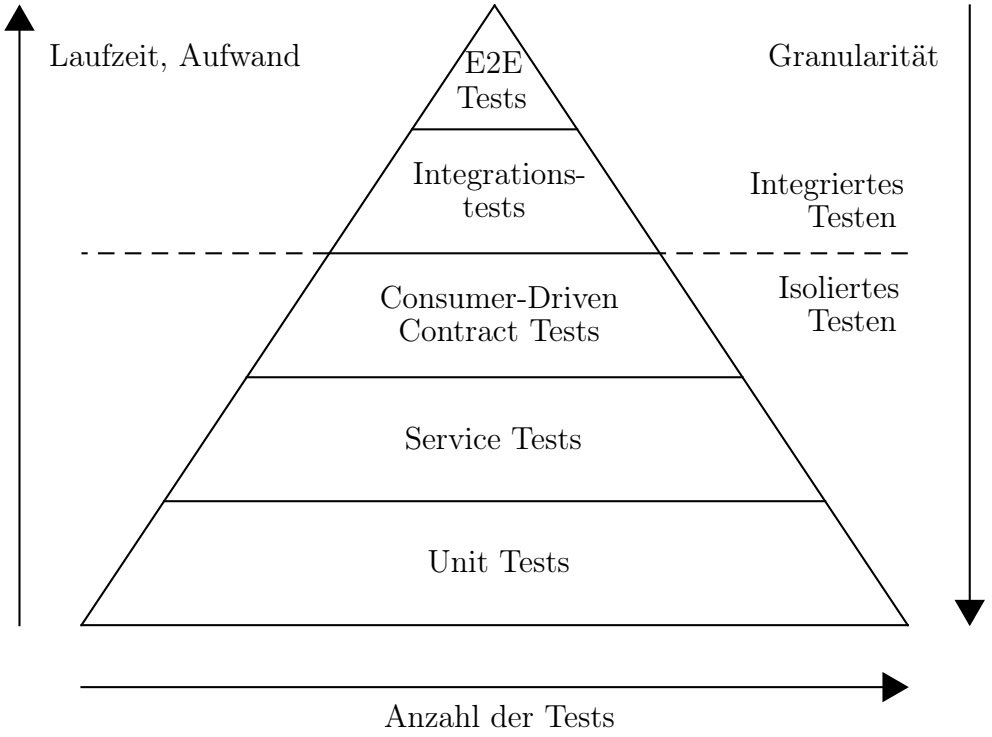


Abbildung 3.3: Testpyramide, welche Consumer-Driven Contract (CDC) Tests beinhaltet. Diese Abbildung wurde sinngemäß aus (Lehvä et al., 2019) übernommen. Ursprünglich stammt die Testpyramide aus (Cohn, 2010).

von Microservices und deren Abhängigkeiten getestet. Dabei ist es für die Testdurchführung notwendig, dass alle zu testenden Microservices und gegebenenfalls deren Abhängigkeiten zur Ausführung gebracht werden. Wie bereits in Sektion 3.1.1.3 erwähnt, kann dies durchaus eine Herausforderung darstellen.

CDC Tests nehmen bezüglich dieser Aufteilung einen besonderen Platz in der Testpyramide ein: Die Testart befindet sich noch im Bereich des isolierten Testens und wird dennoch dazu eingesetzt, Integrationen von Microservices zu testen.

In den nachfolgenden Sektionen werden schließlich die einzelnen Testverfahren aus der Testpyramide vorgestellt.

#### 3.1.2.1 Unit Test

Unit Tests testen individuelle Einheiten, aus denen Microservices bestehen. Es ist nicht strikt vorgegeben, worum es sich bei derartigen Einheiten handeln muss. Üblicherweise sind damit jedoch Funktionen bzw. Methoden (Wolff, 2017) oder

auch Klassen bzw. kleine Gruppen von Klassen (Clemson, 2014) gemeint. Um eine derartige Einheit aus seinem Umfeld zu isolieren, werden dessen Abhängigkeiten typischerweise durch Test Doubles[2] ersetzt (Clemson, 2014).

Ein Vorteil von Unit Tests ist deren kurze Laufzeit. Dadurch können Unit Tests während der Entwicklung sehr oft ausgeführt werden und liefern Entwickler*innen zeitnah Feedback zu Änderungen am Quellcode (Wolff, 2017).

#### 3.1.2.2 Service Test

Service Tests testen einen einzelnen, isolierten Microservice (Newman, 2021). Eine andere Bezeichnung für dieselbe Testart lautet Komponententest, wobei mit dem Begriff „Komponente“ (engl. *component*) in diesem Zusammenhang ein einzelner Microservice gemeint ist (Clemson, 2014).

Clemson (2014) nennt zwei unterschiedliche Ansätze, wie mit Abhängigkeiten des zu testenden Microservices umgegangen werden kann. Eine Möglichkeit besteht darin, den Microservice innerhalb eines Prozesses zu testen. Dazu werden jene Module des Microservices durch Test Doubles ersetzt, die für die Kommunikation zu externen Abhängigkeiten, wie etwa anderen Microservices oder Datenbanksystemen, verantwortlich sind. Eine andere Möglichkeit stellt eine prozessübergreifende Testung dar: Der zu testende Microservice bleibt dabei unverändert, jedoch werden die externen Abhängigkeiten des Services durch Stubs ersetzt.

Service Tests können dazu beitragen, Vertrauen in die einzelnen Microservices zu gewinnen, indem diese prüfen, ob die Verhaltensweisen der Services den Erwartungen entsprechen. Aufgrund des breiteren Umfangs im Vergleich zu Unit Tests sind die Ursachen für auftretende Fehler jedoch meist schwerer zu ermitteln (Newman, 2021).

#### 3.1.2.3 Integrationstest

Newman (2021) führt an, dass Uneinigkeit über die Bedeutung des Begriffs „Integrationstest“ herrscht. Es ist daher hervorzuheben, dass sich in dieser Arbeit auf die Begriffserklärung von Clemson (2014) berufen wird.

Im Rahmen eines Integrationstests werden mehrere Module zusammengeführt und gemeinsam als Subsystem getestet. Das Ziel ist es, anhand der Kollaboration der Module zu prüfen, ob falsche Annahmen über deren Kommunikation existieren. Bezüglich Microservices steht hierbei das Zusammenspiel mit deren externen Abhängigkeiten, wie anderen Microservices, Datenbanksystemen oder anderweitigen Fremdsystemen, im Fokus. In einem Testfall wird dazu die Interaktion zwischen einem Modul eines Microservices, das für die Kommunikation

[2] Test Double ist im Rahmen dieser Arbeit als generischer Oberbegri für Mocks und Stubs zu verstehen.

mit einer externen Abhängigkeit verantwortlich ist, und der zugehörigen externen Abhängigkeit geprüft.

Bei den Interaktionen sollten sowohl grundlegende Erfolgs-, als auch Fehlerszenarien abgedeckt werden. Um solche Fehlerszenarien gezielt herbeizuführen, kann die tatsächliche externe Abhängigkeit durch einen Stub ersetzt werden, der das Verhalten im Fehlerfall simuliert.

Bei der Umsetzung von Integrationstests kann die Zustandsverwaltung eine Herausforderung darstellen, da manche Testfälle möglicherweise voraussetzen, dass abhängige externe Systeme über bestimmte Daten verfügen (Clemson, 2014).

#### 3.1.2.4 End-to-End Test

E2E Tests testen das Gesamtsystem über öffentliche Schnittstellen, wie UIs und Service-APIs. Das primäre Ziel ist dabei, Vertrauen zu darin zu gewinnen, dass das Microservice-System die Geschäftsziele erfüllt. Des Weiteren können E2E Tests auch Aufschlüsse über die Korrektheit des Nachrichtenaustausches und der Konfiguration der Netzwerkinfrastruktur, wie Firewalls, Proxies oder Lastverteilern, geben. Externe Services, die außerhalb der eigenen Kontrolle liegen, sollten ebenfalls Teil des Testumfangs sein. Sofern die E2E Tests dadurch nicht reproduzierbar oder seiteneffektfrei umgesetzt werden können, kann es jedoch von Vorteil sein, externe Services durch Stubs zu ersetzen (Clemson, 2014).

Aufgrund des großen Testumfangs können sich lange Testlaufzeiten und falschnegative Testergebnisse ergeben, die das Deployment von Microservices verzögern. Newman (2021) nennt in diesem Zusammenhang „brüchige“ (engl. *flaky*, *brittle*) Tests, die gelegentlich aufgrund von äußeren Umständen fehlschlagen. Beispiele für derartige Umstände wären beispielsweise zur Testdurchführung benötigte Microservices, die nicht verfügbar sind, oder auch temporäre Netzwerkfehler (Newman, 2021).

#### 3.1.2.5 Consumer-Driven Contract Test

Consumer-Driven Contract Tests oder in mancher Literatur auch nur Contract Tests genannt (Clemson, 2014; Li et al., 2020; Newman, 2021) werden dazu eingesetzt, Integrationen von Services zu testen.
Miteinander interagierende Services werden dafür in zwei Rollen eingeteilt: Ein Service, der die API eines anderen Service in Anspruch nimmt, wird Consumer genannt und diejenigen Services, die APIs anbieten, welche von anderen Services in Anspruch genommen werden, gelten als Provider. Dabei ist es durchaus möglich, dass derselbe Service sowohl die Rolle eines Consumers als auch die eines Providers einnimmt (Lehvä et al., 2019).

Aus einem Verhältnis zwischen Consumern und Providern resultieren verschiede-

ne Arten von Contracts. Worum es sich bei Contracts handelt, zwischen welchen Arten unterschieden wird und inwiefern sie im Rahmen von Consumer-Driven Contract Testing (CDCT) dazu eingesetzt werden können, Integrationen von Microservices zu testen, wird in den folgenden beiden Sektionen erläutert.

### Begri□sabgrenzung unterschiedlicher Arten von Contracts

Durch Robinson (2006) wird zwischen drei verschiedenen Arten von Contracts unterschieden: Provider Contracts, Consumer Contracts und Consumer-Driven Contracts. Diese werden in den folgenden Abschnitten genauer erläutert und verglichen.

**Provider Contracts** Im Wesentlichen drücken Provider Contracts aus, welche Geschäftsfunktionalitäten ein Provider nach außen zur Verfügung stellt. Konkrete Inhalte des Contracts können dabei Strukturen der Ein- und Ausgabenachrichten, Schnittstellenoperationen, verwendete Muster für den Nachrichtenaustausch und qualitative Service-Eigenschaften, wie etwa Verfügbarkeit, Performanz, Durchsatz oder auch Sicherheit sein. Abweichungen oder anderweitige Inhalte sind darüber hinaus nicht auszuschließen.
Ein Provider Contract bildet demnach in Gänze den Funktionsumfang ab, der Consumern bereitgestellt wird. Pro Provider existiert daher genau ein Provider Contract, dessen Inhalte durch den Provider vorgegeben werden. Eine Instanz eines Provider Contracts besitzt zudem nur eine begrenzte Stabilität, da Änderungen am Service-Umfang eine neue Contract-Instanz nach sich ziehen. Dabei kann Versionierung helfen, um zwischen verschiedenen Contract-Instanzen zu unterscheiden (Robinson, 2006).

**Consumer Contracts** Erwartungen eines Consumers an einen Provider Contract werden in einem Consumer Contract festgehalten. Sofern Consumer und Provider kompatibel miteinander sind, ist der Consumer Contract als Teilmenge des Provider Contracts aufzufassen, da ggf. nur ein Teil der zur Verfügung stehenden Geschäftsfunktionalität durch den Consumer erwartet und verwendet wird.
Pro Provider kann es, je nach Anzahl der zugehörigen Consumer, durchaus mehrere, an ihn gerichtete Consumer Contracts geben. Des Weiteren verfügen Instanzen von Consumer Contracts ebenfalls nur über eine begrenzte Stabilität, da sich die Erwartungen, die im Consumer Contract festgehalten sind, durch Änderungen am Consumer wandeln können (Robinson, 2006).

