HAW HAMBURG

BACHELORTHESIS

Dominik Heinrich Tobaben

Konzeption und Evaluierung eines Consumer-Driven Contract Testing Ansatzes in der BI

FAKULTÄT TECHNIK UND INFORMATIK
Department Informatik

Faculty of Computer Science and Engineering
Department Computer Science

HOCHSCHULE FÜR ANGEWANDTE WISSENSCHAFTEN HAMBURG
Hamburg University of Applied Sciences

Dominik Heinrich Tobaben

Konzeption und Evaluierung eines Consumer-Driven Contract Testing Ansatzes in der BI

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung
im Studiengang *Bachelor of Science Angewandte Informatik*
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer: Prof. Stefan Sarstedt
Zweitgutachter: Prof. Martin Schultz

Eingereicht am: 07. Juli 2022

Dominik Heinrich Tobaben

Thema der Arbeit

Konzeption und Evaluierung eines Consumer-Driven Contract Testing Ansatzes in der BI

Stichworte

CDC, CDCT, BI, DWH, Testing, Business Intelligence, Data Warehouse, Data Mesh

Kurzzusammenfassung

Consumer-Driven Contract Testing (CDCT) ist ein bewährtes Konzept für die Stabilisierung von Schnittstellen aus der Softwareentwicklung. In der Business Intelligence (BI) sind unerwartete Änderungen an den Schnittstellen eine häufige Ursache für Störungen im Betriebsablauf der Ladeprozesse. CDCT hat sich in der BI noch nicht etabliert und dessen Evaluierung war Ziel der Arbeit. Ergebnis ist ein auf die BI zugeschnittenes und erweiterbares CDCT-Framework, dass die Erstellung leicht zu implementierender CDCs für Quellsysteme sowie andere BI Teams ermöglicht.

Dominik Heinrich Tobaben

Title of Thesis

Design and evaluation of a Consumer-Driven-Contract Testing approach in BI

Keywords

CDC, CDCT, BI, DWH, Testing, Business Intelligence, Data Warehouse, Data Mesh

Abstract

Consumer-Driven Contract Testing (CDCT) is a proven concept in software development for stabilizing interfaces. For Business Intelligence (BI), unexpected changes in interfaces are often the cause behind failures in data pipelines. CDCT has not yet established itself in BI and its evaluation was the target objective in this work. The result is an extensible CDCT framework tailored to BI Usecases, which enables the creation of easy-to-implement CDCs for source systems and other BI teams.

Inhaltsverzeichnis

Abbildungsverzeichnis vii

Tabellenverzeichnis viii

Listings ix

Abkürzungsverzeichnis xi

Glossar xiv

1 Einleitung 17

1.1 Motivation 17

1.2 Zielsetzung 18

1.3 Aufbau der Arbeit 18

2 Grundlagen 19

2.1 Business Intelligence 19

2.2 Data Warehousing 20

2.3 Qualitätssicherung in BI-Architekturen 22

2.3.1 Während der Entwicklung 22

2.3.2 Stabilisierung von Schnittstellen 23

2.3.3 Im Betrieb 24

2.4 Consumer-Driven Contract Testing 25

3 Rahmenbedingungen 26

3.1 Das Unternehmen 26

3.2 Business Intelligence in der PEG 27

3.3 Schnittstellen 28

3.3.1 Schnittstellen in einer BI Schichtenarchitektur 28

3.3.2 Schnittstellen durch Kombination aus Data Lake und Data Warehouse 29

3.3.3 Schnittstellen in der PEG Data-Mesh Architektur ... 29
3.3.4 Schnittstellen gesamt ... 30
3.4 Bekannte/Bestehende CDCT Prozeduren ... 30
3.4.1 Pact ... 31
3.4.2 Docker CDCs ... 33
3.4.3 AWS Lambda ... 35
4 Anforderungen ... 37
4.1 Durch PEG Architekturvorgaben / Domainteams ... 37
4.2 Anforderungen seitens der BI ... 37
4.3 Anforderungskatalog ... 38
5 Entwurf ... 40
5.1 Architekturübersicht ... 40
5.2 Entwurf des CDCT-Frameworks ... 43
5.3 Grundlage für den Entwurf ... 44
5.4 Auswahl der CDCT Prozedur ... 45
5.5 Auswahl der Programmiersprache ... 46
5.6 Auswahl der Validierungsbibliotheken ... 47
5.6.1 CSV-Schema Validierung ... 47
5.6.2 JSON Schema Validierung ... 48
5.6.3 SQL Query Schema Validierung ... 51
5.7 Entwurf einer Validierungsbibliothek für SQL-Abfragen ... 52
5.7.1 SQL-Schema Definition (SQSD) ... 52
5.8 Zusammenfassung der Validierungsmöglichkeiten ... 55
6 Implementierung ... 56
6.1 Sourcecode Repository ... 56
6.2 Etablieren eines neuen CDCs ... 58
6.3 Implementierung einer Datenverbindung ... 61
6.4 Implementierung eines Validators ... 64
6.5 Implementierung der SQL Query Definition Bibliothek ... 65
6.5.1 Implementierung des BigQueryValidators im CDCT-Framework ... 66
7 Evaluierung / Bewertung ... 68

7.1 Bewertung der Anforderung A1 68
7.2 Bewertung der Anforderung A2 69
7.3 Bewertungen der Anforderungen A3 bis A5 69
7.4 Wesentliche Erkenntnisse 70
7.5 Einschränkungen 71
8 Fazit 71
8.1 Ausblick 72

Abbildungsverzeichnis

Abbildung 1: Abgrenzung zwischen Business Intelligence und Data warehousing (Murillo 2016) 20

Abbildung 2: Klassische Data Warehouse Architektur (Jung und Winter 2000, 45) 21

Abbildung 3: Evolution of BI Architectures (Armbrust et al) 21

Abbildung 4: Testpunkte in einer Data Warehouse Architektur (Collier 2012, 209) 23

Abbildung 5: Testpyramide inkl. Consumer-Driven Contract Tests (Lehvä et al. 2019, 498) 25

Abbildung 6: Generelle Event Struktur (OTTO MA-EC E-Commerce Innovation & -Plattform 2018) 27

Abbildung 7: Übersicht einer Data-Mesh-Architektur (Dehghani 2020)..... 30

Abbildung 8: CDCT mit Pact.io (Pact Foundation 2021) 32

Abbildung 9: PEG Docker Registry CDC-T Ansatz (Kotwal (Thoughtworks) 2020) 34

Abbildung 10: CDC-T Ansatz der Otto GmbH für die Webshop Plattform auf otto.de (Vollerthun 2021)..... 36

Abbildung 11: Zielbild von der Erstellung eines CDCs durch den Consumer, bis zur Nutzung auf Seite des Producers. 41

Abbildung 12: Vereinfachtes UML Klassendiagramm des CDCT-Frameworks..... 43

Abbildung 13: UML Sequenzdiagram des CDCT-Frameworks..... 44

Abbildung 14: ETL Datapipeline visualisiert - Getestet werden soll die Verbindung und das Format des Inputs, nicht aber der Transform Teil der ETL-Prozesse. (Dehghani 2020)..... 45

Abbildung 15: Nicht alle Datenformate können von jedem Validator verarbeitet werden..... 62

Abbildung 16: Klassendiagram der SQSD Bibliothek...... 66

Tabellenverzeichnis

Tabelle 1: Die nötigen Parameter für die Konfiguration eines CDCT Docker Images........... 42

Tabelle 2: Attribute des SQSD JSON-Objekts 53

Tabelle 3: Attribute des SQSD Spalten-JSON-Objekts 53

Tabelle 4: Zusammenfassung der Validierungsmöglichkeiten in Bezug auf die Anforderungen A2 und A5.. 55

Listings

Listing 1: Voraussetzung für den Unit Test ist das Erzeugen eines Pacts, im Beispiel ein Teil des JUnit Tests. (Baeldung 2020) ... 32

Listing 2: Pact JUnit Test auf Provider Seite am Beispiel einer Spring Applikation. (Baeldung 2020) ... 33

Listing 3: Beispielhafte erfolgreiche Ausführung eines vom Consumer bereitgestellten Docker CDC Images ... 35

Listing 4: Beispiel für eine CSVS-Definition für die Validierung von Wertebreichen, Datentypen und regulären Ausdrücken. ... 48

Listing 5: Eine zu Listing 4 passende CSV-Datei. ... 48

Listing 6: Beispiel für eine JSON-Schemadefinition für die Validierung. ... 50

Listing 7: Ein zu Listing 6 passendes JSON Dokument. ... 50

Listing 8: Konzept einer „SQL Query Definition“ für die Validierung von Datenbankabfragen ... 54

Listing 9: Ordnerstruktur des Projektes ... 57

Listing 10: GitHub Action für die Erstellung eines CDC in Verwendung des CDC Templates (*cdc-templates.yml*) ... 59

Listing 11: Beispielhafte Ausführung eines CDCs in der GitHub Action Buildpipeline des Producers. ... 61

Listing 12: Das *MessageConsumer* Interface ... 62

Listing 13: Die *GenericMessageConsumer* Klasse ... 63

Listing 14: Die *SqsConsumer* Klasse ... 63

Listing 15: Der Kern der abstrakten *CDCTValidator* Klasse 64

Listing 16: Implementierung der Methode *getSupportedMessageClasses* des *JsonValidators* 65

Listing 17: Implementierung der Methode *validate* des *JsonValidators* 65

Listing 18: Implementierung des *BigQueryValidator* als *CDCTValidator*............................. 67

Abkürzungsverzeichnis

API Application Programming Interface, Programm Schnittstelle

BI Business Intelligence. Oberbegriff für die Analyse und datengetriebene Entscheidungsfindung mit Hilfe der Unternehmensdaten.

B2B Business to Business, Unternehmen zu Unternehmen

B2C Business to Customer, Unternehmen zu (privat) Kunde

CDC Consumer-Driven Contract, vom konsumierenden System getriebener Schnittstellenvertrag mit dem produzierenden System

CDCT Customer-Driven Contract Testing, das Testen des vom konsumierenden System getriebenen Schnittstellenvertrages

CSV Comma-separated values, Dateiformat, bei dem Werte mit Kommas voneinander getrennt werden. In dieser Arbeit steht CSV-Datei für alle Textdateien, dessen Werte durch einen beliebigen Separator getrennt werden, also auch Tab oder „|“.

CI / CD Continuous Integration / Continuous Delivery. CI: Kontinuierliches Integrieren der Änderungen in den Hauptzweig des Codes, auch

	„Master“, „Main“, oder „Trunk“ genannt. CD: Kontinuierliches „bauen“ und ausrollen der letzten Änderungen.
DDL	Data Definition Language, Bestandteil der SQL für das Beschreiben, Verändern oder Löschen von Datenbankobjekten wie Tabellen, Routinen und Views.
DWH	Data Warehouse
ETL / ELT	Extract Transform Load, Extract Load Transform
FTP	File Transport Protocol, Protokoll für die Übertragung von Dateien
GCP	Google Cloud Plattform, Google Cloud-Dienste
GCS	Google Cloud Storage, REST basierter Dienst zum Speichern von Dateien
MVP	Minimum Viable Product, Minimal einsatzfähiges Produkt
OLAP	On-Line Analytical Processing
OLTP	On-Line Transactional Processing
REST	Represential State Transfer, Zustandsorientierter Architekturstil für APIs
DDD	Domain-Driven-Design, Werkzeugkasten für die Modellierung von Software

S3	Amazon Simple Storage Service, REST basierter Dienst zum Speichern von Dateien
SNS	Simple Notification Service, Amazon Cloud-Dienst für Nachrichtenverteilung an viele Empfänger gleichzeitig
SQL	Structured Query Language, Abfragesprache für eine Datenbank
SQS	Simple Queue Service, Amazon Cloud-Dienst für Warteschlangenbasierte Nachrichtenverteilung
XML	Extensible Markup Language, Textformat für hierarchisch strukturierte Daten
PEG	Payment Entwicklungsgesellschaft mbH, Tochterfirma der OTTO GmbH & Co. KG, zukünftiger Zahlungsdienstleister des Otto Marktplatzes

Glossar

BI (IT gestützt)

In der IT BI geht es darum, „wie man eines der wertvollsten Güter eines Unternehmens - Rohdaten - erfasst, zugänglich macht, versteht, analysiert und in verwertbare Informationen umwandelt, um die Unternehmensleistung zu verbessern." (Azvine et al. 2005, 215)

Compliance

Die Einhaltung der sogenannten „Compliance" bedeutet, sich an vom Gesetzgeber vorgeschriebenen Regulierungen zu halten, dazu zählen zum Beispiel die Datenschutz-Grundverordnung (DSGVO/GDPR) der Europäischen Union oder die Mindestanforderungen an das Risikomanagement durch die Bundesanstalt für Finanzdienstleistungsaufsicht (BaFin). Aber auch die vom Unternehmen selbst vorgegebenen Prozesse gehören in den Bereich Compliance, beispielsweise die Einhaltung gewisser Namenskonventionen.

Consumer-Driven Contract (Testing)

Beim Consumer-Driven Contract Testing Pattern stellt der konsumierende Dienst (Consumer) Erwartungen an den bereitstellenden Dienst (Provider) und formt damit einen Vertrag (Contract). Möchte der Provider Änderungen an der Schnittstelle vornehmen, kann mithilfe des Contracts geprüft werden, ob die Erwartungen des Consumers weiterhin erfüllt werden. Das Pattern schreibt jedoch nicht vor, in welcher Form und Struktur CDCs implementiert werden. (Robinson 2006)

Bei Otto bedeutet CDCT, dass der Consumer dem Provider einen ausführbaren Test bereitstellt, der in die Build Pipeline des Providers eingebunden wird. (Stamm 2013)

Data Lake

"Der Data Lake hingegen nimmt die Daten aus den unterschiedlichen Quellen in ihrem Rohformat auf und legt sie auch unstrukturiert ab. Dabei ist es unerheblich, ob die Daten für spätere Analysen relevant sind. Der Data Lake besitzt eine flache Hierarchie und muss für die Speicherung der Daten nicht die Art der später auszuführenden Analysen kennen." (Luber 2018)

Data Ownership

Data Ownership beschreibt, wer für die Daten, dessen Betrieb und Bereitstellung innerhalb des Unternehmens bzw. BI Systems verantwortlich ist.

Das Wort "Data Ownership" findet auch in rechtlichen Diskussionen Verwendung, beispielsweise wenn es um das Recht auf informelle Selbstbestimmung im Zusammenhang mit der Datenschutzgrundverordnung (DSGVO) geht. (Drexl et al.). In dieser Arbeit wird auf rechtliche Aspekte jedoch nicht eingegangen.

Data-Mesh

„Die Data-Mesh-Plattform ist eine bewusst verteilte Datenarchitektur mit zentraler Steuerung und Standardisierung für Interoperabilität, die durch eine gemeinsame und harmonisierte Self-Service-Dateninfrastruktur ermöglicht wird.“ (Dehghani 2021)

Domain Driven Design

DDD beschreibt Werkzeuge für den Entwurf und die Implementierung von Software. Dabei wird zwischen strategischem und

taktischem Design unterschieden. Im strategischen Design werden geschäftliche Kontexte bzw. Domänen voneinander abgegrenzt (sogn. Bounded Context) und Begriffe innerhalb Dieser zu einer Sprache zusammengefasst (sogn. Ubiquitous Language). So wird Mehrdeutigkeiten von Begriffen in verschiedenen Domänen begegnet. Beispielsweise hat ein "Kunde" in der Domäne Logistik andere Attribute und Bedeutungen als in der Marketing-Domäne. Im taktischen Design werden Entitäten und Wertobjekte in sogenannte Aggregate zusammengefasst. Beispielsweise könnte innerhalb der Logistikdomäne das Aggregat "Kunde" eine Entität "Person" mit dem Wertobjekt "Lieferadresse" enthalten. (Vernon 2017, 1–8)

Domainteam / Produktteam Die Produkte innerhalb der PEG wurden nach DDD geschnitten. Die Softwareentwickler des Produktes und dessen Fachbereich befinden sich meist im selben Team.

OLTP / OLAP Bei OLTP Datenbanken steht die Verarbeitung von Transaktionen in Echtzeit, sowie ein möglichst hoher Datendurchsatz bei parallelen Zugriffen mit minimalen Antwortzeiten von Detailabfragen im Vordergrund. Üblicherweise wird keine Historie der Datensätze vorgehalten, sodass immer nur der aktuelle Zustand eines Datums zur Verfügung steht. Im Gegensatz zu OLTP Systemen fokussiert sich ein OLAP System auf die möglichst schnelle Antwortzeit auf Analytische Abfragen inklusive historischer und zusammengeführter Daten. (Gabriel et al. 2009, 11)

1 Einleitung

1.1 Motivation

Business Intelligence (BI) Abteilungen arbeiten üblicherweise nicht wie reine Entwicklungsabteilungen. Durch den Fokus auf Daten, deren Struktur und vor allem ihrer fachlichen Bedeutung wird sich häufig auf Datenmodellierung beschränkt und fertige, meist visuell unterstützte Lösungen wie Talend-DI, Pentaho oder IBM DataStage für die Datenintegration verwendet, um ETL bzw. ELT Pattern umzusetzen. Dabei werden häufig direkte Datenbankverbindungen oder Datei basierte Schnittstellen wie CSV-Dateien, aber auch nachrichtenbasierte Quellen angebunden.

Häufig passiert es, dass die Quelldaten undurchschaubare implizite Abhängigkeiten haben und nicht immer klar ist, wann, warum und wie sich eine Datenschnittstelle ändert. Bekannte Beispiele sind sich ändernde Datentypen in der Datenbank durch Updates der Software des Quellsystems als auch XML-Validierungsdateien, die zum Zeitpunkt der Anbindung aktuell waren, im Laufe der Zeit aber nie aktualisiert wurden und ungültig werden. Darüber hinaus werden Ladeprozesse in großen Unternehmen im Wandel der Zeit von unterschiedlichen Teams und Personen betrieben und verantwortet, während bei jeder Übergabe Wissen verloren geht. Das Resultat sind immer höher werdende Aufwände für Optimierungen, Erweiterungen, Änderungen, Migrationen und Abschaltungen.

Durch Migration der BI in die Google Cloud finden gezwungenermaßen mehr an der Softwareentwicklung angelehnte Pattern und Methodiken Anwendung. In den (Web-)Entwicklungsabteilungen von Otto haben für den Microservices basierten Architekturstil sogenannte „Consumer-Driven-Contracts“ (CDC) etabliert und für transparente Verantwortung, Wartbarkeit und Verlässlichkeit durch Service Level Agreements (SLA) gesorgt. (Stamm 2013)

Solange eine Änderung in der Schnittstelle noch nicht vom BI-Team verstanden und im Ladeprozess angepasst wurde, kann der vom BI-Team belieferte Fachbereich nicht mit Daten versorgt werden, was wiederum hohe Kosten verursachen kann.

Statt wie bisher in der BI nur festzulegen, wie auf die Daten zugegriffen wird, können durch Kontrakte mit SLA zwischen den Consumern (BI-Team oder Fachbereich) und Provider (BI-Team oder Quellsystem) auch die Schnittstellen samt Inhalt bzw. Schema beschreiben werden. Verträge dokumentieren den Sinn und Zweck der Datenanbindung, die Frage „Wer benutzt die Daten eigentlich?“ kann auch beim Wechsel der Verantwortlichkeit beantwortet werden. Aus Kontrakten können automatisiert Tests generiert werden, ohne dabei von Infrastruktur des anderen Teams abhängig zu sein. Das automatische Testen der Kontrakte bei jeder Änderung im Quellsystem stellt zudem sicher, dass „breaking-changes“ kommuniziert und als neue Verträge verhandelt werden müssen. Dadurch verringert sich gleichzeitig der Betriebsaufwand in den BI-Teams, die sonst üblicherweise bei der Quelle nachfragen, ob es Änderungen gab und was diese bedeuten (z.B. Änderung der Kommastellen bei Zins-Feldern).

1.2 Zielsetzung

Im Rahmen der Bachelorarbeit soll der Einsatz von CDCT Methodiken und Frameworks im BI Umfeld verglichen und erprobt werden. Insbesondere sollen CDCs nicht nur für Schnittstellen vom Quellsystem zum BI-System, sondern auch innerhalb der BI-Systeme als weitere Teststrategie etabliert werden. Ein BI fokussiertes CDCT-Framework soll daher neben den bereits durch gängige Frameworks wie Pact abgedeckte Schnittstellen auch BI typische testen können.

1.3 Aufbau der Arbeit

Zuerst wird das BI Umfeld untersucht sowie Probleme, welche durch CDCT gelöst werden könnten, analysiert. Anschließend werden sich im Einsatz befindliche CDCT Konzepte erfasst, verglichen und für den möglichen Einsatz evaluiert. Zum Abschluss soll ein Konzept mit mindestens einem Domainteam innerhalb der OTTO PEG produktiv verwendet und bewertet werden.

2 Grundlagen

Im folgenden Kapitel wird Business Intelligence von Data Warehousing unterschieden sowie Consumer-Driven Contract Testing in der Testpyramide eingeordnet. Ein besonderes Augenmerk wird dabei auf die in BI-Architekturen üblichen Maßnahmen zur Qualitätssicherung gelegt.

2.1 Business Intelligence

Im Tagesgeschäft eines Unternehmens werden von den Mitarbeitern sowie dem Management kontinuierlich Entscheidungen getroffen, die das Geschäft beeinflussen. Um eine fundierte Entscheidung treffen zu können, sind operative und dispositive Unternehmensdaten essenziell. In der BI geht es darum, „wie man eines der wertvollsten Güter eines Unternehmens - Rohdaten - erfasst, zugänglich macht, versteht, analysiert und in verwertbare Informationen umwandelt, um die Unternehmensleistung zu verbessern.“ (Azvine et al. 2005, 215)

Die für die Entscheidungsfindung wichtigen Daten befinden sich zum Großteil in den OLTP- bzw. operativen Systemen mit transaktionalen Datenbanken wie einem ERP- oder Shopsystem. Um die operativen Systeme nicht durch analytische Datenbankabfragen für Berichte und Analysen zu überlasten, haben sich separate OLAP-Systeme etabliert.

Die Daten für das OLAP-System müssen aus den heterogenen Systemen, welche sich auf unterschiedlichsten Plattformen befinden, extrahiert, in eine analyseoptimierte Struktur transformiert und in das Zielsystem geladen werden. Prozesse nach diesem Pattern werden als ETL-Prozess bezeichnet. (Jung und Winter 2000, 44)

Für die IT gestützte Business Intelligence werden die Daten in sogenannten Data Warehouses gespeichert. Die dadurch anfallenden Aufgaben werden als Data Warehousing zusammengefasst und werden auf Abbildung 1 voneinander abgegrenzt.

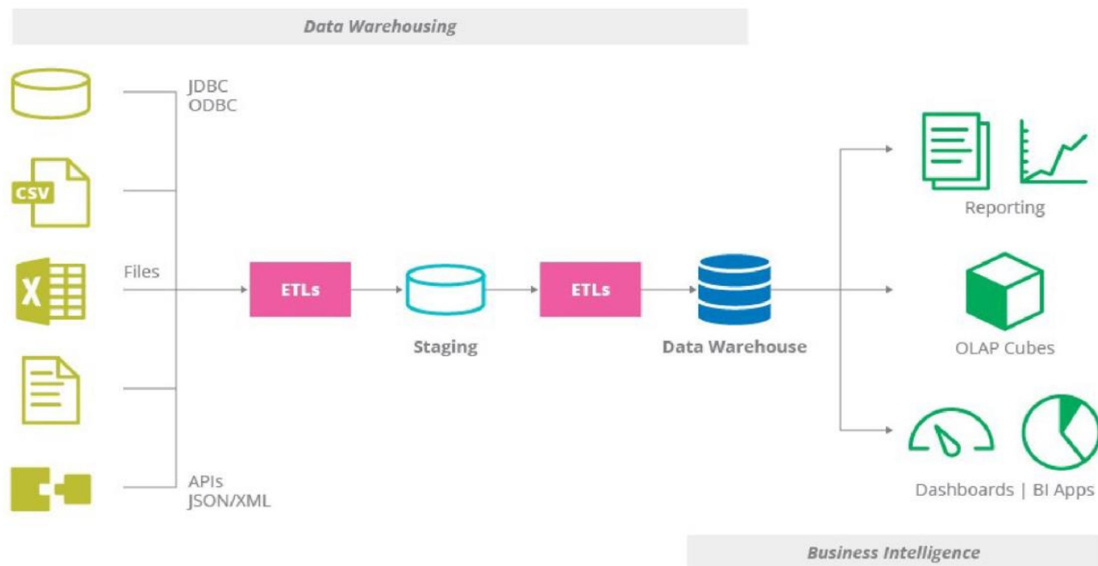


Abbildung 1: Abgrenzung zwischen Business Intelligence und Data warehousing (Murillo 2016)

2.2 Data Warehousing

Der Großteil der beschriebenen BI/DWH Architekturen haben gemeinsam, dass ETL-Prozesse das Rückenmark bilden und deren Stabilität dadurch die Wertschöpfung der BI-Architektur maßgeblich beeinflussen. ETL-Prozesse kommen in praktisch allen BI-Architekturen zum Einsatz, von klassischen Architekturen, zu sehen in der Abbildung 2, bis hin zu modernen Architekturkonzepten mit sogenannten Datalakes, siehe Abbildung 3. Aber auch in weiteren Konzepten wie der Message-Driven Datawarehouse Architektur, bei der Daten im Push Verfahren in das Datawarehouse gelangen, werden letzten Endes klassische ETL-Prozesse für die Aktualisierung der Data Marts verwendet. (Collier 2012, 177–179).

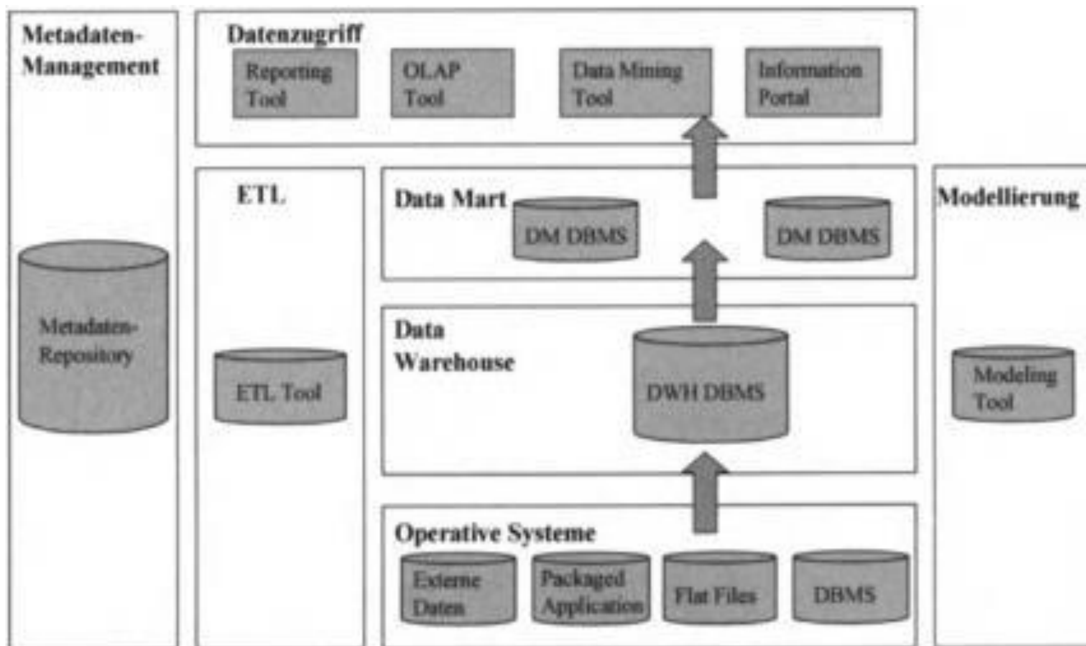


Abbildung 2: Klassische Data Warehouse Architektur (Jung und Winter 2000, 45)

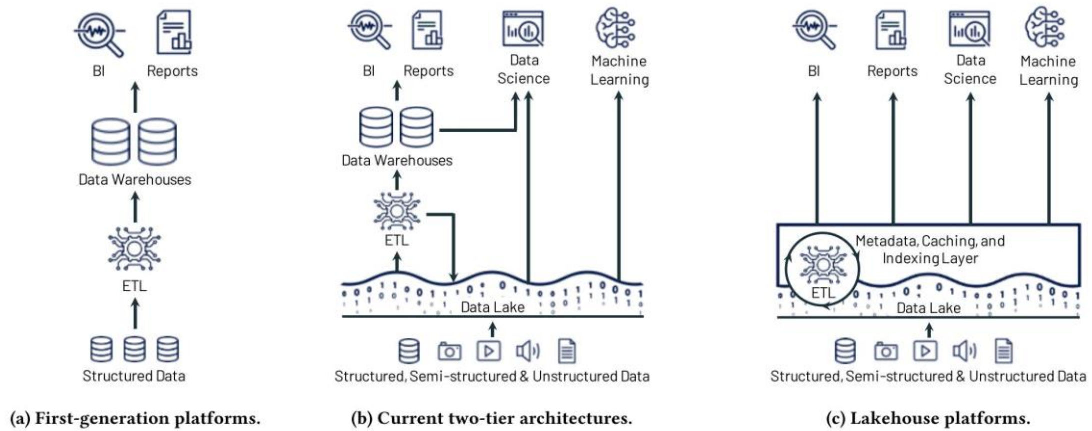


Abbildung 3: Evolution of BI Architectures (Armbrust et al)

2.3 Qualitätssicherung in BI-Architekturen

2.3.1 Während der Entwicklung

Neben manuellen Tests von Datenbankabfragen finden in der BI auch Test- und Story Test-Driven Development Anwendung. Um die Programmteile einer klassischen BI-Architektur nach Collier zu testen, kommen Sandbox Umgebungen je Entwickler und Integrations- sowie Preproduction Umgebungen zum Einsatz. In der Abbildung 4 zeigt Collier die notwendigen Punkte, an denen Tests durchgeführt werden sollten. Punkt 1 setzt dabei im Quellsystem an und liegt in der Verantwortung der Entwickler des operativen Systems. Die Punkte 2 bis 5 beziehen sich auf den Code für die Aktualisierung, die Bereinigung und Transformation der Daten, der üblicherweise innerhalb der ETL-Prozesse zu finden ist. Die Punkte 6 und 7 betreffen den Code für den Zugriff auf die Daten und der analytischen Modelle sowie Anpassungen an der BI Applikation selbst, damit Endnutzer die Daten aus dem Warehouse auch abfragen können. (Collier 2012, 210–224)

Das automatisierte Testen von Reports ist derweil selten anzufinden. Üblicherweise wird bei der Entwicklung eines Reports manuell geprüft, wohingegen die auf die Datenbank auszuführenden Abfragen wiederum in Testsuiten innerhalb der Testumgebung auch automatisch getestet werden können.