**Consumer-Driven Contracts** CDCs, oder auch Consumer-Driven Provider Contracts genannt, enthalten die Erwartungen aller Consumer Contracts, die an einen Provider gerichtet sind. Demnach geht der CDC aus der Vereinigung aller dieser Consumer Contracts hervor. Im Falle der Kompatibilität zwischen einem

Provider und all seinen Consumern stellt der CDC eine Teilmenge des Provider Contracts dar. Dadurch ist es für den Provider ersichtlich, welche Teile seiner Geschäftsfunktionalität insgesamt verwendet werden und welche nicht.
Pro Provider existiert demnach genau ein CDC, dessen Inhalte durch seine Consumer vorgegeben werden. Die Stabilität einer CDC-Instanz ist dabei an die Stabilität der einzelnen Instanzen der Consumer Contracts geknüpft, auf dessen Grundlage der CDC besteht. Demnach kann eine Änderung an einem Consumer Contract eine Änderung am zugehörigen CDC nach sich ziehen (Robinson, 2006).

Anhand eines Beispiels, bestehend aus einem Provider und drei zugehörigen Consumern, werden abschließend in Abbildung 3.4 die wesentlichen Unterschiede der drei Contract Arten dargestellt.

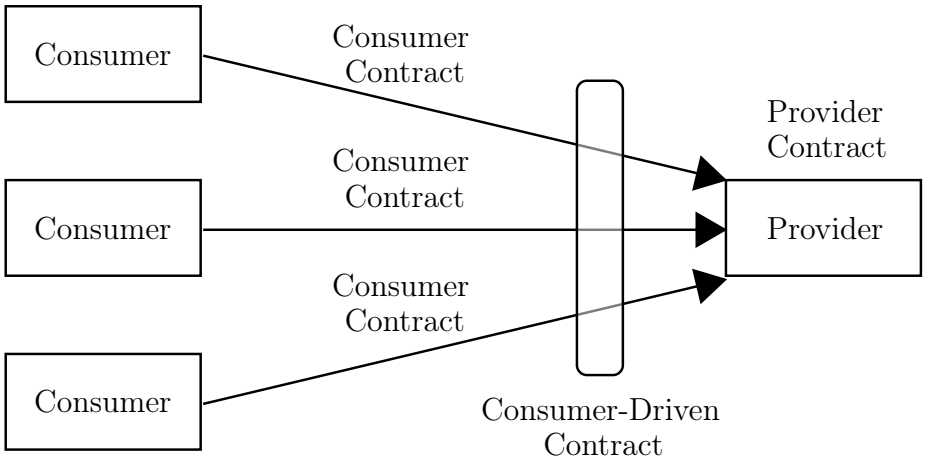


Abbildung 3.4: Veranschaulichung der Unterschiede zwischen Provider Contracts, Consumer Contracts und Consumer-Driven Contracts. Pfeile von den Consumern zum Provider symbolisieren hierbei Erwartungen der Consumer an die API des Providers. Diese Abbildung wurde sinngemäß aus (Wolff, 2017) übernommen.

Provider Contracts decken demnach die gesamten, nach außen zur Verfügung stehenden Geschäftsfunktionalitäten eines Providers ab, Consumer Contracts beinhalten Erwartungen von einzelnen Consumern, die an einen Provider Contract gerichtet sind, und CDCs bestehen aus der Vereinigung aller Consumer Contracts, die an einen Provider gerichtet sind.

### Testen von Microservice-Integrationen

CDCs können dazu eingesetzt werden, Integrationen von Microservices zu testen.

Dazu werden Consumer Contracts durch die Consumer vorgegeben und idealerweise in eine Form überführt, die der Provider dazu nutzen kann, sich anhand dessen zu verifizieren. Im Anschluss ist es nötig, dem Provider diese Contract Artefakte zugänglich zu machen, beispielsweise über das Versionskontrollsystem des Consumers oder des Providers. Zur Testdurchführung bezieht der Provider die Contract Artefakte seiner Consumer, die gemeinsam den CDC verkörpern, und verwendet diese, um zu überprüfen, ob er die Erwartungen seiner Consumer erfüllt (Wolff, 2017).

In der Praxis werden consumerseitige Erwartungen üblicherweise anhand einzelner, beispielhafter Interaktionen beschrieben. Ein Contract Artefakt enthält dann Nachrichten, die bei der Testdurchführung in konkreten Interaktionen zwischen Consumer und Provider ausgetauscht werden. Eine Interaktion, die auf Hypertext Transfer Protocol (HTTP) basiert, würde dann beispielsweise durch eine HTTP-Anfrage und eine HTTP-Antwort beschrieben werden.
Dieser Ansatz erlaubt es auch dem Consumer zu prüfen, ob er sich an seinen eigenen Contract hält. Dafür wird der Consumer im Rahmen eines Testfalls für eine Interaktion dazu veranlasst, die Anfrage der zu testenden Interaktion zu senden, woraufhin ihm dann die Antwort dieser Interaktion zugesendet wird (Richardson, 2018).

Durch den Einsatz von CDCT können Änderungen an Schnittstellen erleichtert werden, da Anforderungen an Schnittstellen explizit gemacht werden und dadurch deutlich wird, welche Teile der Provider-Schnittstelle tatsächlich durch Consumer verwendet werden und welche Teile frei veränderbar sind. Dadurch kann das Risiko, welches mit üblicherweise Schnittstellenänderungen verbunden ist, verringert werden (Wolff, 2017).

Es ist jedoch anzumerken, dass CDC Tests die Geschäftslogik von Services nicht ausgiebig testet (Lehvä et al., 2019; Richardson, 2018). Dieser und weitere Aspekte zu CDCT werden in Sektion 2.4.2 der kurzgefassten Ausarbeitung thematisiert.

Für einen abschließenden Vergleich von CDCT zu anderen Testverfahren, welche imstande dazu sind, Service-Integrationen zu testen, sind in Abbildung 3.5 die Umfänge unterschiedlicher Testverfahren dargestellt. Dabei wird ersichtlich, dass CDC Tests im Vergleich zu Integrationstests und E2E Tests beide Seiten von Integrationen isoliert voneinander testen. Diese Entkopplung wird durch den Consumer Contract ermöglicht, der vom Consumer vorgegeben wird und anschließend durch den Provider für dessen eigene Verifikation verwendet wird.

Für die Umsetzung von CDC Tests werden in der Regel Werkzeuge verwendet. Um Entwicklung der CDC Tests zu erleichtern, sollte dabei ein uniformes Werkzeug für alle Microservices eines Systems zum Einsatz kommen (Wolff, 2017).

In der Praxis wird oftmals das Werkzeug Pact für die Entwicklung von CDC Tests verwendet. Dieses wird in Sektion 3.2.1 vorgestellt.

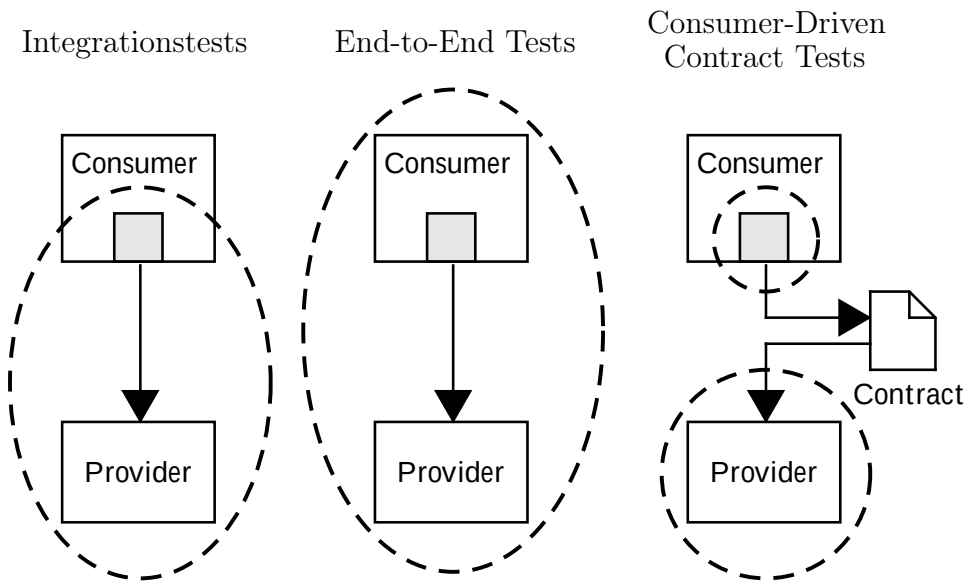


Abbildung 3.5: Vergleich der Umfänge unterschiedlicher Testverfahren zur Testung von Service-Integrationen. Gestrichelte Ellipsen zeigen auf, welche Teile der dargestellten Software durch die entsprechende Testart geprüft werden. Das graue Quadrat innerhalb der Consumer symbolisiert dabei jenes Modul, welches für die Interaktion mit dem Provider verantwortlich ist. Diese Abbildung stammt ursprünglich aus (Lehvä et al., 2019) und wurde abgeändert übernommen.

## 3.2 Werkzeuge

In dieser Sektion werden verschiedene Werkzeuge vorgestellt, die bei der Entwicklung der CDC Tests eine zentrale Rolle eingenommen haben. Thematisiert werden Pact, was für die Umsetzung der CDC Tests verwendet wurde, Docker, was zur isolierten Ausführung der CDC Tests eingesetzt wurde und schließlich GitHub Actions, da diese Technologie bei der Einbindung der CDC Tests in die bestehende Continuous Integration (CI) Pipeline des JValue Open Data Service (ODS) zum Einsatz kam.

### 3.2.1 Pact

Bei Pact handelt es sich um ein Open-Source-Werkzeug für die Realisierung von CDCT. Es existieren unterschiedliche Implementierungen von Pact, wodurch eine Vielzahl unterschiedlicher Technologien unterstützt wird. Diese verfügen allerdings über unterschiedliche Funktionsumfänge. Eine Übersichtstabelle befindet

sich hierzu in der Dokumentation[3].
Des Weiteren ist anzumerken, dass Pact nicht für jedes Projekt bzw. nicht zur Testung beliebiger Service-Integrationen geeignet ist. Einzelheiten hierzu sind ebenfalls der Dokumentation[4] zu entnehmen.

In den folgenden Sektionen wird die Funktionsweise von Pact erläutert, bewährte Praktiken, die in der Dokumentation zu finden sind, angeführt und abschließend der Pact Broker, welcher für die Integration von Pact in CI Pipelines verwendet werden kann, vorgestellt.

#### 3.2.1.1 Funktionsweise

Zur Testung einer Service-Integration wird im Rahmen von Pact eine Menge an beispielhaften Interaktionen zwischen einem Consumer und einem Provider festgelegt, die zusammengenommen einen sog. Pact bilden. Gemäß der in Sektion 3.1.2.5 etablierten Begriffe, handelt es sich bei einem solchen Pact um einen Consumer Contract.
Die Bestandteile einer Interaktion unterscheiden sich je nach Art der Interaktion. Es wird zwischen den folgenden beiden Arten unterschieden:

- HTTP-basierte Interaktionen, die durch einen Provider Zustand, eine erwartete Anfrage und eine minimale erwartete Antwort beschrieben werden.
- Nachrichtenbasierte Interaktionen[5], die durch einen Provider Zustand und eine minimale erwartete Nachricht beschrieben werden.

Dabei beschreibt ein Provider Zustand (engl. *provider state*) den Ausgangszustand des Providers, der vor der Durchführung dieser Interaktion angenommen werden sollte. Mit einer minimalen erwarteten Antwort bzw. Nachricht ist hingegen eine beispielhafte Antwort bzw. Nachricht gemeint, die nur jene Bestandteile aufweist, welche vom Consumer tatsächlich erwartet werden (Pact Foundation, 2021).

Die folgenden beiden Sektionen beschreiben den Ablauf für die Testung von HTTP-basierten Interaktionen zwischen einem Consumer und einem Provider mittels Pact, wie er in (Pact Foundation, 2021) angeführt wird. Der Ablauf ist zweigeteilt, da, wie bereits in Sektion 3.1.2.5 beschrieben, eine beidseitige, isolierte Testung der Integration vorgenommen wird.

##### Consumerseitiger Ablauf

Die Schritte, die zur consumerseitigen Testung einer HTTP-Interaktion durchlaufen werden, sind in Abbildung 3.6 dargestellt.

[3] https://docs.pact.io/roadmap/feature_support/
[4] https://docs.pact.io/getting_started/what_is_pact_good_for/
[5] Damit sind Interaktionen gemeint, bei denen pro Interaktion eine Nachricht von einem Provider an einen Consumer zugestellt wird.

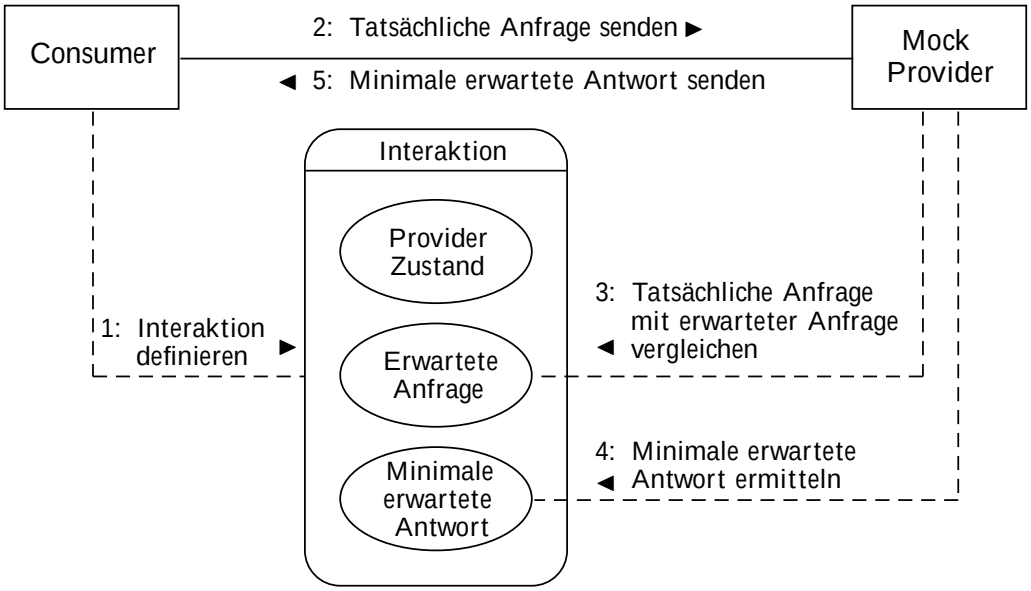


Abbildung 3.6: Testung einer HTTP-Interaktion durch einen Consumer mittels Pact. Die Nummerierung gibt an, in welcher Reihenfolge die einzelnen Schritte ablaufen. Diese Abbildung stammt ursprünglich aus (Pact Foundation, 2021) und wurde abgeändert übernommen.

Zunächst wird die zu testende Interaktion definiert, indem dessen Bestandteile an das Pact Framework übermittelt werden. Das Pact Framework stellt des Weiteren einen Mock Provider bereit, der anstelle des echten Provider Service involviert ist. Das Modul des Consumers, welches für die Interaktion mit dem Provider zuständig ist, wird nun derart aufgerufen, dass der Consumer für diese Interaktion eine tatsächliche Anfrage an den Mock Provider sendet. Nach Erhalt der Anfrage vergleicht der Mock Provider die erhaltene tatsächliche Anfrage mit der in der Interaktion definierten erwarteten Anfrage. Im Falle der Übereinstimmung, wird die minimale erwartete Antwort der Interaktion durch den Mock Provider ermittelt und an den Consumer gesendet. Nach Erhalt der Antwort, erfolgt die Anfrageverarbeitung durch den Consumer. Die daraus resultierenden Nutzdaten können abschließend durch das Unit Testing Framework überprüft werden.

Dieser Vorgang wird für alle zu testenden Interaktionen durchgeführt. Im Anschluss wird daraus eine Pact-Datei im JavaScript Object Notation (JSON) Format generiert, welche die bislang consumerseitig getesteten Interaktionen beinhaltet. Für die nachfolgende providerseitige Verifikation muss die entstandene Pact-Datei dem Provider zugänglich gemacht werden (Pact Foundation, 2021).

### Providerseitiger Ablauf

In Abbildung 3.7 sind die Schritte aufgezeigt, die im Rahmen der providerseitigen Testung einer HTTP-Interaktion durchlaufen werden.

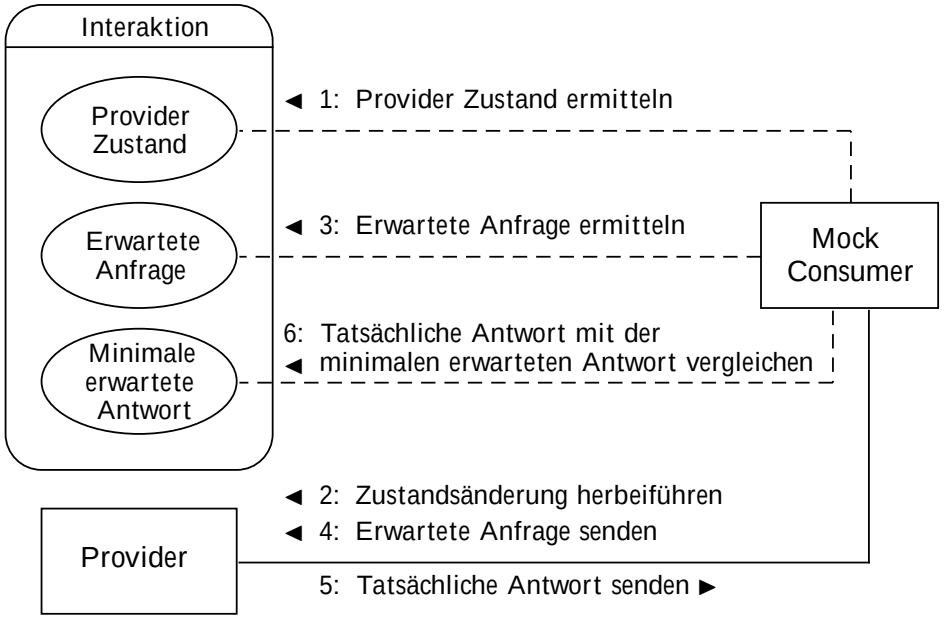


Abbildung 3.7: Testung einer HTTP-Interaktion durch einen Provider mittels Pact. Die Nummerierung gibt an, in welcher Reihenfolge die einzelnen Schritte ablaufen. Diese Abbildung stammt ursprünglich aus (Pact Foundation, 2021) und wurde abgeändert übernommen.

Zu Beginn wird der Provider Service hochgefahren, wobei dessen Abhängigkeiten gegebenenfalls durch Stubs ersetzt werden können. Durch das Pact Framework wird des Weiteren ein Mock Consumer hochgefahren, der anstelle des echten Consumer Service involviert ist. Zu Beginn der providerseitigen Testung bezieht der Mock Consumer zunächst die Pact-Datei, die die zu testenden Interaktionen beinhaltet. Dann ermittelt der Mock Consumer den Provider Zustand, welcher in der zu testenden Interaktion festgehalten ist, und veranlasst den Provider dazu, diesen Zustand anzunehmen. Im Anschluss ermittelt der Mock Consumer die erwartete Anfrage aus der Interaktion und sendet diese regulär an den Provider. Als Konsequenz sendet der Provider die tatsächliche Antwort auf diese Anfrage an den Mock Consumer, welcher die tatsächliche Antwort mit der minimalen erwarteten Antwort der Interaktion vergleicht.

Dieser Vorgang wird für alle Interaktionen wiederholt, die in der entsprechenden Pact-Datei enthalten sind. Um den CDC providerseitig zu testen, ist eine Testung

der Interaktionen aller Pact-Dateien erforderlich, die durch Consumer an diesen Provider gerichtet sind (Pact Foundation, 2021).

Die Striktheit, mit der erwartete Anfragen, Antworten oder Nachrichten mit Tatsächlichen verglichen werden, kann außerdem individuell festgelegt werden. Dies wird im Rahmen von Pact Matching genannt. Ohne den Einsatz von Matchern erfolgt striktes Matching, was einer Gleichheitsprüfung der Inhalte entspricht. Durch den Einsatz von Matchern kann der Abgleich toleranter gestaltet werden, beispielsweise indem lediglich die Übereinstimmung der Datentypen anstatt tatsächlicher Attributwerte geprüft wird oder indem reguläre Ausdrücke angegeben werden, mit denen Zeichenketten übereinstimmen müssen (Pact Foundation, 2021).

#### Unterschiede zu nachrichtenbasierten Interaktionen

Neben HTTP-Interaktionen werden durch Pact auch nachrichtenbasierte Interaktionen unterstützt. Hierzu wird von der konkreten Technologie zur Übermittlung der Nachrichten, wie beispielsweise Advanced Message Queuing Protocol (AMQP), abstrahiert, sodass der Fokus rein auf den übermittelten Nutzdaten liegt. Für die Implementierung ist es daher von Vorteil, wenn die protokollspezifische Kommunikation und die Nutzdatenverarbeitung jeweils im Quellcode der Consumer und Provider voneinander getrennt sind (Pact Foundation, 2021).

Der zuvor beschriebene Ablauf bezog sich lediglich auf HTTP-Interaktionen, jedoch ähnelt er dem Ablauf für nachrichtenbasierte Interaktionen. Im Wesentlichen werden auf Anfragen bezogene Schritte weggelassen und auf Antworten bezogene Schritte werden stattdessen auf Nachrichten bezogen.

### 3.2.1.2 Bewährte Praktiken

In der durch die Pact Foundation (2021) bereitgestellten Dokumentation finden sich diverse Praktiken, die für die Realisierung von CDC Tests mittels Pact empfohlen werden. Die nun folgenden Empfehlungen bilden allerdings nur einen Teil der in der Dokumentation enthaltenen Aspekte ab und sind daher nicht als vollständig zu erachten.

#### Allgemeines

- Ausschließlich der Nachrichtenaustausch zwischen Provider und Consumer sollte im Fokus liegen. Etwaige Seiteneffekte des Nachrichtenaustausches, wie etwa eine dauerhafte Speicherung von Daten, sollten nicht geprüft werden.
- Die Tests sollten die Geschäftslogik nicht ausgiebig testen. Lediglich ein gemeinsames Verständnis zwischen Consumer und Provider bezüglich deren

ausgetauschten Nachrichten sollte sichergestellt werden.

- Pro Testfall sollte genau eine Service-Interaktion getestet werden.

### Consumerseitige Tests

- Der Fokus sollte ausschließlich auf der Erzeugung der Anfrage und der Interpretation der Antwort bzw. Nachricht liegen.
- Die Erzeugung und der Versand von tatsächlichen Anfragen sollte ausschließlich durch das zuständige Modul des Consumers und nicht separat durch den Quellcode des Testfalls erfolgen.
- Für HTTP-Interaktionen sollte es vermieden werden, nach Anfragen, die den Zustand des Providers verändern, eine `GET`-Anfrage zur anschließenden Zustandsüberprüfung durchzuführen.
- Randomisierte Daten sollten nicht in Pacts aufgenommen werden.

**Matching** Grundsätzlich sollte das Matching gemäß des Robustness Principle[6] umgesetzt werden:

> *„Be conservative in what you do, be liberal in what you accept from others.“*
>
> J. Postel (1981)

Für ausgehende Anfragen von Consumern wird demnach ein Matching empfohlen, das so strikt wie möglich und so lose wie nötig ist. Im Gegensatz dazu wird für Antworten bzw. Nachrichten nahe gelegt, das Matching so lose wie möglich und so strikt wie nötig zu gestalten. Generell muss das Matching jedoch stets strikt genug gewählt werden, sodass die Erwartungen von Consumern an Provider ausreichend überprüft werden. Aus diesem Grund ist die zu wählende Striktheit stets im Einzelfall zu entscheiden.

### Providerseitige Tests

Sofern Änderungen am Quellcode des Providers vorgenommen werden, sollten providerseitige Tests für die Pacts aller zugehörigen Consumer durchgeführt werden. Die Motivation hierfür besteht darin, dass der CDC Test in diesem Fall als Regressionstest dient. Das Ziel ist es dabei, etwaige Inkompatibilitäten aufzudecken, welche durch die vorangegangenen Änderungen verursacht wurden.
Bei Änderungen am Quellcode eines Consumers reicht es hingegen, den consumerseitigen Test durchzuführen und nur den daraus neu entstandenen Pact durch den Provider verifizieren zu lassen.

[6] Das Robustness Principle ist auch als Postel's Law bekannt.