In Teilen bieten BI Plattformen wie MicroStrategy mit den sogenannten „Integrity Tests“ die Möglichkeit, SQL-Abfragen und Report Ergebnisse vor einer Änderung, sei es auf der Datenbank oder innerhalb des BI Tools, als „Baseline“ festzuhalten. Nach der Änderung kann gegen diese „Baseline“ getestet werden, sodass Änderungen an den Abfragen, den Ergebnissen oder Ausführungsfehler festgestellt werden können. (MicroStrategy Inc. 2021)

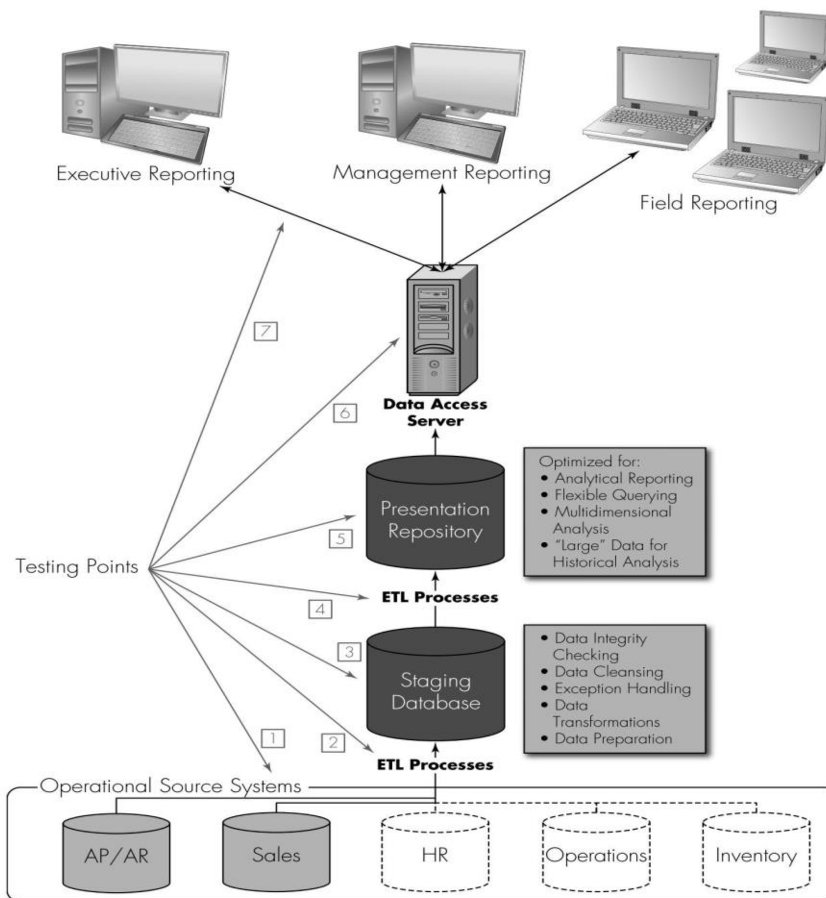


Abbildung 4: Testpunkte in einer Data Warehouse Architektur (Collier 2012, 209)

2.3.2 Stabilisierung von Schnittstellen

Häufig startet das Datawarehouse Projekt, nachdem bereits operative Systeme entwickelt wurden. Daher sind direkte Datenbankzugriffe auf die operativen Quellsysteme oder deren Replikation keine Seltenheit, führen aber zu einer hohen Kopplung der Systeme und Anfälligkeit der ETL-Prozesse bei Änderungen. Um diesem Problem zu begegnen, wird in einigen Unternehmen auf organisatorische Maßnahmen zurückgegriffen. Beispielsweise kann es Absprachen zwischen den operativen Entwicklungsabteilungen und der BI geben, sodass im Falle von Datenbankänderungen die ETL-Teams über die entsprechende Änderung vor dem produktiven Roll-out informiert werden.

In Zeiten von CI/CD wird dies zunehmend eine Herausforderung, weshalb eine Entkopplung der Systeme vorgenommen kann. Dafür kann ein Messagebus-System, wie von Collier beschrieben, verwendet werden (Collier 2012, 178). Soweit vorhanden, können auch APIs der Quellsysteme angebunden werden.

Eine weitere Option wäre das Entwickeln von BI-spezifischen Schnittstellen, wobei Änderungen bzw. Neuerungen an den Daten oder deren Struktur nur durch zusätzlichen Entwicklungsaufwand des operativen Entwicklungsteams in der BI ankommen. Der Zusatzaufwand führt daher nicht selten dazu, dass darauf aufbauende Reports oder Ad-hoc Analysen erst später möglich werden.

2.3.3 Im Betrieb

Durch die Jobautomatisierung ist ein Monitoring der ETL-Prozesse notwendig. Dabei können durch Überwachung der Laufzeiten aufkommende Performanceprobleme identifiziert und die Prozesse aufeinander abgestimmt werden. Außerdem können unerwartete Änderungen im Quellsystem erkannt und auf Probleme schnell reagiert werden. (Jung und Winter 2000, 170)

Häufig dauert die groß angelegte Aktualisierung des Data Warehouse mehrere Stunden. Da die Datenbank tagsüber in Benutzung durch die Anwender ist und am Morgen vor allem die Daten des Vortages interessant sind, findet die Aktualisierung meist im Zeitraum zwischen 0 und 6 Uhr statt.

Kommt es zu Breaking Changes im Quellsystem, kann ein früher Abbruch eines ETL-Prozesses in BI Abteilungen ohne nächtliche Bereitschaft dazu führen, dass geplante Berichte erst später oder gar am Nachmittag versendet werden können und die Fachabteilungen in der Entscheidungsfindung teilweise eingeschränkt werden. Das Monitoring kann für die Gesundheitszustandsanzeige verwendet werden.

Durch Neustart bzw. Retry-Mechanismen in den Prozessen können Abbrüche, die auf Verbindungsprobleme zurückzuführen sind, teilweise abgefedert werden. In diesen Fällen typische Fehlermeldungen finden sich im Monitoring häufig als „*TCP/IP Connection Lost*“, „*TCP/IP Connection reset*“, oder „*Too Many Connections*“.

2.4 Consumer-Driven Contract Testing

In der Fallstudie von Jyri Lehvä et al. wurde der Einsatz von Consumer-Driven-Contract Testing und dessen Auswirkungen auf die etablierten Testing-Methoden in einer Microservice Architektur analysiert. Verglichen mit den aufwendigen End-to-End und Integrationstests konnten die leichtgewichtigen CDC-Tests Breaking Changes in dem System zuverlässig und frühzeitig aufdecken, bevor diese Änderungen in produktionsähnlichen Umgebungen ausgerollt wurden. Abschließend konnten die CDC-Tests alle Fehlerfälle der Integrationstests bei geringem Aufwand aufdecken, sodass die CDCTs nicht nur die Test-Strategien wie in Abbildung 5 ergänzen, sondern auch die Integrationstests ersetzen könnten. (Lehvä et al. 2019, 507–512)

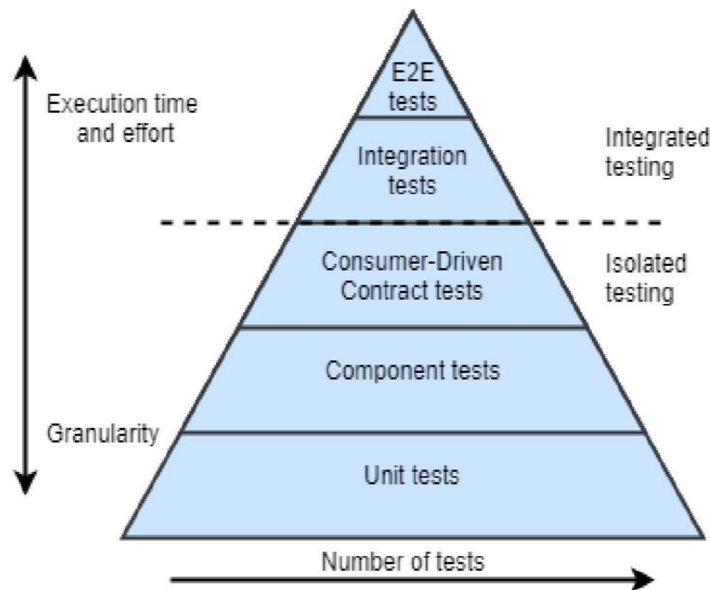


Abbildung 5: Testpyramide inkl. Consumer-Driven Contract Tests (Lehvä et al. 2019, 498)

3 Rahmenbedingungen

Die Evaluierung des CDCT Pattern findet in der BI Abteilung der OTTO Tochter Paymententwicklungsgesellschaft mbH (kurz PEG) statt. Die im Unternehmen verwendeten Technologien und geltenden Architekturvorgaben bilden dabei die Grundlage für spätere Designentscheidungen. Darüber hinaus werden Schnittstellen allgemein verbreiteter BI-Architekturen beschrieben und berücksichtigt.

3.1 Das Unternehmen

Die PEG soll der zukünftige Zahlungsdienstleister des Otto Marktplatzes werden, entsprechend werden hauptsächlich durch Zahlungsabwicklung anfallende Daten verarbeitet. (Klein 2020)

Die Organisation ist in Aufgabenbereiche, sogenannter Domänen aufgebaut. Die jeweiligen Produkte der Domänen werden innerhalb eines Domainteams entwickelt und betrieben. Als Infrastruktur kommt AWS zum Einsatz, sodass größtenteils dessen hauseigenen Produkte wie SNS/SQS für Nachrichtenkommunikation verwendet werden. Alle ein- und ausgehenden Domänenevents werden im JSON-Format verteilt. Dabei erfolgt synchrone Kommunikation über eine HTTP-REST-Schnittstelle gemäß der OpenAPI-[1] und JSON:API-Spezifikation[2]. Asynchrone Kommunikation erfolgt über AWS SNS Topics gemäß der AsyncAPI-Spezifikation[3]. Der Aufbau der Events ist der CloudEvents Spezifikation[4] angelehnt, siehe Abbildung 6.

[1] https://swagger.io/specification/

[2] https://jsonapi.org/format/

[3] https://www.asyncapi.com/docs/specifications/v2.0.0

[4] https://github.com/cloudevents/spec

Key	Required?	Format	Sample values	Description
eventId	required	String (UUID4)	4b8cce05-51a6-412e-89a0-d4ff6494d82c	A unique id for this event. This id should be very unique even for events from different sources and types. A UUID4-Id is commonly used.
version	optional	String	18.04	A string which specifies the version of the emitted event. The semantics of the version depend on the producer.
traceId	required	String (UUID4)	4b8cce05-51a6-412e-89a0-d4ff6494d82c	A unique id for all events in a call hierarchy. That means, if a new event is emitted (eg. caused by a user interaction) a new trace id is generated. Any consumer of the event logs traceId and spanId. When an event is processed and a new event is emitted, the trace id is taken from the incoming event and set into the new event, too. This way it is possible to correlate all these events in logs and other monitoring systems. Matching http-header: X-Trace-Id (https://medium.com/nikeengineering/hit-the-ground-running-with-distributed-tracing-core-concepts-ff5ad47c7058)
spanId	optional	String (UUID4)	4b8cce05-51a6-412e-89a0-d4ff6494d82c	A unique id for all events in a call hierarchy. That means, if a new event is emitted (eg. caused by a user interaction) the eventId of the causing event is set as SpanId, if and only if this event was created during processing of another event. Any consumer of the event logs traceId and spanId. When an event is processed and a new event is emitted, the trace id is taken from the incoming event and set into the new event, too. This way it is possible to correlate all these events in logs and other monitoring systems. Matching http-header: X-Span-Id (https://medium.com/nikeengineering/hit-the-ground-running-with-distributed-tracing-core-concepts-ff5ad47c7058)
type	required	String	sales-order-management.SALES_ORDER_PLACED	The type of the event. The type should have the owning context as prefix. Otherwise they can easy be not unique. It could for example be that, SOM emits an event called POSITION_ITEM_CANCELLED and partner-order-api, too with different meanings. This information **must** also be added at Message Meta-Data as "type" to allow SQS level filtering.
context	required	String	sales-order-management	The name of the responsible context. Normally the context, which emits this event.
eventTime	required	Date (ISO "yyyy-MM-dd'T'HH:mm:ss.SSSZ")	2018-11-28T08:25:56.523Z	The date and time, when this event occured as a iso timestamp.
sequenceNumber	optional	Long	1	The sequenceNumber states the sequence order of all events within one sequenceKey. The first event in the same partitionKey should have 0 as sequenceNumber.
sequenceKey	optional	String	4b8cce05-51a6-412e-89a0-d4ff6494d82c	A sequenceKey determines, which events share the same sequence. Ony Events, where the order is relevant have this attribute together with the sequenceNumber
test	optional	SubObject (Test)	SubObject (Test)	This block will be part of records which are only meant for Test Data to help consumers to handle them correctly
data	required	String (json)	{...}	The body contains the event data as json.

Abbildung 6: Generelle Event Struktur (OTTO MA-EC E-Commerce Innovation & -Plattform 2018)

3.2 Business Intelligence in der PEG

Das BI Team in der PEG hat die Aufgabe die Reporting Anforderungen umzusetzen. Die Abwicklung von Bezahltransaktionen auf dem Marktplatz setzt eine von der BaFin ausgegebene Bankenlizenz voraus. Die Lizenz erfordert unter anderem geeignete Maßnahmen gegen Geldwäsche und Betrug umzusetzen. Dafür werden analytische Prozesse sowie Reportings benötigt,

um Verdachtsfälle den Ermittlungsbehörden unverzüglich melden zu können, sodass hier besonders hohe Qualitätsziele gesetzt werden.

Durch die bisher positiven Erfahrungen bei der Migration von BI Systemen in die Google Cloud innerhalb der Otto Group, ist die GCP derzeit der bevorzugte Cloud Provider in den BI Abteilungen der Konzernfirmen, so auch in der PEG BI. (Westermann und Azad 2021)

Die PEG BI-Architektur basiert auf dem Data-Mesh Prinzip. Das BI Team legt die Rahmenbedingungen fest und stellt den Domainteams dafür Infrastruktur und Frameworks zur Verfügung. Anstatt eines zentralen DWH, welches allein vom BI Team betrieben und befüllt wird, werden im Data-Mesh Ansatz die Daten von einem Domainteam veredelt und für die weitere Nutzung zur Verfügung gestellt. Wie sich die Architektur auf die Schnittstellen auswirkt, wird im folgenden Abschnitt beschrieben.

3.3 Schnittstellen

3.3.1 Schnittstellen in einer BI Schichtenarchitektur

In einer klassischen Schichtenarchitektur werden Daten von Schicht zu Schicht übergeben. Beim Importieren der Daten durch ETL-Prozesse oder beim Bereitstellen von Informationen an weitere Systeme, kommen externe Schnittstellen hinzu (siehe auch Abbildung 1). Die Schichten innerhalb des Data Warehouses können durch Datenbankschema getrennt werden. Da die Schichtenarchitektur komplett vertikale Schichten beschreibt, bleibt die Schnittstelle zwischen diesen Layern eine View oder Tabelle (Leipert 2021). Vorläufig ergeben sich daraus folgende zu testende Schnittstellen, welche am Ende des Kapitels zusammengefasst werden:

- Direkte Datenbankzugriffe (SQL)
- Messaging Queues (XML oder JSON über Kafka, ActiveMq, SQS und SNS)
- Dateibasierte Im-/Exporte (XML, CSV oder JSON über GCS, S3 und FTP)

3.3.2 Schnittstellen durch Kombination aus Data Lake und Data Warehouse

Durch einen Data Lake kommen weitere interne Schnittstellen im BI System hinzu. Dazu zählt meist ein Import von Daten, welche im Data Lake aufbereitet wurden, aber auch ein Export strukturierter Daten für die Verwendung im Data Lake sind möglich (siehe auch Abbildung 3, „(b) Current two-tier architectures“). Die Liste der zu testenden Schnittstellen erweitert sich um folgende Anwendungsfälle:

- Dateibasierte Im-/Exporte im CSV, JSON, Parquet oder Avro Format über das HDFS oder Hive

3.3.3 Schnittstellen in der PEG Data-Mesh Architektur

In der Data-Mesh-Architektur werden Datenprodukte durch sogenannte Input Ports mit Rohdaten versorgt und machen veredelte Daten über Output Ports nutzbar. Technologisch gibt es keine Beschränkung durch das Data-Mesh Paradigma, sodass alle vorher genannten Schnittstellen auch hier als Output Port in Frage kommen. Datenprodukte werden durch eine Konfigurationsdatei beschrieben und durch die eigens entwickelte Plattform provisioniert und ausgerollt. Daher limitiert hier die Plattform, welche technischen Schnittstellen als Output Port zur Verfügung stehen und welche nicht. In der PEG sind folgende Schnittstellen implementiert bzw. geplant, welche in der Gesamtliste berücksichtigt werden müssen:

- Dateibasiert (JSON über GCS)
- REST API basiert (JSON über http)
- SQL basiert (SQL über BigQuery)

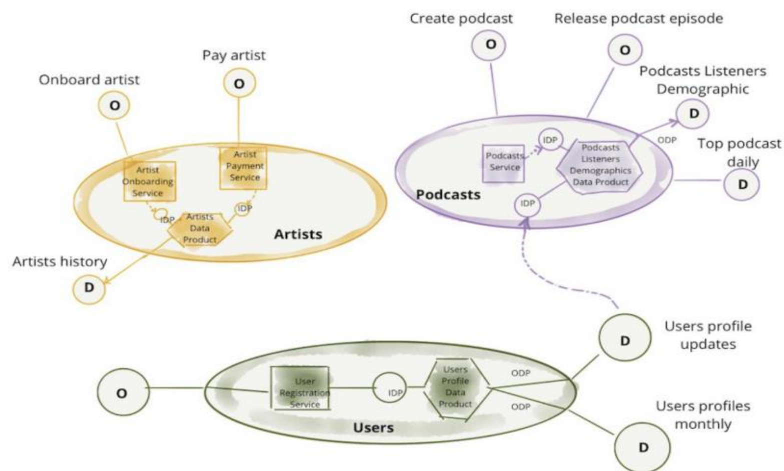


Abbildung 7: Übersicht einer Data-Mesh-Architektur (Dehghani 2020)

3.3.4 Schnittstellen gesamt

Aus den in vorherigen Abschnitten ergibt sich die Gesamtliste der zu testenden Schnittstellen:

- Datenbankzugriffe sowie SQL-fähige Cloud-Dienste (BigQuery)
- Messaging Queues (Kafka, ActiveMq, SQS und SNS)
- Dateibasierte Im-/Exporte über FTP, Cloud Storages (GCS, S3) und HDFS
- REST API basiert (http)

3.4 Bekannte/Bestehende CDCT Prozeduren

Martin Fowler beschreibt in seinem Artikel lediglich das Paradigma des CDCT. Die exakte Implementierung wird nicht vorgegeben. Dadurch haben sich in der Industrie verschiedene Lösungen entwickelt, die dem Paradigma auf ihre eigene Art und Weise folgen. In diesem Abschnitt werden Pact, die mutmaßlich bekannteste CDCT Lösung, sowie weitere Varianten durchleuchtet, die innerhalb das Otto Konzerns Anwendung finden.

3.4.1 Pact

Pact ist ein für Microservices entwickeltes Contract Testing Framework und für verschiedene Programmiersprachen verfügbar. In isolierten Komponententests (vgl. Abbildung 5) würde der Consumer den Provider mocken und der Provider einen Consumer simulieren. Pact erweitert diese Komponenten Tests um einen Contract.

Das Verfahren ist beschrieben in Abbildung 8. Wie üblich schreibt der Client seine Tests gegen den Provider Mock (Schritt 1). Über Notationen kann Pact für den Unit Test einen Provider auf einem lokalen Port mocken. Maßgeblich für Pact ist, dass es die Nachrichten des Consumers an den Mock Provider aufzeichnet und die Nachricht als auch die Antwort in einer Datei als Contract festhält (Schritt 2). In dem Code Beispiel von Listing 1, welcher die Pact/Contract Datei erzeugt, erwartet der Consumer vom Producer, dass der Producer bei einem HTTP POST mit dem HTTP Status "201" antwortet.

Diese Datei muss mit dem Provider geteilt werden (Schritt 3). Der Provider startet den Test mit dem bereitgestellten Contract. Pact wiederholt die Anfragen des Consumers und prüft das Ergebnis (Schritt 4). Das Code Listing 2 zeigt, wie der Provider das Testziel über die „TestTarget“ Annotation definiert. Durch die „State“ Annotation wird die aufgezeichnete Nachricht des Consumers an das „TestTarget“ versendet und Pact kontrolliert im Hintergrund, ob das Ergebnis der Erwartung entspricht.

Da der Consumer die Anfrage als auch das Ergebnis für den Test auf Providerseite vorgibt, ist Pact Consumer-Driven.

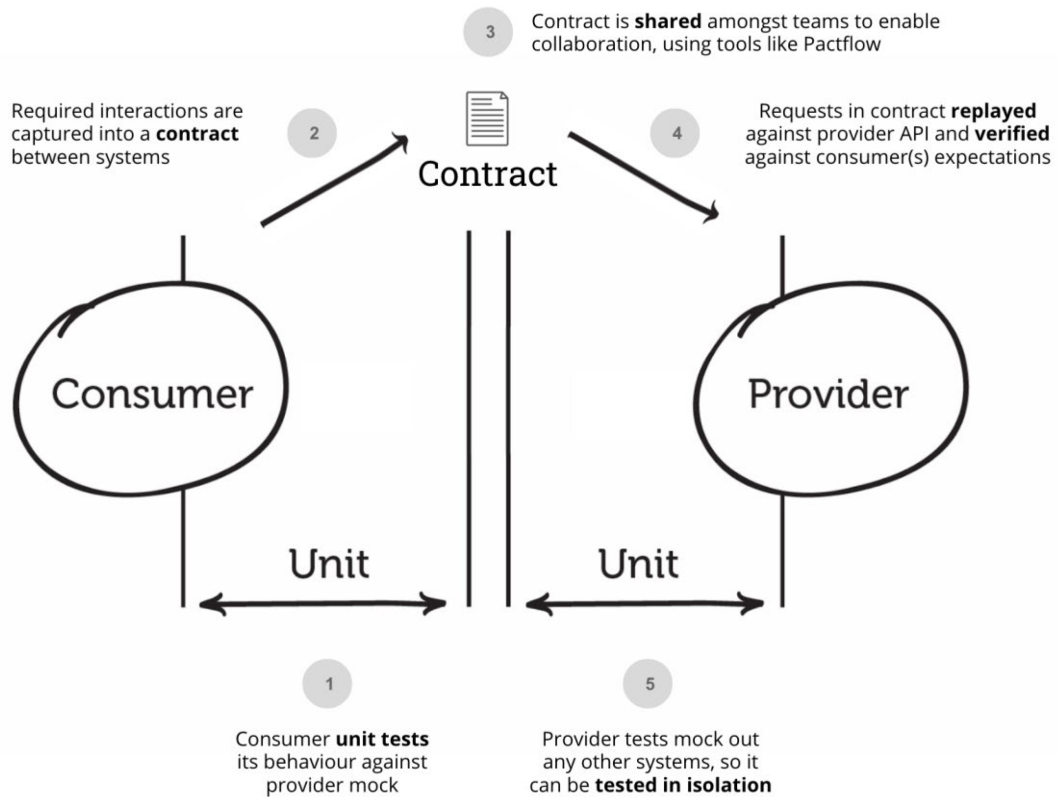


Abbildung 8: CDCT mit Pact.io (Pact Foundation 2021)

Listing 1: Voraussetzung für den Unit Test ist das Erzeugen eines Pacts, im Beispiel ein Teil des JUnit Tests. (Baeldung 2020)

```
(...)
pactDslWithProvider
  .given("test POST")
   .uponReceiving("POST REQUEST")
   .method("POST")
   .headers(headers)
   .body("{\"name\":\"Michael\"}")
   .path("/pact")
  .willRespondWith()
   .status(201)
  .toPact()
(...)
```

Listing 2: Pact JUnit Test auf Provider Seite am Beispiel einer Spring Applikation. (Baeldung 2020)

```
(...)

@TestTarget
public final Target target = new HttpTarget("http", "localhost", 8082, "/spring-rest");

private static ConfigurableWebApplicationContext application;

@BeforeClass
public static void start() {
  application = (ConfigurableWebApplicationContext) SpringApplication.run(Main.class);
}

@State("test POST")
public void testPostState() { }

(...)
```

Für den Test startet der Provider seine Applikation und teilt den Endpunkt dem Pact-Framework über die *@TestTarget* Annotation mit. Die *@State* Annotation bezieht sich dabei auf das vom Consumer definierte *.given()* in Listing 1. Auf Providerseite holt sich Pact die Erwartung aus dem Contract. Assertions im Code sind daher nicht mehr notwendig, der Methoden-Korpus kann entsprechend leer bleiben.

3.4.2 Docker CDCs

Bei den Otto PEG Produktteams kommt eine „Docker CDC“ genannte Variante zum Einsatz. In diesem Verfahren erzeugt der Consumer ein Docker Image, welches vom Producer mit Nachrichten gefüttert und ausgeführt wird. Die Nachrichten werden vom Producer über ein Docker-Volume in den Container gegeben. Das Docker Image stellt in dem Fall die Schnittstelle des Consumers dar und beinhaltet alle Abhängigkeiten, um die vom Consumer implementierte Geschäftslogik zu testen. Die Voraussetzungen sowie der Prozess sind in Abbildung 9 beschrieben.

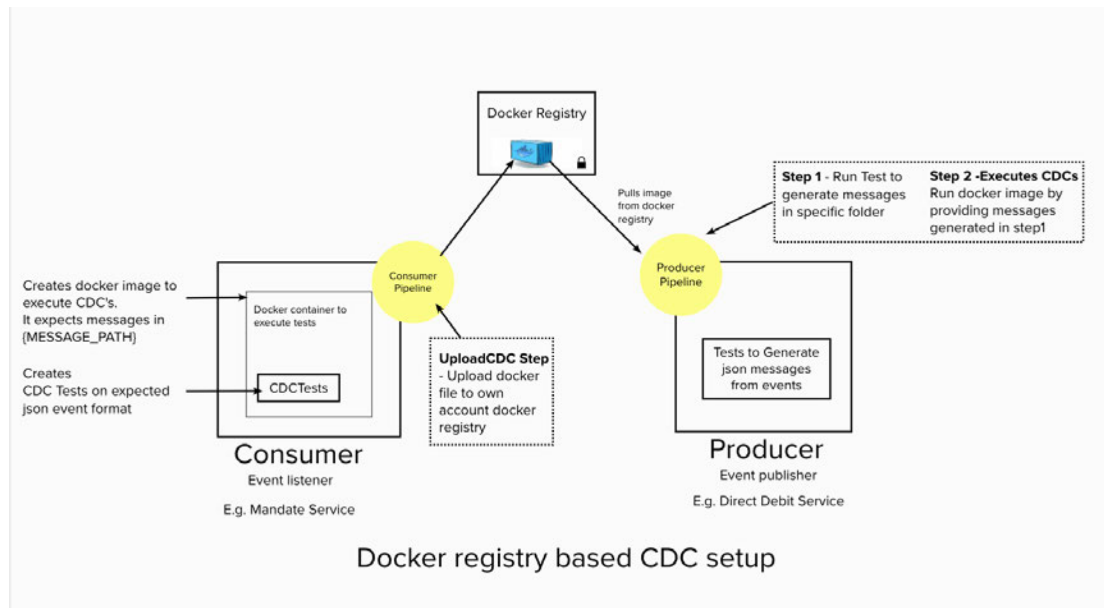


Abbildung 9: PEG Docker Registry CDC-T Ansatz (Kotwal (Thoughtworks) 2020)

Bei jeder Änderung am Code des Producers wird der Container in der CI/CD Pipeline ausgeführt. Das Kommando für ein nach Abbildung 9 gebautes Image wird in Listing 3 veranschaulicht. Über den Exit Code erfährt der Producer, ob die ggf. angepassten Nachrichten vom Consumer verarbeitet werden können.

Der Exit Code 0 sagt aus, dass die Nachrichten voraussichtlich zu keinen Problemen führen werden und der Vertrag damit immer noch gültig ist. Schlägt ein Test im Container fehl, führt das zu einem anderen Exit Code ungleich 0 und der Vertrag gilt damit als verletzt. Die CI/CD Pipeline des Producers sollte dann fehlschlagen und der neuste Commit damit nicht ausgerollt werden.