**Einsatz von Stubs** Generell sollten Stubs nur dann eingesetzt werden, wenn dies zwingend für die Realisierung der CDC Tests erforderlich ist. Sofern der Bedarf jedoch besteht, sollte der Quellcode für die Extraktion und Verifikation von Nutzdaten aus der Anfrage keinesfalls durch einen Stub ersetzt werden. Alternativ kann der Provider für die Testung auch unverändert hochgefahren werden und dafür dessen Abhängigkeiten, wie Datenbanksysteme oder andere Services, durch Stubs ersetzt werden.

**Provider Zustände** Für Interaktionen, in denen ein Ausgangszustand des Providers vorausgesetzt wird, sollten Provider Zustände zum Einsatz kommen. Insbesondere sollte ein Ausgangszustand keinesfalls durch vorhergehende Interaktionen herbeigeführt werden.

#### 3.2.1.3 Pact Broker

Der Pact Broker ist eine Open-Source-Anwendung, die Pacts verwaltet[7], wodurch sie versioniert und zwischen Services ausgetauscht werden können. Zudem können auch Verifikationsergebnisse von Pacts eingetragen und abgerufen werden, aus denen hervor geht, ob die Verifikation erfolgreich war und welche Service-Versionen miteinander getestet wurden.

Der Pact Broker stellt dazu eine entsprechende Representational State Transfer (REST) Schnittstelle bereit, die auch durch das Pact Framework verwendet werden kann. Außerdem wird die Einrichtung von Webhooks[8] unterstützt, die ausgelöst werden können, sobald ein Pact eingebracht bzw. verändert wird oder ein Verifikationsergebnis eingetragen wird (Pact Foundation, 2021).

##### Verwendung für Continuous Integration

Mithilfe des Pact Brokers kann Pact in die jeweiligen CI bzw. CD Pipelines der zu testenden Microservices integriert werden. Das nun folgende Beispiel illustriert, wie dies erfolgen könnte.

Nach der Durchführung eines consumerseitigen Tests wird der entstandene Pact in den Pact Broker eingebracht. Dadurch wird eine Webhook ausgelöst, welche, sofern der Pact neu ist oder sich dieser verändert hat, die CI Pipeline des Providers auslöst. Der anschließende providerseitige Test bezieht den Pact aus dem Pact Broker und führt anhand dessen die Verifikation durch. Die Verifikationsergebnisse werden nach Abschluss des providerseitigen Tests an den Pact Broker übermittelt. Die CI Pipeline des Consumers wird durch eine Webhook benachrichtigt, dass nun neue Verifikationsergebnisse vorliegen. Sofern die Verifikation

[7] Darüber hinaus werden auch beliebige Inhalte im JSON Format unterstützt.

[8] Dabei handelt es sich um konfigurierbare HTTP-Anfragen, deren Ausführung durch bestimmte Ereignisse ausgelöst wird (Pact Foundation, 2021).

erfolgreich war, kann schließlich das Deployment des Consumers fortgesetzt werden (Pact Foundation, 2021).

### 3.2.2 Docker

Docker ist ein Open-Source-Werkzeug zur Isolierung von Softwareanwendungen durch containerbasierte Virtualisierung. Dazu können Anwendungen in einer isolierten Umgebung erstellt und ausgeführt werden, in welcher beispielsweise Bibliotheken, Laufzeitumgebungen und anderweitige Werkzeuge bereitgestellt werden können (Docker, Inc., 2021).

In den nachfolgenden Sektionen werden Konzepte von Docker erläutert und eine Einführung in das Werkzeug Docker Compose gegeben.

#### 3.2.2.1 Konzepte

Diese Sektion gibt eine Übersicht über grundlegende Konzepte von Docker. In den anschließenden Sektionen wird auf Möglichkeiten zur persistenten Speicherung von Daten eingegangen und es werden sog. Multi-Stage Builds erläutert.

**Image:** Schreibgeschützte Vorlage, aus der lauffähige Container erzeugt werden können. Oftmals basieren Images auf anderen Images, jedoch mit zusätzlichen, benutzerdefinierten Anpassungen. Images können mithilfe eines Dockerfiles selbst erzeugt werden, allerdings können auch bereits erstellte Images verwendet werden, die beispielsweise in einer Registry, wie Docker Hub[9], verwaltet werden.

**Dockerfile:** Textdatei, welche alle Instruktionen enthält, die für die Erzeugung eines Images benötigt werden. Beispiele für derartige Instruktionen sind die Festlegung eines Basis-Images, das Kopieren von Dateien oder die Ausführung von Kommandozeilenbefehlen. Zur Erzeugung eines Images werden die Instruktionen des entsprechenden Dockerfiles sequenziell durch Docker ausgeführt.

**Layer:** Schreibgeschützte Bestandteile von Images. Jede Instruktion in Dockerfiles, welche Dateien hinzufügt, ändert oder löscht, erzeugt für das jeweilige Image ein neues Layer, das diese Dateiänderungen verkörpert. Ein Image wird demnach aus einer Serie von Layern aufgebaut.

**Container:** Lauffähige Instanz eines Images. Container können durch Netzwerke miteinander verbunden werden und es können persistente Speicher in Container eingebunden werden. Der Grad der Isolation eines Containers zu anderen Containern oder dem Hostsystem kann benutzerdefiniert festgelegt werden.

[9] https://hub.docker.com/

**Container Layer:** Spezieller, schreibbarer Layer, welcher an den Lebenszyklus eines Containers gebunden ist. Der Container Layer wird bei der Erstellung eines Containers erzeugt. Dateiänderungen, die durch einen laufenden Container durchgeführt werden, werden standardmäßig in seinem Container Layer vorgenommen. Die Löschung eines Containers hat auch die Löschung seines Container Layers zur Folge.

Die Informationen zu den soeben vorgestellten Konzepten stammen aus der Docker Dokumentation (Docker, Inc., 2021).

Abseits von Container Layern existieren andere Möglichkeiten, um Daten, die während der Ausführung von Containern erzeugt werden, persistent und über den Lebenszyklus des Containers hinweg zu speichern. Diese werden in der folgenden Sektion vorgestellt.

### Persistente Speicherung von Daten

Um Daten, die im Rahmen der Ausführung von Containern anfallen, persistent und unabhängig vom Container zu speichern, können sog. Volumes eingesetzt werden. Diese werden durch Docker verwaltet und können in die Dateisysteme von Containern eingebunden werden. Dasselbe Volume kann außerdem von mehreren Containern verwendet werden (Docker, Inc., 2021).

Als Alternative zu Volumes stehen sog. Bind Mounts zur Verfügung. Durch diese kann eine Datei oder ein Verzeichnis des Hostsystems in das Dateisystem des Containers eingebunden werden. Im Gegensatz zu Volumes werden Bind Mounts nicht durch Docker verwaltet (Docker, Inc., 2021).

### Multi-Stage Builds

Bei Multi-Stage Builds handelt es sich um einen Ansatz zur Reduzierung der Größe von Images. Dazu werden verschiedene Stages innerhalb von Dockerfiles definiert, die durch Instruktionen eingeleitet werden, welche ein neues Basis-Image festlegen. Layer, die vor der Festlegung eines neuen Basis-Images erzeugt wurden, werden nicht in das finale Image eingebracht. Aus diesen Layern können jedoch selektiv Daten übernommen werden, die innerhalb der neu angefangenen Stage benötigt werden. Anstatt eines Basis-Images kann jedoch auch der Name einer vorherigen Stage angegeben werden. Auf diese Weise kann die vorherige Stage fortgeführt werden, während Daten aus der vorherigen Stage erhalten bleiben (Docker, Inc., 2021).

Darüber hinaus muss bei der Erzeugung eines Images nicht zwingend jede Stage durchlaufen werden. Dazu kann eine beliebige Stage ausgewählt werden, bis zu der die Instruktionen des Dockerfiles ausgeführt werden (Docker, Inc., 2021).

#### 3.2.2.2 Docker Compose

Bei Docker Compose handelt es sich um ein Open-Source-Werkzeug, welches die Definition und Ausführung von Docker-Anwendungen ermöglicht, welche aus mehreren Containern bestehen (in diesem Zusammenhang Services genannt). Dazu werden Compose Dateien im YAML-Format eingesetzt, durch welche die Konfiguration von Services festgelegt werden kann. Ein Service kann dabei entweder auf Grundlage eines Dockerfiles oder anhand eines bereits existierenden Images erstellt werden. Durch Kommandozeilenbefehle können schließlich alle Services einer Anwendung erzeugt, gestartet und gestoppt werden (Docker, Inc., 2021).

Es können auch mehrere Compose Dateien pro Anwendung eingesetzt werden. Dabei dient eine Compose Datei als Grundlage, während deren Inhalt durch weitere Compose Dateien selektiv überschreiben werden kann. Dies kann dazu eingesetzt werden, um eine Anwendung für unterschiedliche Umgebungen, wie beispielsweise die Produktivumgebung oder die CI-Umgebung, zu konfigurieren (Docker, Inc., 2021).

### 3.2.3 GitHub Actions

GitHub Actions ist eine Plattform für CI und CD, die Repositories auf GitHub[10] zur Verfügung steht. Diese ermöglicht eine Automatisierung des Erstellungsprozesses (engl. *build*), der Testung und des Deployments von Software, indem derartige Aufgaben durch sog. Workflows umgesetzt und ereignisgesteuert ausgeführt werden (GitHub, Inc., 2021).

Es folgt eine Vorstellung der Konzepte von GitHub Actions, auf deren Grundlage CI/CD Pipelines realisiert werden können. Hilfestellungen für die praktische Umsetzung solcher Pipelines sind der Dokumentation (GitHub, Inc., 2021) zu entnehmen.

**Workflow:** Konfigurierbarer, automatisierter Prozess, der einen oder mehrere Jobs ausführt. Workflows werden durch YAML-Dateien definiert, die sich innerhalb des Repositories befinden. Die Ausführung von Workflows kann durch Events, zeitgesteuert oder manuell über das UI von GitHub ausgelöst werden.

**Event:** Spezifisches Ereignis, welches die Ausführung eines Workflows auslösen kann. Beispiele für derartige Ereignisse sind Pull Requests oder Commits, die in das Repository eingebracht werden.

**Job:** Menge von Steps, die innerhalb eines Workflows auf demselben Runner ausgeführt werden. Standardmäßig werden Jobs parallel zueinander ausgeführt. Es können jedoch auch Abhängigkeiten zwischen Jobs definiert

[10] https://github.com/

werden, sodass die Ausführung eines Jobs erst nach dem Abschluss anderer Jobs erfolgt.

**Step:** Auszuführender Kommandozeilenbefehl oder auszuführende Action. Alle Steps eines Jobs werden grundsätzlich sequenziell auf demselben Runner ausgeführt. Dateien, die in einem Step angelegt werden, bleiben daher für die nachfolgenden Steps erhalten.

**Action:** Benutzerdefinierte Anwendung für die GitHub Actions Plattform, die eine komplexe, wiederkehrende Aufgabe bewältigt. Beispielsweise kann damit das Repository heruntergeladen werden, Werkzeuge für den Softwareerstellungsprozess eingerichtet werden oder Artifacts hoch- und heruntergeladen werden. Actions können entweder aus dem GitHub Marketplace[11] bezogen oder eigenständig entwickelt werden.

**Runner:** Server, der einzelne Jobs in einer neu erzeugten virtuellen Maschine ausführt. Durch GitHub werden Runner mit unterschiedlichen Betriebssystemen zur Verfügung gestellt, jedoch können Runner auch selbst gehostet werden.

**Artifact:** Konzept zur Speicherung von Dateien, um diese anderen Jobs desselben Workflows zugänglich zu machen. Standardmäßig bleiben Artifacts nur eine begrenzte Anzahl an Tagen erhalten, bis sie gelöscht werden. Für Entwickler stehen zwei Actions zur Verfügung, mit denen Artifacts während eines Jobs hoch- und heruntergeladen werden können.

Die soeben vorgestellten Konzepte wurden allesamt aus der GitHub Actions Dokumentation (GitHub, Inc., 2021) entnommen.