Listing 3: Beispielhafte erfolgreiche Ausführung eines vom Consumer bereitgestellten Docker CDC Images

```
$> docker run -e MESSAGES_PATH=/messages -v "$PWD/samples":/messages
bluewhale-piranha_customer-context-data:latest
$> echo $?
0
```

3.4.3 AWS Lambda

Die “AWS Lambda” getaufte CDCT-Variante findet in der Entwicklung des Otto Onlineshops Anwendung. Die Besonderheit hierbei ist, dass Producer und Consumer nicht auf zusätzliche Tools außerhalb der AWS angewiesen sind, wie es bei anderen CDCT-Varianten der Fall wäre. Zusätzlich wird die Infrastruktur mitgetestet. Der Consumer stellt den CDCT als Teil ihres Systems bereit. Dies kann eine AWS Lambda sein, die eine Test-Nachricht an den Producer sendet. Die genaue Ausgestaltung obliegt dabei den kommunizierenden Teams.

In Abbildung 10 ist der Ablauf exemplarisch dargestellt. Der Consumer (im Bild Grün) erzeugt durch seine Pipeline den Test inklusive der benötigten Infrastruktur. Die Pipeline (unten rechts) des Producers (im Bild Blau) triggert dann die AWS Lambda (grünes λ), die wiederum einen Request an den Producer sendet. Der Consumer prüft das Ergebnis und überliefert das Ergebnis der Producer Pipeline. Das Fazit von Vollerthun: „Da der Test gleichzeitig mit dem Productioncode ausgeliefert wird, hat er stets die richtige Version. Das Server-Team muss nichts mehr herunterladen und die einzige technische Abhängigkeit ist die Fähigkeit, einen HTTP-Request aus der Pipeline absetzen zu können.“ (Vollerthun 2021)

In dieser Variante ist der CDCT jedoch von funktionierender Infrastruktur des Consumers abhängig und ist damit aus meiner Sicht nicht isoliert im Sinne der in Abbildung 5 gezeigten Testpyramide.

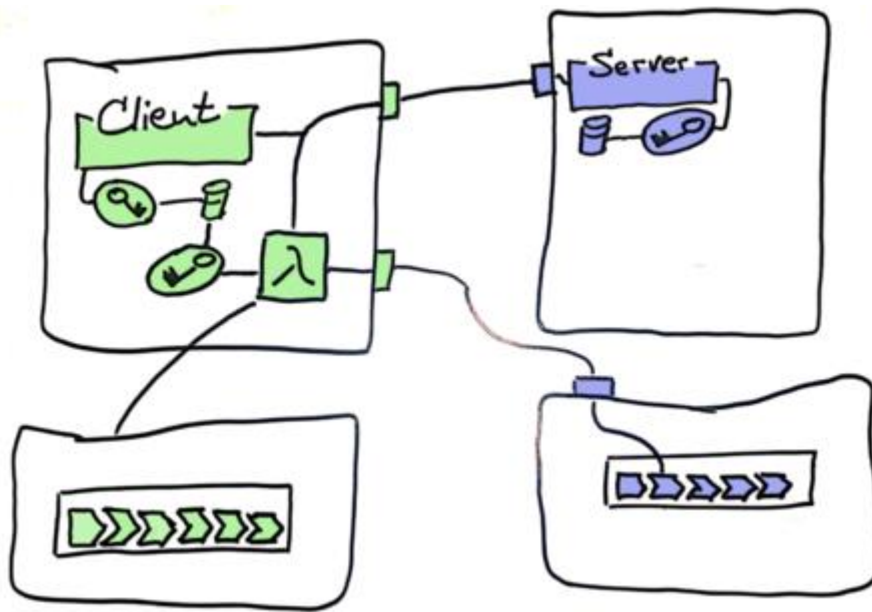


Abbildung 10: CDC-T Ansatz der Otto GmbH für die Webshop Plattform auf otto.de
(Vollerthun 2021)

4 Anforderungen

Die Rahmenbedingungen stellen die Grundlage für die Evaluierung des Consumer-Driven Contract Testing Ansatzes in der BI Abteilung der PEG dar. Im folgenden Kapitel werden die Anforderungen an das CDCT-System seitens der BI Abteilung mit Berücksichtigung bestehender Architekturvorgaben konkretisiert.

4.1 Durch PEG Architekturvorgaben / Domainteams

Die CDCTs müssen vom Provider in ihre Buildpipeline eingebunden werden. Um eine möglichst hohe Akzeptanz zu erreichen, muss der CDCT-Ansatz kompatibel zu denen sein, den die Domainteams bereits untereinander nutzen. Derzeit werden Docker CDCs von allen Teams verwendet und es befindet sich Pact in der Evaluierungsphase. Die CDCT-Lösung muss daher einer der beiden Varianten entsprechen. Darüber hinaus befinden sich die Systeme der PEG zum aktuellen Zeitpunkt in der Entwicklungsphase, sodass Consumer möglichst dem Tolerant Reader Pattern folgenden sollten.[5]

4.2 Anforderungen seitens der BI

In erster Linie hat die BI Abteilung der PEG die Anforderung, eine stabile Schnittstelle zwischen der BI und den Domainteams zu gewährleisten. Da die Domainteams wie in den Rahmenbedingungen beschrieben ausschließlich Nachrichten im JSON austauschen, soll eine Validierung der Nachricht möglich sein. Die Validierung soll dabei vor allem die Struktur der Nachrichten und dessen Datenfelder prüfen, da häufig dann Probleme auftreten, wenn sich das

[5] https://martinfowler.com/bliki/TolerantReader.html

Schema weiterentwickelt. Insbesondere werden folgende Validierungen gefordert: Reguläre Ausdrücke, Datentypen, Wertebereiche sowie minimale und maximale Länge.

Die Data-Mesh-Architektur ist flexibel in der Ausgestaltung der sogenannten Output Ports. Folgende Output Ports sind bereits implementiert oder in Zukunft geplant: Tabellen & Views in BigQuery sowie CSV Dateien, die über GCS bereitgestellt werden.

4.3 Anforderungskatalog

Die beschriebenen Anforderungen werden in folgenden Kapiteln wie folgt referenziert:

A1: Pact oder Docker CDC

Ein weiteres Framework oder eine gänzlich neue Vorgehensweise würde bei den Produktteams nicht auf Akzeptanz stoßen. Darüber hinaus sollen die Contracts innerhalb der BI sich in der Art und Weise nicht fundamental von denen der Produktteams unterscheiden.

A2: Die CDCs müssen das Tolerant Reader Pattern unterstützen

Das Tolerant Reader Pattern, wie von Martin Fowler beschrieben, sagt aus, beim Lesen bzw. dem Konsumieren von Schnittstellen so tolerant wie möglich zu sein. Beispielsweise ist ein neues Feld kein Grund in einen Fehler zu laufen. Besser ist, das neue Feld einfach zu ignorieren. Die Systeme der PEG werden sich konstant weiterentwickeln, daher sollen die erstellten Verträge immer noch gültig sein, auch wenn neue Felder in den Schnittstellen hinzukommen. In der BI ist das komplette Abziehen von Daten eine gängige Praxis. Bei Schnittstellen, die personenbezogene Daten bereitstellen, kann dies jedoch problematisch sein, da die DSGVO eine Verschlüsselung und einen Anwendungszweck für die Verarbeitung vorschreibt. Daher müssen auch restriktive Verträge möglich sein, die den Producer dazu zwingen, diese Änderungen an den Consumer zu kommunizieren.

A3: Unterstützung der Datenformate JSON und CSV sowie von Datenbanktabellen, -Views und -Abfragen

Die operativen Systeme bieten ausschließlich JSON Nachrichten als Datenquelle an. Innerhalb der BI sind bisher ausschließlich Tabellen und Views als Output Ports implementiert. Hierfür sollen SQL-Abfragen in den Verträgen festgehalten werden, sodass der Producer bei einer Datenmodelländerung prüfen kann, dass auch alle vom Consumer produktiv genutzten Abfragen weiterhin funktionieren.

CSV Im- und Exporte sind derzeit zwar nicht geplant, jedoch gilt ein Anwendungsfall in der Zukunft als wahrscheinlich. Auch hierfür sollen Consumer die Möglichkeit haben, Verträge mit den Erwartungen zu erzeugen und dem Producer zur Verfügung zu stellen.

A4: Verbindung zu den Datenquellen SNS/SQS, BigQuery und auf das Dateisystem

Für die Anbindung der Quelldaten aus den operativen Systemen steht derzeit ausschließlich SNS/SQS zur Verfügung. Innerhalb der BI werden derzeit BigQuery Tabellen, Views und Abfragen als Output Ports zur Verfügung gestellt. Für Dateibasierte Im- und Exporte muss darüber hinaus ein Dateisystem als Datenquelle möglich sein.

A5: Validierung, darunter optionale und erforderliche Datenfelder, deren Datentyp sowie inhaltlich über reguläre Ausdrücke und Wertebereiche.

Es muss möglich sein, die Daten auf erforderliche oder optionale Datenfelder prüfen zu können. Darüber hinaus unterstützt beispielsweise das JSON Format sehr flexible Datentypen wie *string* oder *number*, während Datenbankentabellen meist deutlich präziser mit Längenangaben wie *VARCHAR(255)* oder *DECIMAL(18,2)* definiert werden. Deshalb gibt es zusätzliche Anforderungen, neben der Struktur auch die Daten über reguläre Ausdrücke und Wertebereiche zu validieren.

5 Entwurf

CDCT soll im Data Warehousing und in der Business Intelligence (vgl. Abbildung 1) zum Einsatz kommen. Ersteres betrifft Schnittstellen zwischen Produktteams und der BI sowie rein innerhalb der BI. Letzteres bezieht sich auf die Schnittstelle zwischen BI und Fachbereich. Im Entwurfskapitel wird erörtert, wie die genannten Probleme gelöst und die Anforderungen umgesetzt werden sollen.

5.1 Architekturübersicht

Dieser Abschnitt zeigt die entworfene Lösung im Überblick. In den anschließenden Abschnitten werden die getroffenen Entscheidungen näher erläutert und begründet.

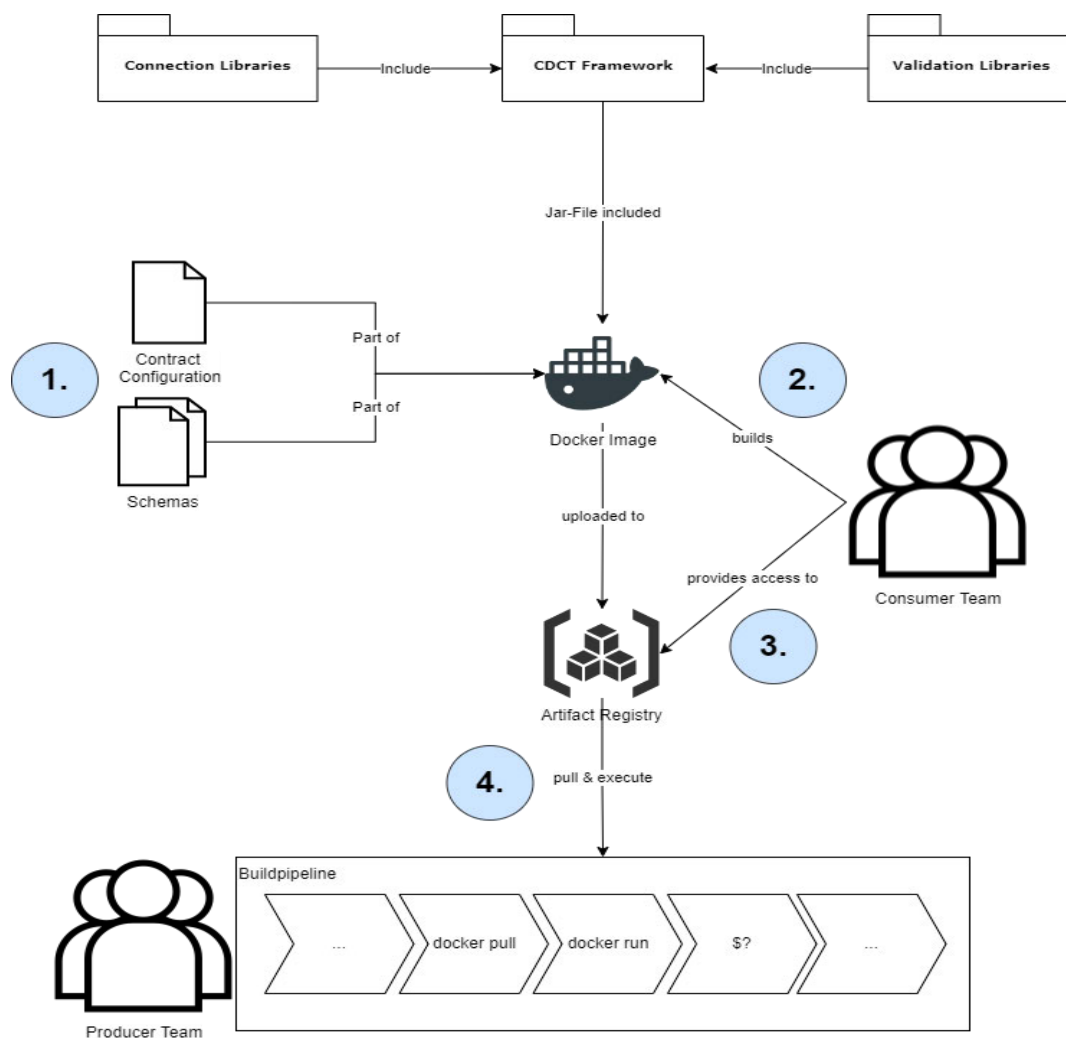


Abbildung 11: Zielbild von der Erstellung eines CDCs durch den Consumer, bis zur Nutzung auf Seite des Producers.

Abbildung 11 beschreibt das Zielbild und die notwendigen Schritte 1 bis 4 (blau umkreist), von der Erstellung des CDCs durch den Consumer, bis zur Einbindung in die CI-Pipeline des Producers. Das CDCT-Framework soll dabei alle Validierungen und Datenverbindungen generalisieren. Der Code sowie die Konfiguration werden in GitHub verwaltet. Als Docker Registry kommt GitHub Packages und als CI/CD Lösung GitHub Actions zum Einsatz.

Zuerst muss die Schnittstelle durch den Consumer beschrieben werden (Schritt 1), dafür ist eine Schema-Definition (JSON-Schema, XSD, CSVS, …) sowie eine Konfiguration für das Docker Image nötig (siehe Tabelle 1).

Tabelle 1: Die nötigen Parameter für die Konfiguration eines CDCT Docker Images.