[11] https://github.com/marketplace

## 3.3 Entwicklung der Consumer-Driven Contract Tests

In dieser Sektion werden ergänzende Inhalte zur Entwicklung der CDC Tests thematisiert, die im Rahmen der Aktionsforschung durchgeführt wurde. Die Inhalte umfassen die Umsetzung des Testaufbaus, wodurch die lokale Ausführung der CDC Tests ermöglicht wird, und die Einbindung der CDC Tests in die bestehende CI Pipeline des ODS.

In Sektion B des Anhangs befinden sich darüber hinaus die 4 ergänzenden Abbildungen A2 bis A5, welche die Testumfänge der entwickelten CDC Tests je Service-Integration darstellen.

### 3.3.1 Testaufbau

Der Testaufbau der CDC Tests umfasst im Wesentlichen jene Aspekte, die für die lokale Ausführung der Tests von Bedeutung sind. Diese umfassen die Organisation von Test- und Contract-Dateien, die separate Ausführung der consumer- und providerseitigen Tests mithilfe eines Test-Frameworks und dessen Isolierung mittels Docker bzw. Docker Compose.
Die nachfolgenden beiden Sektionen zeigen auf, wie dies im Rahmen der Aktionsforschung umgesetzt wurde.

#### 3.3.1.1 Organisation der Testdateien

Testdateien sind stets auf oberster Ebene im `src`-Verzeichnis des jeweiligen Microservice angesiedelt. Dies erleichtert das Auffinden der Testdateien, da sie sich somit serviceübergreifend an selbiger Stelle befinden.
In allen Microservices, für welche CDCT umgesetzt wurde, kommt das Open-Source Test-Framework Jest[12] zum Einsatz, womit auch bereits andere Tests der Microservices realisiert wurden. Für die Realisierung der CDC Tests wurde außerdem die Open-Source Bibliothek Jest-Pact[13] verwendet, welche den Einsatz von Pact unter Verwendung von Jest erleichtert. Gemäß der Empfehlung von Jest-Pact wurden zu Testdateien zugehörige `fixtures`-Dateien angelegt, in welche die konkreten Anfragen und Antworten der getesteten Interaktionen ausgelagert werden. Dies verbessert die Übersichtlichkeit des Quellcodes und erlaubt die Wiederverwendung von Anfragen und Antworten in unterschiedlichen Service-Interaktionen.

Um bei der Testausführung zwischen consumer- bzw. providerseitigen Tests und Unit Tests zu unterscheiden, wurde sich dazu entschieden, die Testdateien unter-

[12] https://jestjs.io/
[13] https://github.com/pact-foundation/jest-pact

schiedlich zu benennen. Folgende Konventionen wurden für die Dateibenennung eingesetzt:

- `*.consumer.pact.test.ts` für consumerseitige Tests, wobei `*` den Namen des zugehörigen Providers symbolisiert.
- `*.provider.pact.test.ts` für providerseitige Tests, wobei `*` den Namen des zugehörigen Consumers[14] symbolisiert.

Alternativ hätten beispielsweise auch unterschiedliche Verzeichnisse für consumer- und providerseitige Tests angelegt werden können.

Alle durch Pact generierten Pact- und Log-Dateien werden in einem gemeinsamen, serviceübergreifenden Verzeichnis namens `pacts` gespeichert, wodurch die Pact-Dateien im Rahmen der providerseitigen Tests leicht aufgefunden werden können.

#### 3.3.1.2 Ausführung mittels Docker

Um die Ausführung der CDC Tests in Docker zu ermöglichen, wurde der bestehende Multi-Stage Build Ansatz angepasst. Abbildung 3.8 zeigt auf, welche Änderungen vorgenommen wurden.

Aus der Abbildung wird ersichtlich, dass die zuvor existierende `build`-Stage in zwei Stages namens `base` und `build` unterteilt wurde. Die Unterteilung wurde derart getroffen, dass die CDC Tests bereits nach Abschluss der `base`-Stage lauffähig sind, ohne dass die nachfolgende `build`-Stage durchlaufen werden muss.

Für die Ausführung mit Docker Compose wurden zwei Compose Dateien angelegt, wobei eine der consumerseitigen und eine der providerseitigen Testausführung dient. Diese werden dazu eingesetzt, die Konfiguration von Services aus der regulären, bereits bestehenden Compose Datei derart zu überschreiben, dass die `base`-Stage als Ziel-Stage festgelegt wird und beim Start der Container der geeignete Befehl zur Durchführung der CDC Tests ausgeführt wird. Außerdem wird das `pacts`-Verzeichnis durch ein Bind Volume in den Container eingebunden, sodass die Pact-Dateien nach Abschluss der CDC Tests im Dateisystem des Hosts vorliegen.
Durch den gewählten Ansatz können die bereits vorhandenen Dockerfiles und die vorhandene Compose Datei wiederverwendet werden, während durch die neu eingeführte `base`-Stage eine vergleichsweise schnelle Image-Erzeugung ermöglicht wird. Ein Nachteil besteht jedoch darin, dass nun unterschiedliche Images für das CDCT und die Produktivsetzung eines Microservice nötig sind.

[14] Sofern mehrere Consumer involviert sind, wird für * ein adäquater, allgemeinerer Begri verwendet.

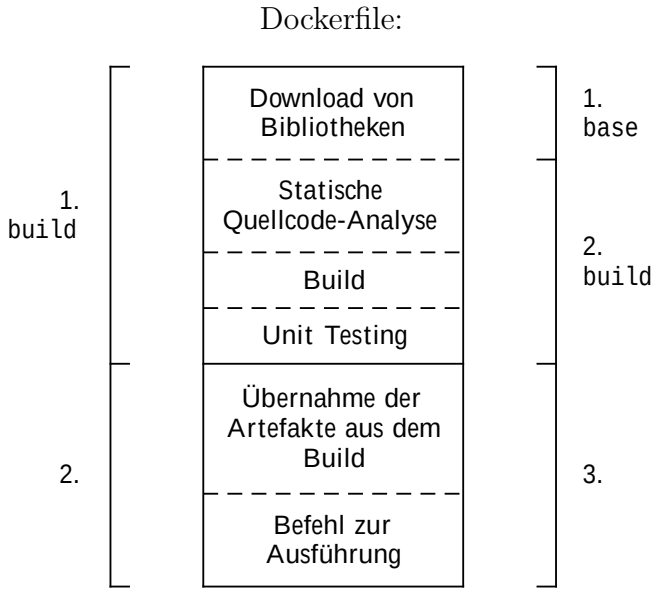


Abbildung 3.8: Stage-Unterteilungen der Dockerfiles von Services vor und nach der Umsetzung der CDC Tests. Die mittlere Spalte listet die Instruktionen des Dockerfiles in vereinfachter Form auf. Die durchgezogene, horizontale Linie innerhalb des Dockerfiles symbolisiert, dass die dort beginnende Stage ein neues Basis-Image verwendet. Auf der linken Seite ist die Stage-Unterteilung vor der Umsetzung der CDC Tests zu sehen, während rechts die Stage-Unterteilung nach deren Umsetzung dargestellt wird.

Mithilfe von Skripten können die CDC Tests noch leichter zur Ausführung gebracht werden. Diese starten neben dem zu testenden Service auch dessen abhängige Systeme, wie Datenbanksysteme, Outboxer-Services oder den Message Broker, sofern diese für die Testdurchführung benötigt werden.

### 3.3.2 Einbindung in die bestehende CI Pipeline

Im Rahmen der Aktionsforschung wurde CDCT in die bestehende CI Pipeline des ODS integriert. Diese ist mit GitHub Actions realisiert und verfügt über einen Workflow, in welchem mittels Docker bzw. Docker Compose die Unit Tests sowie Integrationstests aller Services und im Anschluss die auch die E2E Tests ausgeführt werden. In diesen Workflow wurden schließlich die CDC Tests eingebunden.

#### 3.3.2.1 Ablauf der CI Pipeline

Zu Beginn des Workflows werden alle Microservices isoliert voneinander in parallel ablaufenden Jobs getestet. Hierzu ist für jeden Service ein Job vorhanden, in welchem zunächst das Docker Image des Services erzeugt wird. Hierbei erfolgt bereits die Ausführung der Unit Tests. Im Anschluss wird der Integrationstest unter Verwendung des zuvor erstellten Images durchgeführt. Danach wird das Image als Artifact hochgeladen, sodass es im späteren E2E Test wiederverwendet werden kann. Zuletzt erfolgen die consumerseitigen CDC Tests. Für diese wird jeweils das zuvor erstellte Image des Services überschrieben, da stattdessen ein Image verwendet wird, das ausschließlich die neu eingeführte `base`-Stage umfasst. Die erzeugten Pact-Dateien werden schließlich als Artifact hochgeladen, um diese den nachfolgenden providerseitigen Tests zur Verfügung zu stellen.
Im Rahmen der Aktionsforschung wurden auch andere Ansätze zur Übermittlung der Contracts in Betracht gezogen. Die nachfolgende Sektion 3.3.2.2 ist speziell diesem Thema gewidmet.

Im Anschluss erfolgt die providerseitige Testung. Hierzu wurden parallel ablaufende Jobs pro Provider-Service eingerichtet. Als Abhängigkeiten dieser Jobs werden die vorhergehenden Jobs derjenigen Services deklariert, welche eine Pact-Datei für diesen Provider erzeugen. Zur providerseitigen Testung wird zunächst das Artifact, welches die Pact-Dateien beinhaltet, in das `pacts`-Verzeichnis heruntergeladen. Im Anschluss werden schließlich die providerseitigen Tests durchgeführt. Darüber hinaus verläuft der E2E Test hierzu parallel in einem eigenen, separaten Job.

#### 3.3.2.2 Ansätze zur Übermittlung von Contracts

Für die Integration der CDC Tests in die CI Pipeline musste eine Entscheidung darüber getroffen werden, wie die Pact-Dateien, welche durch die consumerseitigen Tests erzeugt werden, den providerseitigen Tests zur Testdurchführung zur Verfügung gestellt werden. Insgesamt wurde zwischen 3 Ansätzen abgewägt, welche in den folgenden Sektionen beschrieben werden.

##### Inklusion von Contracts in das Repository

Eine Möglichkeit besteht darin, Contract-Dateien mit in das Repository einzubringen. Dies wurde als erster Ansatz für das ODS-System umgesetzt. Um zu verhindern, dass veraltete oder inhaltlich falsche Contracts eingebracht werden, wurden im Rahmen der CI, unabhängig von den zuvor eingebrachten Contracts, stets aktuelle Contracts durch die consumerseitigen Tests erzeugt. Im Anschluss wurden diese mithilfe einer Action durch einen automatischen Commit in das Repository eingebracht. Dieser zusätzliche Commit konnte jedoch entfallen, sofern die zuvor eingebrachten und die neu erzeugten Contract-Dateien syntaktisch

äquivalent waren.

Ein Vorteil dieses Ansatzes ist, dass die Contract-Dateien durch die Versionskontrolle erfasst sind und somit die Evolution der Contracts nachverfolgt werden kann. Als Nachteil kann jedoch die Notwendigkeit zu automatischen Commits erachtet werden. Außerdem befindet sich das Repository vor der Durchführung eines automatischen Commits möglicherweise kurzzeitig in einem Zustand, in welchem es ungültige Contracts beinhaltet. Darüber hinaus kann es vorkommen, dass Contract-Dateien sich in ihrer Syntax unterscheiden, z.B. wenn Interaktionen unterschiedlich angeordnet sind, obwohl diese semantisch äquivalent sind. Auch in diesem Fall würde ein automatischer Commit durchgeführt werden, obwohl keine Notwendigkeit dazu bestehen würde. Um dies zu verhindern, sollte bei der Contract Erzeugung daher sichergestellt werden, dass die Interaktionen immer in derselben Reihenfolge angeordnet sind.

#### Bereitstellung von Contracts als Artifact

Ein anderer Ansatz besteht darin, die Contract-Dateien im Rahmen von GitHub Actions als Artifact hochzuladen, sodass sie für die providerseitigen Tests heruntergeladen werden können. Dieser Ansatz wurde letztendlich für das ODS-System gewählt und umgesetzt. Hierzu wurde gänzlich verhindert, dass Contract-Dateien im Rahmen von Commits in das Repository eingebracht werden.

Durch diesen Ansatz kann es vermieden werden, dass ungültige Contracts in das Repository eingebracht werden. Aufgrund dessen sind ebenfalls keine nachträglichen, automatischen Commits nötig und auch Unterschiede in der Syntax der Contracts haben keine weiteren Auswirkungen. Nachteile dieser Herangehensweise sind jedoch, dass Contracts keiner Versionskontrolle unterliegen und dass die Artifacts, welche die Contract-Dateien beinhalten, nach gewisser Zeit automatisch durch GitHub gelöscht werden.

#### Verwaltung von Contracts durch den Pact Broker

Durch den Pact Broker können Contracts verwaltet und zugänglich gemacht werden. Für das ODS-System wurde sich jedoch gegen dessen Einsatz entschieden, da von den Vorteilen des Pact Brokers nicht ausreichend profitiert werden kann.

Der Pact Broker ermöglicht beispielsweise die Eintragung und den Abruf von providerseitigen Verifikationsergebnissen. Dies kann jedoch ebenfalls anhand des Erfolgs oder Fehlschlags jener Jobs ermittelt werden kann, welche die providerseitige Testung durchführen.
Des Weiteren können durch den Pact Broker die Versionen der getesteten Services erfasst werden. Auch dieser Vorteil spielt für das ODS-System keine Rolle, da die einzelnen Services nicht versioniert werden.

Darüber hinaus ermöglicht der Pact Broker die Einrichtung von Webhooks, welche die gezielte Ausführung von CI Pipelines einzelner Microservices ermöglichen. Auch diese Funktionalität wird nicht benötigt, da die Services des ODS-Systems über keine separaten CI Pipelines verfügen, die zur Durchführung der CDC Tests gezielt ausgeführt werden müssten.
Außerdem kommt noch der Nachteil des Pact Brokers hinzu, dass dieser eigenständig oder durch einen Anbieter gehostet werden muss.

Aus den Überlegungen ging schließlich hervor, dass der Einsatz des Pact Brokers insbesondere dann als sinnvoll erachtet wird, sobald die Services des ODS in separaten Repositories verwaltet werden und separat voneinander versioniert werden.

## 3.4 Defect Seeding

Die Methodik des Defect Seedings wurde weitestgehend von Lehvä et al. (2019) übernommen. Aus deren Arbeit wird jedoch nicht ersichtlich, welche einzelnen Integrationsfehler in das Microservice-System eingebracht wurden, da nur vereinzelte Beispiele für Fehler[15] genannt werden.

In der Literatur konnte darüber hinaus keine Klassifizierung bzw. Auflistung von Integrationsfehlern aufgefunden werden, die für das Defect Seeding hätte Verwendung finden können. Aus diesem Grund wurde sich dazu entschieden, die einzubringenden Integrationsfehler eigenständig herzuleiten. Das Vorgehen hierzu ist der nachfolgenden Sektion 3.4.1 zu entnehmen, während die darauf folgende Sektion 3.4.2 die Durchführung des Defect Seedings beschreibt.

### 3.4.1 Herleitung der einzubringenden Integrationsfehler

Für die Herleitung der einzubringenden Integrationsfehler wurden zuerst diverse Integrationsänderungen aufgestellt. Diese wurden zunächst systematisch, anhand von Kriterien für den Parameter bzw. das Attribut, an welchem die Änderung vorzunehmen ist, und der Operation, durch welche die Änderung hervorgerufen wird, charakterisiert. Im Detail umfassen die Kriterien, ob die Änderung consumerseitig oder providerseitig vorgenommen wird, ob die Änderung auf eine HTTP-Anfrage, HTTP-Antwort oder eine asynchrone Nachricht bezogen wird, ob die Änderung an einem Query-Parameter, Pfad-Parameter oder JSON-Attribut der Nutzdaten vorgenommen wird und ob der Parameter bzw. das Attribut optional oder obligatorisch ist. Die Operationen, durch welche die Änderungen hervorgerufen werden, umfassen die Entfernung, die Umbenennung, die Hinzufügung, die Wertebereichsvergrößerung und die Wertebereichsverkleinerung von Parametern bzw. Attributen. Das kartesische Produkt all dieser Kriterien und Operationen umfasst jedoch auch Änderungen, die nicht durchführbar sind. Ein Beispiel hierfür wären Änderungen an Query- bzw. Pfad-Parametern von asynchronen Nachrichten, da solche Nachrichten über keine Query- und Pfad-Parameter verfügen. Derartige unmögliche Änderungen wurden daher identifiziert und aus der fortlaufenden Betrachtung ausgeschlossen.

Zusätzlich zu den systematisch hergeleiteten Integrationsänderungen wurden zusätzliche Änderungen festgelegt, die weitere Aspekte bezüglich HTTP berücksichtigen, welche andernfalls nicht berücksichtigt worden wären. Diese umfassen die Änderung eines erwarteten bzw. versendeten Status Codes, die Weglas-

[15] Von Lehvä et al. (2019) werden folgende Beispiele für die eingebrachten Integrationsfehler genannt: Consumerseitig etwa die Umbenennung von Query-Parametern, Formatänderungen des Anfrage-Bodies und Modifizierungen der Anfrage-Header; providerseitig hingegen Änderungen an der API und dessen Antworten.

sung, Umbenennung und Hinzufügung von consumer- und providerseitig versendeten Headern, die gänzliche Entfernung einer REST-Schnittstellenoperation, die consumer- und providerseitige Änderung der statischen URL einer Schnittstellenoperation und die Änderung der consumerseitig verwendeten bzw. providerseitig erwarteten HTTP-Methode einer Schnittstellenoperation.

Im Anschluss wurde für jede dieser Integrationsänderungen beurteilt, ob die Änderung eine Inkompatibilität zwischen einem Consumer und einem Provider hervorrufen würde. Für manche Änderungen wurden Voraussetzungen identifiziert, die erfüllt sein müssen, damit die Änderung eine Inkompatibilität verursacht.

### 3.4.2 Durchführung

Für die Durchführung des Defect Seedings wurde im Repository zunächst ein neuer Branch angelegt. In diesem wurde die CI Pipeline derart abgeändert, dass ausschließlich die CDC Tests und die E2E Tests jeweils unabhängig voneinander ausgeführt werden. Die Unit Tests und Integrationstests wurden hierbei von vornherein aus der Betrachtung ausgeschlossen, da sie keine Service-Integrationen testen und somit über ein mangelndes Aufdeckungspotenzial für Integrationsfehler zwischen Services verfügen.

Für jeden einzubringenden Integrationsfehler wurde ein separater Branch angelegt, in welchem der Fehler an einer willkürlichen, geeigneten Stelle[16] eingebracht wurde. Die Fehler wurden jedoch ausschließlich in Integrationen eingebracht, für welche zuvor CDC Tests umgesetzt wurden. Für manche Integrationsfehler gab es außerdem keine Möglichkeit, diese einzubringen. Beispielsweise konnten keine Integrationsfehler für optionale Query-Parameter eingebracht werden, da in getesteten Integrationen keine optionalen Query-Parameter aufgefunden werden konnten.
Nach der Einbringung der Integrationsfehler gab die modifizierte CI Pipeline Aufschluss darüber, welche Fehler durch welche Testverfahren aufgedeckt werden konnten. Sofern ein eingebrachter Fehler durch die CDC Tests nicht aufgedeckt werden konnte, wurde geprüft, ob der Fehler durch die Ergänzung einer Interaktion aufgedeckt werden könnte. Sofern eine derartige Interaktion im Rahmen der regulären Entwicklung der CDC Tests identifiziert und abgedeckt worden wäre, wurde diese Interaktion nachträglich in den CDC Tests ergänzt und die Ergänzung in den Ergebnissen vom Defect Seeding vermerkt. Eine Übersicht über die eingebrachten Integrationsfehler und die zugehörigen Testergebnisse ist Sektion D des Anhangs zu entnehmen.

[16] An dieser Stelle unterscheidet sich die Vorgehensweise zu der von Lehvä et al. (2019), da die Fehler in dieser Arbeit nicht pro getesteter Interaktion sondern willkürlich eingebracht werden. Dies ermöglicht es, dass eingebrachte Integrationsfehler potenziell von mehreren Interaktionen aufgedeckt werden und dass Integrationsfehler aufgrund mangelnder Interaktionen möglicherweise nicht aufgedeckt werden.

# Anhang

# A Microservices des JValue Open Data Service

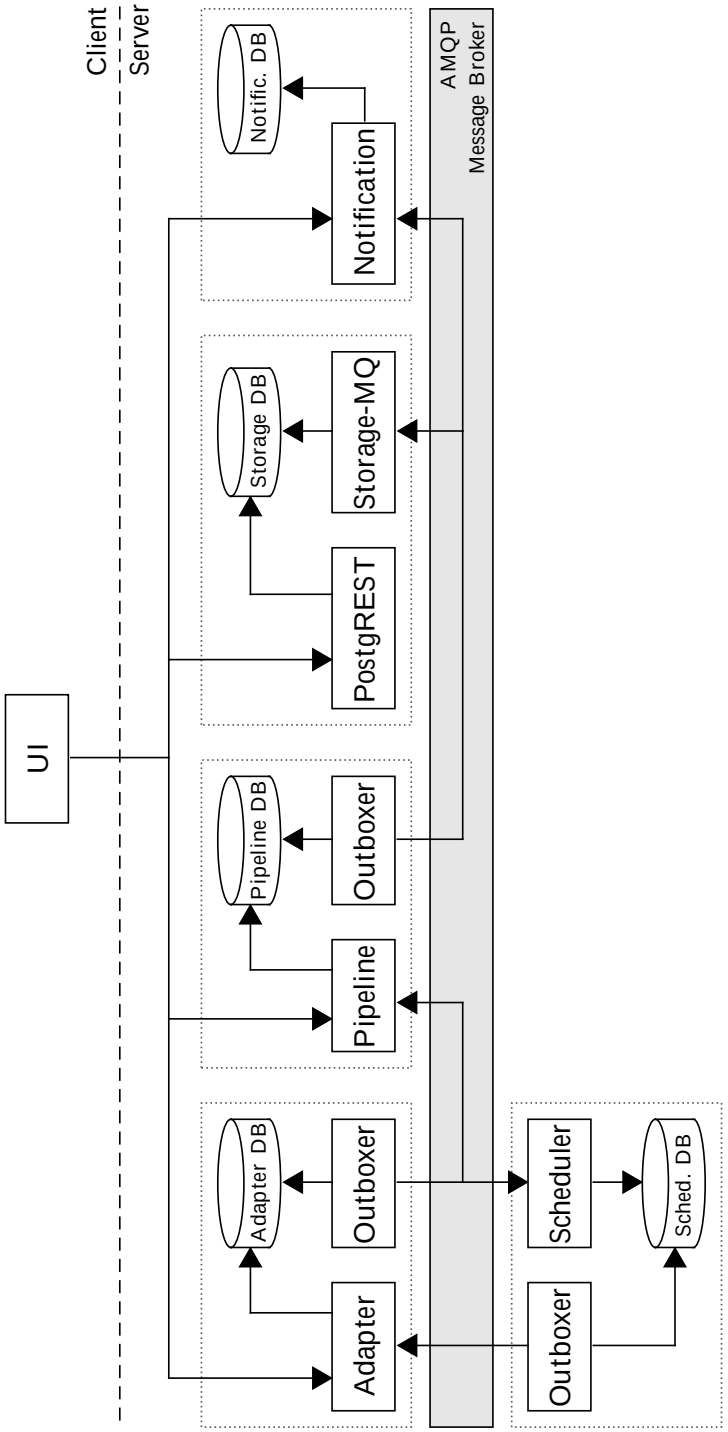


Abbildung A1: Überblick über die Microservices des ODS, deren abhängige Systeme und deren Kommunikationsbeziehungen. Zylinder, welche die Aufschrift „DB“ beinhalten, symbolisieren Datenbanksysteme.