Parameter	***Beispiel***	***Verwendungszweck***
CONSUMER_TEAM	Team1	Teil des Docker Imagetags
PRODUCER_TEAM	Team2	Teil des Docker Imagetags
API_NAME	Sample-API	Teil des Docker Imagetags
CONSUMER_TYPE	FILES	Parameter des CDCT-Frameworks
CONTRACT_ADDITIONS	--ignoreUnknownMessages	Parameter die beim Aufruf des CDCT-Frameworks angehängt werden

Im zweiten Schritt triggert das Consumer Team die Buildpipeline für den Bau des Docker Images. Die in Tabelle 1 genannten Parameter werden dabei für eine einheitliche Namenskonvention des Imagetags herangezogen. Das Image heißt dann *<CONSUMER_TEAM>-<PRODCUER_TEAM>_<API_NAME>*, also *team1-team2_sample-api*. Der Upload in die Docker Registry erfolgt automatisch.

Damit das Producer Team das CDCT Image verwenden kann, muss der Consumer das GitHub Repository des Producers den Zugriff auf das erstellte Image erlauben.

Im vierten Schritt integriert das Producer Team die Ausführung des CDC Image in dessen Buildpipeline.

5.2 Entwurf des CDCT-Frameworks

Das CDCT-Framework soll eine einheitliche Schnittstelle für verschiedene Validierungsbibliotheken und Datenverbindungen schaffen. In Abbildung 12 wird diese Abstraktion beschrieben. Für jede Validierungs- und Verbindungsbibliothek muss das *CDCTValidator* und *GenericMessageConsumer* Interface implementiert werden. Der *CDCTWorkflow* ist die Main-Class, sie bekommt die Kommandozeilenparameter und übernimmt die Orchestrierung.

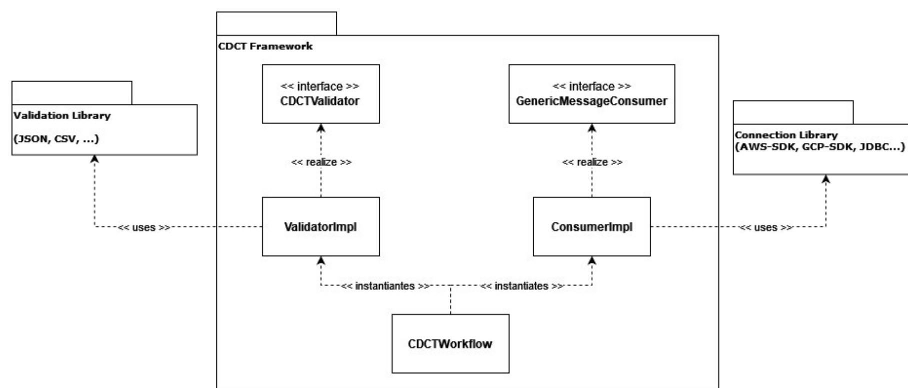


Abbildung 12: Vereinfachtes UML Klassendiagramm des CDCT-Frameworks

Das Zusammenspiel zwischen dem *CDCTWorflow* sowie einer Implementierung des *CDCTValidator* und *GenericMessageConsumer* Interfaces wird im folgenden Sequenzdiagramm auf Abbildung 13 entwurfsweise veranschaulicht.

Der *CDCTWorkflow* verarbeitet die Parameter, dadurch ist der Klasse bekannt, welche Implementierungen zu instanziieren sind (Aufrufe 1 und 2). Der *CDCTValidatorImpl* bekommt dann den *GenericMessageConsumerImpl* (Schritt 3) und die als Parameter angegebenen Schema (Schritt 4) durch den *CDCTWorkflow* übergeben. Durch den einfachen *validate()* Aufruf in Schritt 5 soll eine klare Abgrenzung der Verantwortlichkeit geschaffen werden. Für den *CDCT-Workflow* ist an dieser Stelle nur entscheidend, ob die Validierung erfolgreich ist. Wie die *CDCTValidatorImpl* mit den Nachrichten des *GenericMessageConsumerImpl* arbeitet, ist in dessen Implementierung gekapselt (Schritt 6 und 7).

Schlägt eine Validierung fehl, so wird eine *CDCTException* geworfen, die den Exit Status des Programms beeinflusst. Wird keine *CDCTException* geworfen, wird der Workflow mit dem Exit Code 0 beendet.

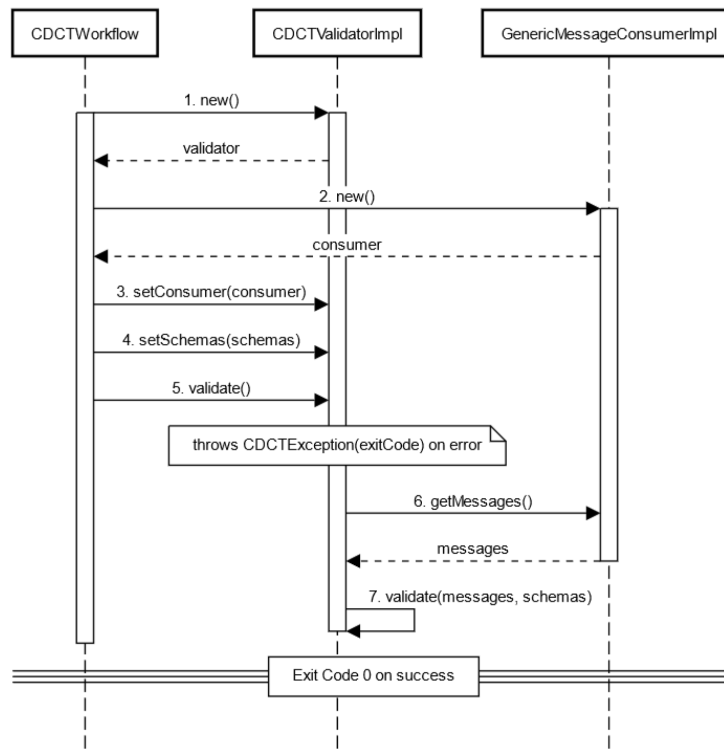


Abbildung 13: UML Sequenzdiagram des CDCT-Frameworks

5.3 Grundlage für den Entwurf

CDCT ist ein Ansatz, um Schnittstellen auf beiden Seiten isoliert testbar zu machen. Rückblickend auf die im Grundlagenkapitel gezeigte Abbildung 4 liegen die Testpunkte 2 und 4, dabei speziell das „Extract" der ETL-Prozesse, sowie Testpunkt 6, dem Datenzugriff, im Fokus.

Beim Data Warehousing ist der „Extract" Teil des ETL-Prozesses der Zugriff auf die Schnittstelle des Producers. Es kann auch mehrere „Extracts" geben, wenn innerhalb eines Prozesses zwei oder mehr Quellen zusammengeführt werden sollen, welche von unterschiedlichen

Providern bereitgestellt gestellt werden. Der Fall in Abbildung 14 würde folglich zu drei Contracts führen.

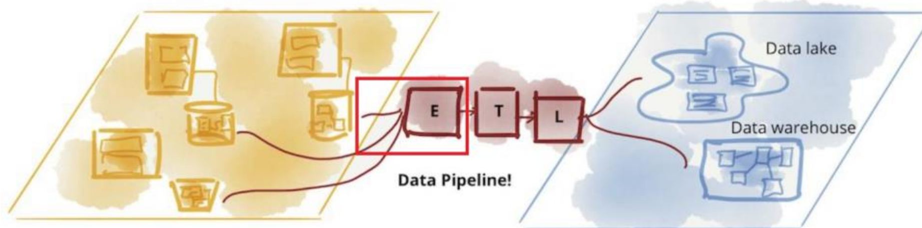


Abbildung 14: ETL Datapipeline visualisiert - Getestet werden soll die Verbindung und das Format des Inputs, nicht aber der Transform Teil der ETL-Prozesse. (Dehghani 2020)

In der Annahme, dass der „Extract“ Teil des Prozesses auf jeden Fall eine bestimmte Datenlieferung durch den Contract sicherstellt, können „Transform“ sowie „Load“ Teile mit synthetischen Daten und isolierten Unit Tests abgedeckt werden.

Auf der Business Intelligence Seite sind Reports gefordert. In der PEG wird Microsoft Power BI mit BigQuery Tabellen/Views/Abfragen als Datenquelle verwendet. Die SQL-Abfragen bilden dabei die Schnittstelle zwischen BI Abteilung und dem Fachbereich. Die Korrektheit dieser Abfragen gilt es sicherzustellen, sodass jede kritische Abfrage über einen Contract abgesichert werden können muss.

5.4 Auswahl der CDCT Prozedur

Durch den Einsatz im Unternehmen mit bestehenden Strukturen limitiert die Anforderung A1 die möglichen Techniken. Dennoch erscheinen beide Varianten ausreichend flexibel für den breiten Einsatz außerhalb des Otto Konzerns.

Pact ist frei verfügbar, wird seit 2013/2014 entwickelt und bietet Unterstützung für derzeit beliebte Programmiersprachen. Darunter sind auch in der BI genutzte Sprachen wie Python,

Java und Scala. Für die in der PEG eingesetzten Sprachen Python, Java, Go und JavaScript existieren Implementierungen von Pact. Darüber hinaus ist Pact bzw. dessen Spezifikation quelloffen und eine Unterstützung für weitere Programmiersprachen damit möglich. Um Pact CDCs zu testen, muss die jeweilige Pact-Bibliothek eingebunden und die Verteilung der CDCs organisiert werden. Pact beschränkt sich derzeit auf http und Nachrichtenbasierte Interaktionen.

Docker ist in der IT-Welt weit verbreitet. Docker ist eine Linux-Container Lösung, die CDCs sind daher theoretisch an keine Sprache und keine bestimmte Interaktion gebunden. Die beschriebene Variante schreibt zwar vor, Dateien in den Container zu mounten, eine Erweiterung für http, Socket oder nachrichtenbasierte Interaktion wäre problemlos möglich, ohne bestehende CDCs ablösen zu müssen.

Schlussendlich hat sich die Docker Variante gegenüber Pact durchgesetzt. Um CDCs über Pact einzuführen, benötigen der Producer als auch der Consumer Pact, während Docker bereits unabhängig von CDCs im Tech-Stack vorhanden ist. Der ausschlaggebende Punkt allerdings war die Erfahrung von Inkompatibilitäten, die zwischen zwei Teams mit derselben Pact Version gesammelt wurden. In einer beispielhaften Implementierung waren einige *Matcher* der pact-jvm[6] nicht mit denen der pact-python[7] Implementierung kompatibel. Eine Übersicht, welche Features von welcher Implementierung unterstützt werden, findet sich in der Dokumentation von Pact[8].

5.5 Auswahl der Programmiersprache

Die Docker Variante, die Nachrichten über ein Volume in den Container zu geben, macht die Implementierung von der Programmiersprache unabhängig. Für die Datenintegration wird Google DataFlow verwendet, welches die Ausführung von Apache Beam Pipelines in der GCP

[6] https://github.com/pact-foundation/pact-jvm

[7] https://github.com/pact-foundation/pact-python

[8] https://docs.pact.io/roadmap/feature_support

ermöglicht. Das BI Team der PEG hat sich in der Vergangenheit für das Java SDK entschieden, da die meisten Entwickler diese Sprache beherrschen. Daher bietet sich Java als Programmiersprache für das Framework innerhalb des CDC Containers an.

5.6 Auswahl der Validierungsbibliotheken

Für die Validierung der verschiedenen Datenformate sollen im besten Falle quelloffene und aktiv gepflegte Bibliotheken zum Einsatz kommen. Als aktiv werden Projekte bezeichnet, dessen letztes Release oder letzter Commit im Master Branch nicht älter als sechs Monate ist.

5.6.1 CSV-Schema Validierung

Der Im- und Export von CSV Dateien ist eine typische Anforderung in der BI. *The National Archives UK* pflegt eine CSV Schema Definition[9] und dazugehörige Validierungsbibliothek[10] auf GitHub und erfüllt damit die Bedingung der Quelloffenheit.

Diese Bibliothek wurde in der Vergangenheit bereits erfolgreich in der Otto BI eingesetzt und soll die Grundlage für die CSV Unterstützung der Anforderung A3 bilden.

Die Validierungsbibliothek als auch die CSV Schema Language (folgend CSVSL) befinden sich im aktiven Zustand. Gewählt wird die Version 1.1 der CSVSL, da Version 1.2 vom 24. Februar 2022 noch nicht finalisiert wurde.

Für die Erfüllung der Anforderung A2 müssen Spalten als optional sowie die gesamte Anzahl der Spalten flexibel oder fix gesetzt werden können. Nach CSVSL muss für jede Spalte eine Definition gegeben werden, unbekannte Felder führen zu Validierungsfehler. A2 wird für CSV daher nicht unterstützt.

---

[9] http://digital-preservation.github.io/csv-schema/

[10] https://github.com/digital-preservation/csv-validator

Im folgenden Listing 4 sind die Validierungen für die Anforderung A5 sichtbar. Das *ID* Feld wird auf den Datentyp geprüft, *RANGE_VALUE* auf den Wertebereich zwischen 2000 und 2999 und *DECIMAL_VALUE* muss „*-0*“ gefolgt von einem Komma und zwei Zahlen entsprechen. Das *@optional* sagt, dass ein leeres Feld erlaubt ist, das Feld darf aber nicht in der CSV Datei fehlen.

Listing 4: Beispiel für eine CSVS-Definition für die Validierung von Wertebreichen, Datentypen und regulären Ausdrücken.

```
version 1.0
@separator ';'
ID: positiveInteger
RANGE_VALUE: range(2000,2999)
DECIMAL_VALUE: regex("-0,\d{2}") @optional
```

Listing 5: Eine zu Listing 4 passende CSV-Datei.

```
ID;RANGE_VALUE;DECIMAL_VALUE;OPTIONAL_FIELD
123;2021;-0,55;
```

5.6.2 JSON Schema Validierung

Eine Recherche zu möglichen Bibliotheken einer JSON-Validierung stellt das JSON-Schema[11] eine Möglichkeit dar, JSON-Dokumente zu validieren und zu beschreiben.

[11] https://json-schema.org/

Java Bibliotheken, die JSON-Dokumente gegen ein definiertes Schema validieren können, sind die Projekte everit-org/json-schema[12] und java-json-tools/json-schema-validator[13]. Davon hat nur Ersteres ein Commit in jüngster Vergangenheit und unterstütz Draft 7[14] für reguläre Ausdrücke, welche mit Anforderung A5 gestellt werden.

JSON-Schema Draft 7 unterstützt das mit Anforderung A2 geforderte Tolerant Reader Pattern über die *"additionalProperties": false* Direktive, sodass unbekannte Felder in dem JSON-Dokument nicht zu Validierungsfehlern führen.

Die *type* Beschreibung erlaubt nur die JSON-Datentypen. Mit Hilfe von *format* und *regex* lassen sich die Inhalte jedoch ausreichend spezifizieren. Darüber hinaus decken *range*, *oneOf* und *required* alle Aspekte der Anforderungen A2.

Ein Beispiel findet sich in Listing 6 mit zugehöriger Nachricht in Listing 7.

[12] https://github.com/everit-org/json-schema

[13] https://github.com/java-json-tools/json-schema-validator

[14] https://json-schema.org/draft-07/json-schema-release-notes.html

Listing 6: Beispiel für eine JSON-Schemadefinition für die Validierung.

```
{
    "$schema": "http://json-schema.org/draft/7/schema",
    "$id": "http://example.com/schema.json",
    "type": "object",
    "title": "Example Schema",
    "required": [ "id", "range_field"],
    "additionalProperties": true,
    "properties": {
        "id": {
            "type": "string",
            "format": "uuid"
        },
        "range_field": {
            "type": "number",
            "minimum": 2000,
            "maximum": 2999
        },
         "optional_field": {
            "type": "number",
            "regex": "-0,\\d{2}"
        }
    }
}
```

Listing 7: Ein zu Listing 6 passendes JSON Dokument.

```
{
    "id": "bc3646dc-9819-4311-a569-9e3cccb16e70",
    "range_field": 2001,
    "optional_field": -0.55,
    "unkown_field": "valid with additionalProperties: true"
}
```

5.6.3 SQL Query Schema Validierung

Die Ergebnis- und Schemavalidierung von SQL-Abfragen muss für Reports (vgl. Testpunkt 6 Abbildung 4) und für das Weiterreichen von Daten zwischen Teams innerhalb der BI-Abteilung (vgl. Testpunkt 4 Abbildung 4) möglich sein. Die im Qualitätssicherungskapitel 2.3.1 beispielhaften „Integrity Tests“ der MicroStrategy Plattform lassen sich nur innerhalb dieser Plattform nutzen und das von der PEG genutzte Power BI bietet kein vergleichbares Tool. Darüber hinaus wären mit den plattformspezifischen Tools maximal die Testpunkte 6 und 7, also die Interaktion mit der BI-Plattform testbar, nicht jedoch für Anforderungen aus dem Data Warehousing, bei der Datenprodukte bzw. Tabellen/Views innerhalb der BI Abteilung ausgetauscht werden.

Bei der Recherche nach plattformunabhängigen Bibliotheken für Schema/Query Validation finden sich diverse Tools, u.a. das "Schema Validation Tool" (SVT), welches wünschenswerte Eigenschaften wie Integrität und Erreichbarkeit prüfen kann. Nach Änderungen am Datenmodell kann das SVT dadurch Fehler im eigenen Schema finden. (Ernest Teniente, Carles Farre, Toni Urpi, Carlos Beltran, and David Ganan (University of Catalunya, Spain)

Ähnliche Möglichkeiten, Schema zu vergleichen und zu validieren, bieten auch Tools wie ApexSQL[15] und SQL-Compare von Red Gate[16]. Diese Art Tests werden von dem Team durchgeführt, das für das Schema verantwortlich ist, folglich auf der Provider Seite. Diese Tests können jedoch nicht garantieren, ob ein ETL-Prozess oder Bericht eines außenstehenden Teams nach einer Änderung immer noch fehlerlos die Daten abziehen kann. Unter diesen Umständen sind diese Tools auf der Testpyramide von Abbildung 5 einem isolierten Komponententest zuzuordnen.

Für diese Arbeit wird daher eine eigene, schlanke Bibliothek entworfen, die analog den bisher ausgewählten Bibliotheken den Anforderungen gerecht werden soll und im folgenden Kapitel beschrieben wird.

---

[15] http://solutioncenter.apexsql.com/compare-sql-server-database-schemas-automatically/

[16] https://www.red-gate.com/products/sql-development/sql-compare/

5.7 Entwurf einer Validierungsbibliothek für SQL-Abfragen

Bei der Validierung von CSV, JSON und XML-Dateien wird vorher ein Schema definiert. Die Validierung von SQL-Abfragen, die ein ETL-Prozess oder Bericht auf das DWH durchführen würde, soll daher analog der vorher beschrieben Bibliotheken in einem Schema festgehalten werden. Dieses Schema für eine SQL-Abfrage ist abzugrenzen von dem Begriff „Schema“ innerhalb einer Datenbank, die ganze Tabellen/Views usw. enthalten und mit der DDL beschrieben werden.

5.7.1 SQL-Schema Definition (SQSD)

Voraussetzung für ein breiten Einsatz ist die Unabhängigkeit zu einer bestimmten Datenbankplattform und wird durch die Anforderung A4 (verschiedene Datenquellen) vorausgesetzt. Für die Anforderung A5 (Validierung) sollen im Schema folgende Definitionen möglich sein:

- Auszuführende SQL-Abfrage
- Die im Ergebnis erwarteten Spalten mit Datentyp
- Keine Einschränkung auf eigene Interpretation von Datentypen wie „String“ im JSON Schema, es müssen alle datenbankspezifischen Typen beschrieben werden können
- Feldlänge, Präzision und Skala für variable Datentypen wie *Char* und *Decimal*
- Prüfung auf Nullable
- Reguläre Ausdrücke für die Validierung

Diese Definition soll in einer UTF-8 kodierten Textdatei mit der Dateiendung *.sqsd* gespeichert werden. Die Definition des SQSD wird ein JSON-Objekt, dadurch kann die SQSD mithilfe eines JSON-Schemas syntaktisch geprüft werden.

Tabelle 2: Attribute des SQSD JSON-Objekts

JSON Pfad	***JSON Datentyp***	***Erforderlich***
$.$version	String	Nein
$.query	String	Ja
$.columns	Array(Column)	Ja

Tabelle 3: Attribute des SQSD Spalten-JSON-Objekts

JSON Pfad	***JSON Datentyp***	***Erforderlich***
$.columns[*].name	String	Ja
$.columns[*].type	String	Ja
$.columns[*].length	Integer	Nein
$.columns[*].scale	Integer	Nein
$.columns[*].precision	Integer	Nein
$.columns[*].nullable	Boolean	Nein
$.columns[*].regex	String	Nein

Eine exemplarische Definition wird folgend mit Listing 8 gezeigt, welche alle mit Datenbanken möglichen Aspekte der Anforderung A5 prüfen kann. Hervorzuheben ist, dass unter *type* ein von der Datenbankplattform abhängiger Datentyp spezifiziert werden kann.

Beziehend auf die Anforderung A2, dem Tolerant Reader Pattern, steht die Unterstützung eingeschränkt zur Verfügung. Die in Listing 8 eingetragene SQL-Abfrage erwartet genau die genannten Spalten. Das Ändern der Tabelle *table_name* wäre möglich, solange nicht die geforderten Spalten geändert werden. Enthält die SQL-Abfrage jedoch das *-Literal, sodass alle Felder zurückgegeben werden, würde eine Änderung an der Tabelle zu einem Fehler in der Validierung führen.

Listing 8: Konzept einer „SQL Query Definition“ für die Validierung von Datenbankabfragen

```
{
  "$version": "1"
  "query": "select text_column, decimal_column from table_name",
  "columns": [
    {
      "name": " text_column",
      "type": "VARCHAR",
      "length": 255,
      "regex": "[a-z]+"
    }, {
      "name": "decimal_column",
      "type": "DECIMAL",
      "precision": 18,
      "scale": 4,
      "nullable": false,
      "regex": "[0-9]+(\\.[0-9]{0,4})?"
    }
  ]
}
```

5.8 Zusammenfassung der Validierungsmöglichkeiten

Die in Kapitel 4.3 genannten Anforderungen an verschiedene Datenquellen, Formate und Validierungsmöglichkeiten, ergeben sich aus den konkreten Anwendungsfällen innerhalb der PEG.

Die folgende Tabelle fasst den Funktionsumfang der Validierung in Bezug auf die Datenquelle und des Formates zusammen. Vorwegnehmend von Kapitel 8.1 – Ausblick, werden auch weitere BI typische Quellen dargestellt.

Tabelle 4: Zusammenfassung der Validierungsmöglichkeiten in Bezug auf die Anforderungen A2 und A5.

Datentransport	***Format***	***Schema***	***A2***	***A5***	***MVP***
File	JSON	JSON-Schema	Ja	Ja	Ja
SQS/SNS	JSON	JSON-Schema	Ja	Ja	Ja
BigQuery	BigQuery	SQSD	Teilw.	Ja	Ja
JDBC	ResultSet	SQSD	Teilw.	Ja	Nein
File	CSV	CSVD	Nein	Ja	Nein
File	XML	XSD	Ja	Ja	Nein
Kafka	JSON	JSON-Schema	Ja	Ja	Nein

6 Implementierung

In diesem Kapitel werden Kernelemente des CDCT-Frameworks sowie der SQL Query Schema Validierung gezeigt. Das Kapitel beginnt mit dem Aufbau sowie der Anwendung des Frameworks und mündet in detaillierten Code Beispielen, die auf das wesentliche reduziert sind, um die Weiterentwicklung des Projektes zu ermöglichen.

6.1 Sourcecode Repository

Wie im Kapitel 5.1 genannt, wird der Quellcode in GitHub verwaltet. Als Build Tool wird Gradle[17] eingesetzt und das Projekt besitzt eine für Java typische Ordnerstruktur. Listing 9 enthält die Kernelemente des Projektes.

Darüber hinaus werden folgende Erweiterungen bzw. Plugins verwendet, deren Konfigurationen sich in Listing 9 wiederfinden.

CI/CD

GitHub Actions wird für CI/CD verwendet. Dessen Konfiguration sind unter *.github/workflows* abgelegt. Das *cdc-template.yml* kapselt dabei alle Schritte, die für die Erstellung eines CDCs nötig sind. Details zur Verwendung werden in Kapitel 6.2 erläutert.

Compliance

Die Compliance stellt Anforderungen an alle Softwareprojekte des Unternehmens. Für dieses Softwareprojekt bedeutet dies:

1. Regelmäßige Funktionsprüfung der Software
2. Regelmäßige Überprüfung auf Schwachstellen

[17] https://gradle.org/

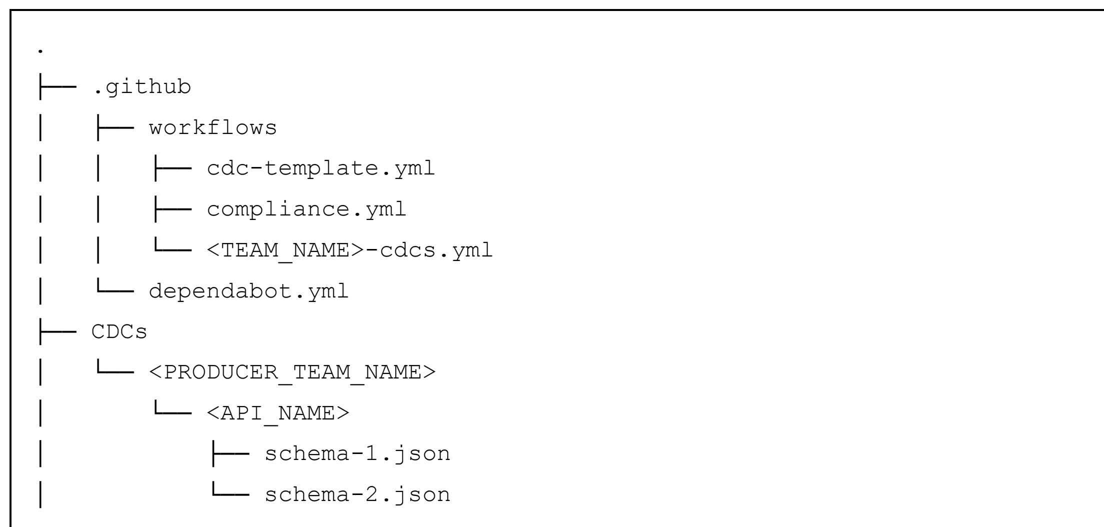
3. Es dürfen nur aktive Open Source Projekte mit unproblematischen Lizenzen verwendet werden.

Die für die Compliance nötigen Schritte 1 bis 3 sind in der *.github/workflows/compliance.yml* definiert, welche bei jedem Build sowie täglich ausgeführt werden.

Die Tests werden über Gradle ausgeführt. Damit die Software stets aktuell gehalten wird und somit Schwachstellen durch veraltete Abhängigkeiten vermieden werden, ist der GitHub *Dependabot* mit der gleichnamigen YAML konfiguriert[18]. Dieser prüft regelmäßig auf neue Softwareversionen und erstellt automatisch Pull-Requests.

Für konkrete Schwachstellenprüfung kommt das Gradle Plugin *OWASP Dependency Check*[19] zum Einsatz. Die Prüfung auf problematische Lizenzen erfolgt durch das Gradle Plugin *Gradle License Report*[20]. Die Konfiguration der genannten Gradle Plugins befinden sich in der *build.gradle* sowie innerhalb des *config* Ordners.

Listing 9: Ordnerstruktur des Projektes



```
.
├── .github
│   ├── workflows
│   │   ├── cdc-template.yml
│   │   ├── compliance.yml
│   │   └── <TEAM_NAME>-cdcs.yml
│   └── dependabot.yml
├── CDCs
│   └── <PRODUCER_TEAM_NAME>
│       └── <API_NAME>
│           ├── schema-1.json
│           └── schema-2.json
```

[18] https://github.com/dependabot

[19] https://jeremylong.github.io/DependencyCheck/dependency-check-gradle/index.html

[20] https://github.com/jk1/Gradle-License-Report

```
├── config
│   ├── allowed-licenses.json
│   └── owasp-suppression.xml
├── gradle
│   └── wrapper
│       ├── gradle-wrapper.jar
│       └── gradle-wrapper.properties
├── src
│   ├── main
│   │   ├── java
│   │   └── resources
│   └── test
│       ├── java
│       └── resources
├── .gitignore
├── Dockerfile
├── README.md
├── build.gradle
├── cdctest.sh
├── gradlew
├── gradlew.bat
└── settings.gradle
```

6.2 Etablieren eines neuen CDCs

Das in der Architekturübersicht entworfene Verfahren zeigt in Abbildung 11 die nötigen Schritte und soll in diesem Abschnitt konkretisiert werden.

Abbildung 11 Schritt 1 - Konfiguration: Die CDCs sind in dieser Implementierung Teil des Code Repository. Damit die CI/CD-Pipeline das Schema findet, ist eine Namenskonvention einzuhalten. Entsprechend Listing 9 wird innerhalb des *CDCs* Ordners je Team und darin je

API ein Verzeichnis angelegt. Innerhalb des API-Verzeichnisses können mehre Schemas des gleichen Typs liegen.

Abbildung 11 Schritt 2 - Build: Für jedes Producer Team wird eine GitHub Action angelegt (siehe Listing 9 in *.github/workflows/*), damit nicht bei jeder Änderung alle CDCs neu gebaut werden. Die mit Listing 10 gezeigte Vorlage für eine CDC GitHub Action kann entweder manuell (Keyword *workflow_dispatch*) oder durch gepushte Commits, mit Änderungen innerhalb des Producer Team Ordners, (Keyword *paths:*) angestoßen werden.