# B Testumfänge der entwickelten Consumer-Driven Contract Tests

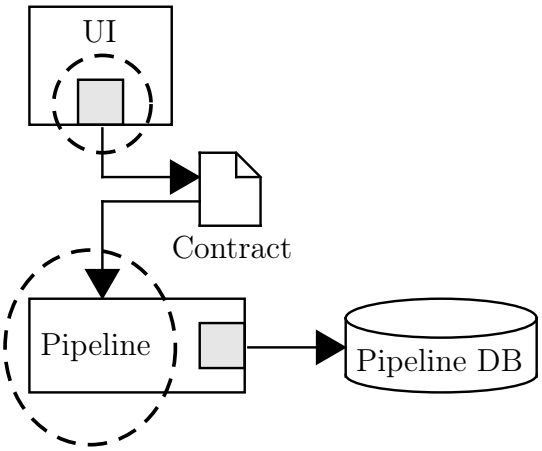


Abbildung A2: Testumfang der entwickelten CDC Tests für die HTTP-Integration zwischen UI und Pipeline Service.
Consumerseitig umfassen die CDC Tests ausschließlich jenes Modul des UI, welches für die Kommunikation mit dem Pipeline Service verantwortlich ist. Providerseitig ist der Pipeline Service umfasst, jedoch wurde das Modul, welches für die Kommunikation zum Datenbanksystem verantwortlich ist, durch einen Mock ersetzt. Die Exklusion des Datenbanksystems verringert die Laufzeit der CDC Tests, da das Datenbanksystem zur Testdurchführung nicht hochgefahren werden muss, und erleichtert die Herstellung von Ausgangszuständen.

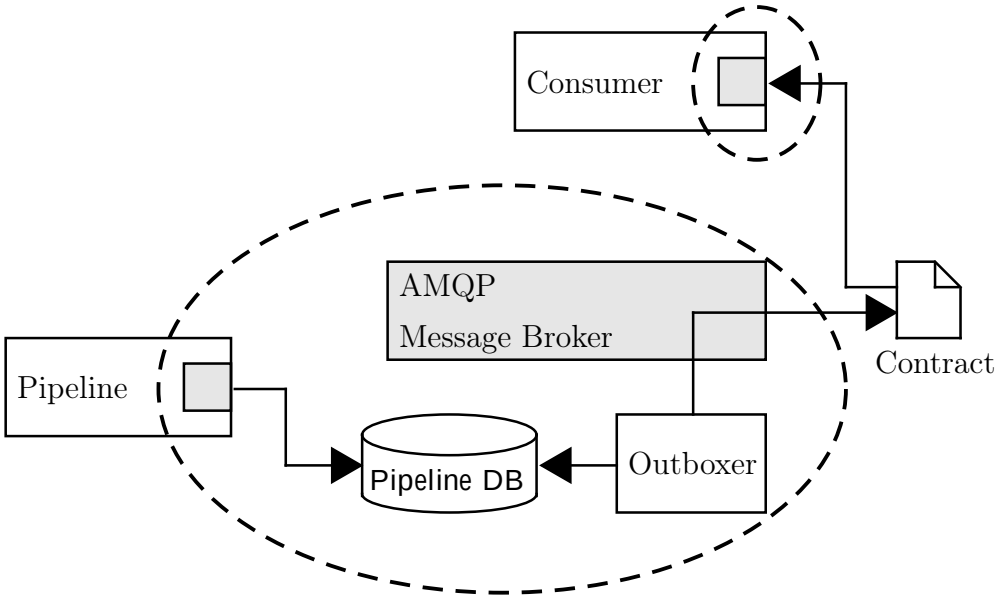


Abbildung A3: Testumfang der entwickelten CDC Tests für die nachrichtenbasierten Integrationen. Die Bezeichnung „Consumer“ wird hierbei stellvertretend für den Storage-MQ Service und den Notification Service verwendet.
Consumerseitig werden dem Consumer die in den Interaktionen hinterlegten Nachrichten zugespielt. Dabei ist ausschließlich das Modul des Consumers beteiligt, welches für die Nachrichtenverarbeitung zuständig ist. Providerseitig ist bezüglich des Pipeline Services nur jenes Modul beinhaltet, welches für die Kommunikation zum Datenbanksystem verantwortlich ist. Ansonsten sind ebenfalls das Datenbanksystem, der Outboxer-Service und der Message Broker involviert, sodass der gesamte Prozess der Nachrichtenerzeugung und -versendung umfasst wird.

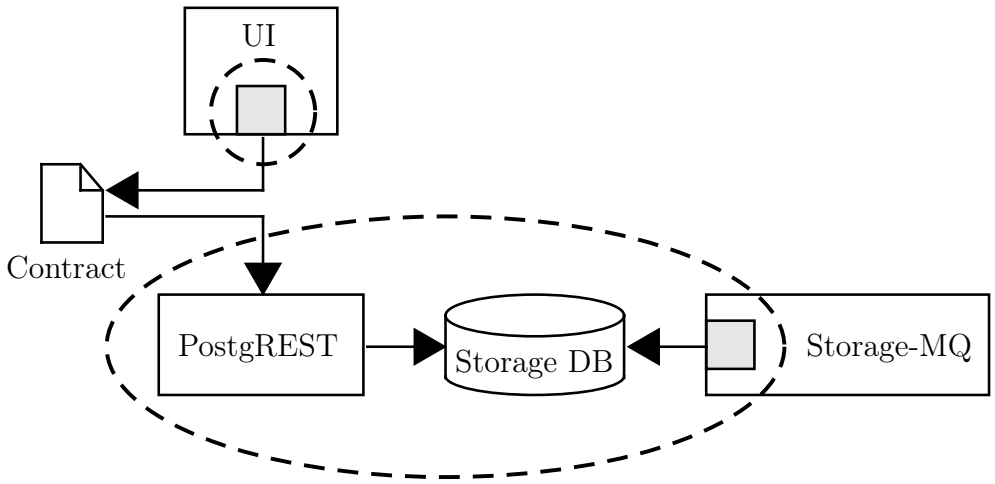


Abbildung A4: Testumfang der entwickelten CDC Tests für die HTTP-Integration zwischen UI und Storage Service.
Consumerseitig umfassen die CDC Tests ausschließlich jenes Modul des UI, welches für die Kommunikation mit dem PostgREST Service verantwortlich ist. Providerseitig werden sowohl der PostgREST Service als auch die zugehörige Datenbank umfasst, da die durch den PostgREST bereitgestellte REST-API auf Grundlage des Datenbankschemas vom zugehörigen Datenbanksystem bereitgestellt wird. Durch das Modul des Storage-MQ Services, welches für die Kommunikation mit dem Datenbanksystem verantwortlich ist, werden die Ausgangszustände der Datenbank hergestellt.

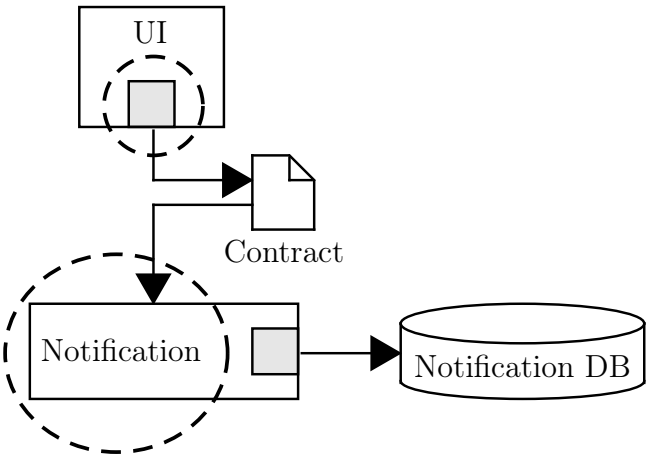


Abbildung A5: Testumfang der entwickelten CDC Tests für die HTTP-Integration zwischen UI und Notification Service. Die Tests für diese Integration wurde durch die 3 interviewten Entwickler des ODS realisiert. Wesentliche Unterschiede bezüglich des Testumfangs gab es nicht.
Consumerseitig umfassen die CDC Tests ausschließlich jenes Modul des UI, welches für die Kommunikation mit dem Notification Service verantwortlich ist. Providerseitig ist der Notification Service umfasst, jedoch wurde das Modul, welches für die Kommunikation zum Datenbanksystem verantwortlich ist, durch einen Mock ersetzt.

# C Leitfaden zur Interviewführung

Zwecks der Übersichtlichkeit wurde der Leitfaden in insgesamt 4 Teile aufgeteilt, welche in den nachfolgenden Abbildungen A6 bis A9 dargestellt werden.

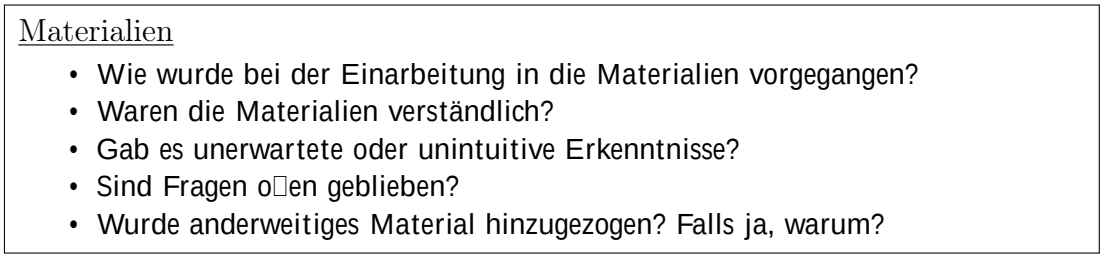


Abbildung A6: Auszug des Interview-Leitfadens, in welchem sich auf die Einarbeitung in die zur Verfügung gestellten Materialien bezogen wird.

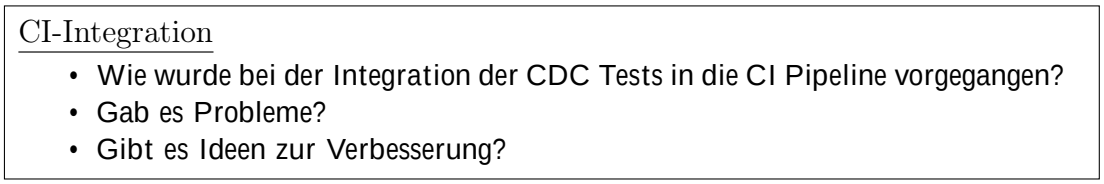


Abbildung A7: Auszug des Interview-Leitfadens, in welchem sich auf die Integration der CDC Tests in die CI Pipeline bezogen wird.

Implementierung der CDC Tests

- Wie wurde bei der Implementierung der CDC Tests vorgegangen?
- Identifikation der zu testenden Service-Interaktionen:
  - Welche Interaktionen wurden identifiziert?
  - Wie wurde bei der Identifikation vorgegangen?
  - Wie wurde mit den unterschiedlichen Notification-Arten umgegangen?
  - Wurde der Wert „Not a Number“ für IDs in Interaktionen berücksichtigt?
- Umsetzung der consumerseitigen Tests:
  - Welche Anpassungen waren im Quellcode vom UI erforderlich?
  - Gab es Probleme oder Unsicherheiten bei der Umsetzung? Falls ja, wie wurde damit umgegangen?
  - Wird die Aufteilung zwischen Test- und Fixtures-Dateien als sinnvoll erachtet?
- Umsetzung der providerseitigen Tests:
  - Welche Anpassungen waren im Quellcode vom Notification Service erforderlich?
  - Wie wurde das Mocking umgesetzt?
  - Gab es Probleme oder Unsicherheiten bei der Umsetzung? Falls ja, wie wurde damit umgegangen?
- Konnten Softwarefehler aufgedeckt werden?
- Wurde auf Limitationen von Pact gestoßen?
- Wurde bei der Entwicklung ein Aufwandsunterschied zwischen den consumerseitigen und providerseitigen Tests festgestellt?

Abbildung A8: Auszug des Interview-Leitfadens, in welchem sich auf die Implementierung der CDC Tests bezogen wird.

CDCT im Allgemeinen

- Vergleich von CDCT mit anderen Testverfahren:
  - Mit welchen Testverfahren, die im ODS eingesetzt werden, wurde bisher gearbeitet?
  - Einarbeitungsaufwand: Gab es Unterschiede zwischen CDCT und den anderen Testverfahren?
  - Entwicklungsaufwand: Gab es Unterschiede zwischen CDCT und den anderen Testverfahren?
  - Aufwand, um Tests zur Ausführung zu bringen: Gab es Unterschiede zwischen CDCT und den anderen Testverfahren?
  - Laufzeit: Gab es Unterschiede zwischen CDCT und den anderen Testverfahren?
- Vorteile, Nachteile, Herausforderungen und Richtlinien:
  - Welche Vorteile werden mit CDCT assoziiert?
  - Welche inhärenten Nachteile werden mit CDCT assoziiert?
  - Welche bewältigbaren Herausforderungen werden mit CDCT assoziiert?
  - Welche Richtlinien werden mit CDCT assoziiert? Wird die Einhaltung dieser Richtlinien als sinnvoll erachtet?

Abbildung A9: Auszug des Interview-Leitfadens, in welchem sich auf CDCT im Vergleich zu anderen Testverfahren und im Allgemeinen bezogen wird.

# D Eingebrachte Integrationsfehler des Defect Seedings

Für die nachfolgenden Tabellen gilt:

- Ein 3-Symbol gibt an, dass mindestens ein Testfall des entsprechenden Testverfahrens fehlgeschlagen ist und der Integrationsfehler somit offenbart wurde.
  - Ein 3*-Symbol gibt an, dass hierbei eine consumerseitige Interaktion zu den CDC Tests ergänzt wurde, ohne die der Fehler nicht aufgedeckt worden wäre.
- Ein 7-Symbol gibt an, dass kein Testfall des entsprechenden Testverfahrens fehlgeschlagen ist und der Integrationsfehler somit nicht offenbart wurde.
- In grau aufgelistete Ergebnisse der E2E Tests sind nicht mit den jeweiligen Ergebnissen der CDC Tests vergleichbar. Der Grund dafür ist, dass die Integrationen, in welche die Fehler eingebracht wurden, in diesen Fällen nicht durch die E2E Tests abgedeckt wurden. Weitere Details hierzu sind Sektion 2.5.2 der kurzgefassten Ausarbeitung zu entnehmen.

| | | | Operation | CDC | E2E |
|---|---|---|---|---|---|
| HTTP-Anfrage | Query-Param. | opt. | Umbenennung | 3 | 7 |
| | | | Wertebereichsvergrößerung | 3* | 7 |
| | Pfad-Param. | | Wertebereichsvergrößerung | 3* | 7 |
| | JSON-Attribut | opt. | Umbenennung | 3 | 7 |
| | | oblig. | Entfernung | 3 | 7 |
| | | | Umbenennung | 3 | 7 |
| | | | Wertebereichsvergrößerung | 3* | 7 |
| HTTP-Antwort | JSON-Attribut | opt. | Umbenennung | 3 | 7 |
| | | | Hinzufügung | 3* | 7 |
| | | | Wertebereichsverkleinerung | 7 | 7 |
| | | oblig. | Umbenennung | 3 | 7 |
| | | | Hinzufügung | 3 | 7 |
| | | | Wertebereichsverkleinerung | 7 | 7 |
| Async. Nachricht | JSON-Attribut | opt. | Umbenennung | 3 | 7 |
| | | | Hinzufügung | 3* | 7 |
| | | | Wertebereichsverkleinerung | 7 | 7 |
| | | oblig. | Umbenennung | 3 | 3 |
| | | | Hinzufügung | 3 | 7 |
| | | | Wertebereichsverkleinerung | 3 | 3 |

Tabelle A1: Consumerseitige, systematisch hergeleitete Integrationsfehler, die im Rahmen des Defect Seedings in das ODS-System eingebracht wurden. In dieser Tabelle existiert keine Spalte „Voraussetzung“, da für die hier aufgelisteten Integrationsfehler keine Voraussetzungen identifiziert wurden.

| | | | **Operation** | **Voraussetzung** | **CDC** | **E2E** |
|---|---|---|---|---|---|---|
| HTTP-Anfrage | Query-Param. | opt. | Entfernung | Mindestens ein Consumer verwendet den Query-Parameter | ✓ | ✗ |
| | | | Umbenennung | Mindestens ein Consumer verwendet den Query-Parameter | ✓ | ✗ |
| | | | Wertebereichsverkleinerung | Mindestens ein Consumer verwendet den Query-Parameter | ✗ | ✗ |
| | Pfad-Param. | | Wertebereichsverkleinerung | | ✗ | ✗ |
| | JSON-Attribut | opt. | Entfernung | Mindestens ein Consumer verwendet das JSON-Attribut | ✓ | ✗ |
| | | | Umbenennung | Mindestens ein Consumer verwendet das JSON-Attribut | ✓ | ✗ |
| | | | Wertebereichsverkleinerung | Mindestens ein Consumer verwendet das JSON-Attribut | ✗ | ✗ |
| | | oblig. | Entfernung | | ✗ | ✓ |
| | | | Umbenennung | | ✓ | ✓ |
| | | | Hinzufügung | | ✓ | ✓ |
| | | | Wertebereichsverkleinerung | | ✗ | ✗ |
| HTTP-Antwort | JSON-Attribut | opt. | Entfernung | Mindestens ein Consumer erwartet das JSON-Attribut | ✓ | ✗ |
| | | | Umbenennung | Mindestens ein Consumer erwartet das JSON-Attribut | ✓ | ✗ |
| | | | Wertebereichsvergrößerung | Mindestens ein Consumer erwartet das JSON-Attribut | ✗ | ✗ |
| | | oblig. | Entfernung | Mindestens ein Consumer erwartet das JSON-Attribut | ✓ | ✓ |
| | | | Umbenennung | Mindestens ein Consumer erwartet das JSON-Attribut | ✓ | ✓ |
| | | | Wertebereichsvergrößerung | Mindestens ein Consumer erwartet das JSON-Attribut | ✗ | ✗ |
| Async. Nachricht | JSON-Attribut | opt. | Entfernung | Mindestens ein Consumer erwartet das JSON-Attribut | ✓ | ✗ |
| | | | Umbenennung | Mindestens ein Consumer erwartet das JSON-Attribut | ✓ | ✗ |
| | | | Wertebereichsvergrößerung | Mindestens ein Consumer erwartet das JSON-Attribut | ✗ | ✗ |
| | | oblig. | Entfernung | Mindestens ein Consumer erwartet das JSON-Attribut | ✓ | ✓ |
| | | | Umbenennung | Mindestens ein Consumer erwartet das JSON-Attribut | ✓ | ✓ |
| | | | Wertebereichsvergrößerung | Mindestens ein Consumer erwartet das JSON-Attribut | ✗ | ✗ |

Tabelle A2: Providerseitige, systematisch hergeleitete Integrationsfehler, die im Rahmen des Defect Seedings in das ODS-System eingebracht wurden.

| | Änderung | Voraussetzung | CDC | E2E |
|---|---|---|---|---|
| Consumerseitig | Änderung des erwarteten Status Codes einer HTTP-Antwort | | 3 | 7 |
| | Entfernung eines HTTP-Headers | Der Provider erwartet den Header | 3 | 7 |
| | Umbenennung eines HTTP-Headers | Der Provider erwartet den Header | 3 | 7 |
| | Änderung der statischen URL eines Schnittstellenaufrufs | | 3 | 7 |
| | Änderung der verwendeten HTTP-Methode eines Schnittstellenaufrufs | | 3 | 7 |
| Providerseitig | Änderung des Status Codes einer HTTP-Antwort | Der Consumer erwartet den Status Code | 3 | 7 |
| | Entfernung eines HTTP-Headers | Der Consumer erwartet den Header | 3 | 3 |
| | Umbenennung eines HTTP-Headers | Der Consumer erwartet den Header | 3 | 3 |
| | Entfernung einer Schnittstellenoperation | Der Consumer verwendet die Schnittstellenoperation | 3 | 3 |
| | Änderung der statischen URL einer Schnittstellenoperation | Der Consumer verwendet die Schnittstellenoperation | 3 | 7 |
| | Änderung der erwarteten HTTP-Methode einer Schnittstellenoperation | Der Consumer verwendet die Schnittstellenoperation | 3 | 3 |

**Tabelle A3**: Bezüglich HTTP hergeleitete Integrationsfehler, die im Rahmen des Defect Seedings in das ODS-System eingebracht wurden.

# Literaturverzeichnis


Aderaldo, C. M., Mendonça, N. C., Pahl, C. & Jamshidi, P. (2017). Benchmark Requirements for Microservices Architecture Research. *2017 IEEE/ACM 1st International Workshop on Establishing the Community-Wide Infrastructure for Architecture-Based Software Engineering (ECASE)*, 8–13. https://doi.org/10.1109/ECASE.2017.4

Baskerville, R. L. (1999). Investigating information systems with action research. *Communications of the association for information systems*, *2*(1), 19. https://doi.org/10.17705/1CAIS.00219

Clemson, T. (2014). *Testing Strategies in a Microservice Architecture*. https://martinfowler.com/articles/microservice-testing/, zuletzt aufgerufen am 28. Dezember 2021.

Cohn, M. (2010). *Succeeding with Agile: Software Development Using Scrum*. Addison-Wesley.

Conway, M. E. (1968). How Do Committees Invent? *Datamation*. https://www.melconway.com/research/committees.html, zuletzt aufgerufen am 18. Dezember 2021.

Docker, Inc. (2021). *Docker Documentation*. https://docs.docker.com/, zuletzt aufgerufen am 27. Dezember 2021.

Fielding, R., Gettys, J., Mogul, J., Frystyk, H., Masinter, L., Leach, P. & Berners-Lee, T. (1999). Hypertext Transfer Protocol – HTTP/1.1. https://doi.org/10.17487/RFC2616

GitHub, Inc. (2021). *GitHub Actions Documentation*. https://docs.github.com/en/actions, zuletzt aufgerufen am 30. Dezember 2021.

Kitchenham, B. & Charters, S. (2007). *Guidelines for performing Systematic Literature Reviews in Software Engineering* (Techn. Ber. EBSE-2007-01). Keele University und Durham University Joint Report.

Koschel, A., Astrova, I., Bartels, M., Helmers, M. & Lyko, M. (2021). On Testing Microservice Systems. In K. Arai, S. Kapoor & R. Bhatia (Hrsg.), *Proceedings of the Future Technologies Conference (FTC) 2020, Volume 3* (S. 597–609). Springer International Publishing. https://doi.org/10.1007/978-3-030-63092-8_40

Lehvä, J., Mäkitalo, N. & Mikkonen, T. (2019). Consumer-Driven Contract Tests for Microservices: A Case Study. In X. Franch, T. Männistö & S. Martínez-Fernández (Hrsg.), *Product-Focused Software Process Improvement* (S. 497–512). Springer International Publishing. https://doi.org/10.1007/978-3-030-35333-9_35

Lewis, J. & Fowler, M. (2014). *Microservices: a definition of this new architectural term*. https://martinfowler.com/articles/microservices.html, zuletzt aufgerufen am 6. Dezember 2021.

Li, H., Wang, J., Dai, H. & Lv, B. (2020). Research on Microservice Application Testing System. *2020 IEEE 3rd International Conference on Information Systems and Computer Aided Education (ICISCAE)*, 363–368. https://doi.org/10.1109/ICISCAE51034.2020.9236829

Namiot, D. & Sneps-Sneppe, M. (2014). On Micro-services Architecture. *2*(9), 24–27. http://injoit.org/index.php/j1/article/view/139, zuletzt aufgerufen am 18. November 2021.

Newman, S. (2021). *Building Microservices: Designing Fine-Grained Systems* (2nd). O'Reilly Media, Inc.

Pact Foundation. (2021). *Pact Docs*. https://docs.pact.io/, zuletzt aufgerufen am 21. Dezember 2021.

Parnas, D. (1971). Information Distribution Aspects of Design Methodology. *IFIPS Congress*, *71*, 339–344.

Postel, J. (1981). *Transmission Control Protocol* (RFC Nr. 793). Internet Engineering Task Force. https://doi.org/10.17487/RFC0793

Raychev, N. (2020). Test automation in microservice architecture. *IEEE-SEM Journal Publication*, *8*, 42–56.

Richardson, C. (2018). *Microservices Patterns: With examples in Java*. Manning.

Robinson, I. (2006). *Consumer-Driven Contracts: A Service Evolution Pattern*. https://martinfowler.com/articles/consumerDrivenContracts.html, zuletzt aufgerufen am 15. Dezember 2021.

Wohlin, C. (2014). Guidelines for Snowballing in Systematic Literature Studies and a Replication in Software Engineering. *Proceedings of the 18th International Conference on Evaluation and Assessment in Software Engineering*. https://doi.org/10.1145/2601248.2601268

Wolff, E. (2017). *Microservices: Flexible Software Architecture*. Addison-Wesley.