Listing 10: GitHub Action für die Erstellung eines CDC in Verwendung des CDC Templates (*cdc-templates.yml*)

```
name: "Build <PRODUCER_TEAM> CDCs"
on:
  push:
    paths:
      - CDCs/<PRODUCER_TEAM>/**
  workflow_dispatch:

jobs:
  <PRODUCER_TEAM>-cdc-<API_NAME>:
    uses: Org/repo/.github/workflows/cdc-template.yml@master
    with:
      consumer_team: "<CONSUMER_TEAM>"
      producer_team: "<PRODUCER_TEAM>"
      api_name: "<API_NAME>"
      consumer_type: "<CONSUMER_TYPE>"
      contract_additions: "<CONTRACT_ADDITIONS>"
```

Abbildung 11 Schritt 3 – Rechte: Nach dem ersten Build erscheint das Image in GitHub Packages[21] innerhalb der GitHub-Organisation[22]. Der Test soll schlussendlich in der (GitHub Actions) Buildpipeline des Producers ausgeführt werden. Damit das gelingt, muss dem GitHub Repository des Producers lesender Zugriff gewährt werden. Dies erfolgt manuell über die Weboberfläche von GitHub.

Abbildung 11 Schritt 4 – Ausführung: Wie der Test in GitHub Actions ausgeführt werden kann, wird in Listing 11 gezeigt. Hat der Consumer die Rechte erteilt, benötigt die Pipeline des Producers die Rechte ein Authentifizierungstoken zu erstellen (*id-token: write*) und auf Packages lesend (*packages-read*) zuzugreifen.

Für den Login in die Container Registry selbst kann die *docker/login-action*[23] verwendet werden. Die Credentials (*github.actor* und *secrets.GITHUB_TOKEN*) werden von GitHub Actions automatisch bereitgestellt. Vorausgesetzt wird, dass der Producer die Nachrichten, die er dem Consumer schicken würde, in */samples* abgelegt hat. Schlussendlich erfolgt die Ausführung über ein einfaches *docker run* Kommando.

[21] https://docs.github.com/en/packages/learn-github-packages/introduction-to-github-packages

[22] https://docs.github.com/en/organizations/collaborating-with-groups-in-organizations/about-organizations

[23] https://github.com/docker/login-action

Listing 11: Beispielhafte Ausführung eines CDCs in der GitHub Action Buildpipeline des Producers.

```
name: CDC Execution example

on:
  push:

jobs:
  main-build:
    runs-on: ubuntu-latest
    permissions:
      id-token: write
      packages: read

[…]

      - name: Login to GHCR of Consumer Team
        id: login
        uses: docker/login-action@v1
        with:
          registry: ghcr.io
          username: ${{ github.actor }}
          password: ${{ secrets.GITHUB_TOKEN }}

      - name: Pull CDC Docker Images
        run: docker pull ghcr.io/org/consumer-producer_api:latest

      - name: Run CDC
        run: docker run -e MESSAGES_PATH=/messages -v "$PWD/sam-
ples":/messages consumer-producer_api:latest
```

6.3 Implementierung einer Datenverbindung

In Abbildung 12 und Abbildung 13 des Entwurfskapitels sind die Interfaces und ihr Zusammenspiel dokumentiert. Die Daten für den *CDCTValidator* werden über den *GenericMessageConsumer* bereitgestellt.

Folgendes Vorwissen ist für das Verständnis einer Implementierung eines *GenericMessagesConsumers* wichtig: Es ist zu beachten, dass die Validierungsbibliotheken nicht ohne weiteres jeden Input unterstützen. Beispielsweise erlaubt der CSV Validator nur Pfade zu Dateien. Die JSON-Schema Bibliothek benötigt hingegen den Input als *JSONObject*. Da jedoch beim bereits etablierten Verfahren (siehe Abbildung 9) die JSON-Nachrichten als Datei in den CDC-Container gelegt werden, wurde die *CDCTValidator* Implementierung *JsonValidator* dahingehend erweitert, dass diese auch mit einer Liste von Pfaden arbeiten kann (siehe Abbildung 15).

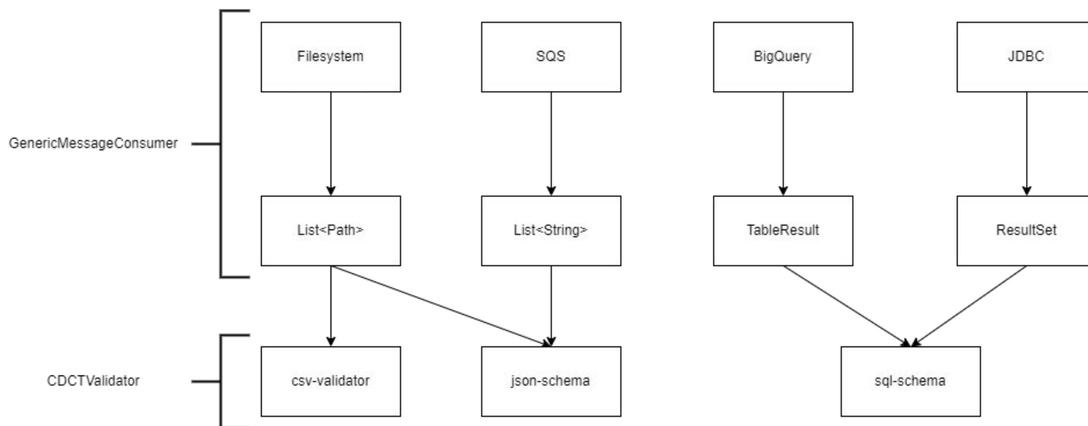


Abbildung 15: Nicht alle Datenformate können von jedem Validator verarbeitet werden.

Damit eine Implementierung des *CDCTValidator* den Input unterscheiden kann, muss diese wissen, welcher Typ in der Liste übergeben wird. Aufgrund der Type Erasure[24] bei Generics ist das jedoch nicht möglich. Deswegen wurde das ursprünglich angedachte einfach aussehende Interface *MessageConsumer* (Listing 12) in einer abstrakten Klasse verpackt (Listing 13), bei der der Typ im Konstruktor übergeben wird und somit später für einen Vergleich abgerufen werden kann. Die in Listing 12 genannte Klasse *GenericMessageList* funktioniert analog und erweitert die Java Standard *ArrayList*.

Listing 12: Das *MessageConsumer* Interface

```
public interface MessageConsumer<T> {
    GenericMessageList<T> getMessages();
}
```

[24] https://docs.oracle.com/javase/tutorial/java/generics/erasure.html

Listing 13: Die *GenericMessageConsumer* Klasse

```
public abstract class GenericMessageConsumer<T> implements MessageConsumer<T> {

    private final Class<T> genericType;

    public GenericMessageConsumer(Class<T> type) {
        this.genericType = type;
    }

    public Class<T> getGenericType() {
        return genericType;
    }

}
```

Im folgenden Listing 14 wird die Implementierung der *SqsConsumer* Klasse veranschaulicht. Im Konstruktor werden die Nachrichten aus einer gegebenen Queue abgeholt und in einer Liste abgespeichert. Ein Validator, dem dieser *GenericMessageConsumer* zugewiesen wurde, kann nun den Typ der Listenelemente prüfen, die über *getMessages* zurückgegeben werden.

Listing 14: Die *SqsConsumer* Klasse

```
public class SqsConsumer extends GenericMessageConsumer<String> {

    private final GenericMessageList<String> messagesList;

    public SqsConsumer(String queueUrl) {
        super(String.class);
        AmazonSQS sqs = AmazonSQSClientBuilder.defaultClient();
        List<Message> sqsMessages = sqs.receiveMessage(queueUrl).getMessages();
        for (Message m : sqsMessages) {
            sqs.deleteMessage(queueUrl, m.getReceiptHandle());
        }
        this.messagesList = new GenericMessageList<>(String.class);
        sqsMessages.stream().map(Message::getBody).forEach(messagesList::add);
    }

    @Override
    public GenericMessageList<String> getMessages() {
        return this.messagesList;
    }
}
```

6.4 Implementierung eines Validators

Die Schnittstelle für die Workflows bildet die abstrakte Klasse *CDCTValidator*. Im vorherigen Abschnitt wurde zusammen mit Abbildung 15 verdeutlicht, dass die Fähigkeit, verschiedene Quellen bzw. Inputformate zu lesen, von der Implementierung des Validators abhängig ist. Damit eine Validierung in Abhängigkeit des Inputs stattfinden kann, müssen zwei Methoden implementiert werden. Die dafür wichtigen Methoden finden sich in Listing 15.

Listing 15: Der Kern der abstrakten *CDCTValidator* Klasse

```
public abstract class CDCTValidator<T> {

…
    protected GenericMessageConsumer<T> consumer;
    protected List<Path> schemas;

    protected abstract List<Class<?>> getSupportedMessageClasses();

    protected abstract void validate(
            GenericMessageList<T> listOfSubjects,
            List<Path> listOfSchemaPaths)
            throws CDCTException;

    public void validate() throws CDCTException {
        if (this.consumer == null || this.schemas == null || this.schemas.isEmpty()) {
            throw new ValidatorSetupException(
                    "Consumer or Schemas not set. Consumer=<"
                            + this.consumer
                            + "> Schemas=<"
                            + this.schemas + ">");
        }
        if (!getSupportedMessageClasses().contains(this.consumer.getGenericType())) {
            throw new ValidatorSetupException("Validator supports Input to be one of: "
                    + this.getSupportedMessageClasses() + " but Consumer provides "
                    + consumer.getGenericType());
        }
        validate(this.consumer.getMessages(), this.schemas);
    }
}
```

Die erste Methode *getSupportedMessageClasses()* gibt eine Liste des Inputs zurück, ein Beispiel aus der *JsonValidator* Klasse wird in Listing 16 gezeigt.

Diese Implementierung spielt in der *validate()* Methode der *CDCTValidator* Klasse eine zentrale Rolle, um Fehlnutzung über aussagekräftige Fehlermeldungen aufzuzeigen.

Listing 16: Implementierung der Methode *getSupportedMessageClasses* des *JsonValidators*

```
public class JsonValidator<T> extends CDCTValidator<T> {
…
    @Override
    protected List<Class<?>> getSupportedMessageClasses() {
        return Arrays.asList(Path.class, String.class);
    }
}
```

Der *CDCTWorkflow* führt die Methode *validate()* der abstrakten Klasse *CDCTValidator* aus, welche in Listing 15 gezeigt wurde. Die darin verwendete Methode *validate(GenericMessageList<T>, List<Path>)* wird vom *JsonValidator* implementiert, der die Validierung der unterschiedlichen Inputtypen an jeweils eigene Methoden delegiert. In Listing 17 wird deutlich, dass der *JsonValidator* entweder Nachrichten vom Typ *String* oder Pfade zu Dateien validieren kann.

Listing 17: Implementierung der Methode *validate* des *JsonValidators*

```
public class JsonValidator<T> extends CDCTValidator<T> {
…
  @Override
  protected void validate(GenericMessageList<T> listOfSubjects, List<Path> listOfSchemaPaths)
        throws ValidationFailedException, ValidatorSetupException {
    if (listOfSubjects.getGenericType().equals(Path.class)) {
        validateFromPaths((List<Path>) listOfSubjects, listOfSchemaPaths);
        return;
    }
    if (listOfSubjects.getGenericType().equals(String.class)) {
        validateFromStrings((List<String>) listOfSubjects, listOfSchemaPaths);
        return;
    }
    // should never be here...
    throw new ValidatorSetupException("Trying to validate unsupported input type <"
            + listOfSubjects.getGenericType() + ">");
  }

}
```

6.5 Implementierung der SQL Query Definition Bibliothek

Die SQSD-Bibliothek besteht im Kern aus der abstrakten *SQSDValidator* Klasse, die das Einlesen des Schemas aus dem Entwurfskapitel 5.7.1 (SQL-Schema Definition (SQSD)) übernimmt und in Java Objekte der Klasse *SQSDColumnDefinition* überträgt.

Die Kommunikation zur Datenbank oder das Extrahieren der Schemainformationen sowie der Datenfelder ist je Datenbanksystem unterschiedlich, sodass dieser Code in der jeweiligen Implementierung gekapselt wird, die die abstrakte Klasse realisiert. Siehe dazu das Klassendiagram in Abbildung 16.

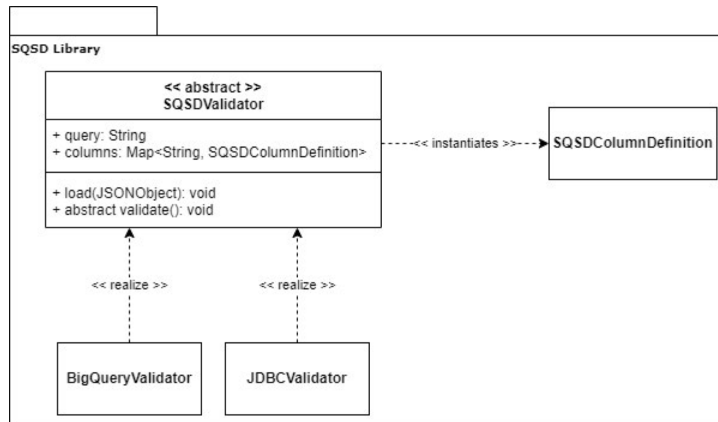


Abbildung 16: Klassendiagram der SQSD Bibliothek.

6.5.1 Implementierung des BigQueryValidators im CDCT-Framework

Damit BigQuery Abfragen bzw. Tabellen mithilfe des Frameworks als CDC bereitgestellt werden können, ist ein Wrapper nötig, der den *BigQueryValidator* als *CDCTValidator* implementiert.

Im Listing 18 ist zu sehen, wie dem *BigQueryValidator* erst das Schema (Aufruf der *load(JSONObject)* Methode, vgl. Abbildung 16) und dann das BigQuery *TableResult* aus einer *GenericMessageList* übergeben wird.

Weiter muss ein *MessageConsumer* implementiert werden, der das BigQuery *TableResult* dem *CDCTValidator* liefert. (Vgl. Abbildung 12: Vereinfachtes UML Klassendiagramm des CDCT-Frameworks).

Listing 18: Implementierung des *BigQueryValidator* als *CDCTValidator*.

```
public class BigQueryCDCTValidator extends CDCTValidator<TableResult> {

    @Override
    protected void validate(GenericMessageList<TableResult> listOfSubjects,
List<Path> listOfSchemaPaths) throws CDCTException {
        if(listOfSubjects.size() != 1 || listOfSchemaPaths.size() != 1){
            throw new ValidatorSetupException(
                     "BigQuery can only handle one TableResult being validated.");
        }
        BigQueryValidator bqv = new BigQueryValidator();
        try {
            JSONObject sqsd = new JSONObject(new String(Files.readAllBytes(listOf-
SchemaPaths.get(0))));
            bqv.load(sqsd);
        } catch (IOException | JSONException e) {
            throw new ValidatorSetupException("Could not read SQSD due " + e.get-
Message());
        }
        try {
            bqv.validate(listOfSubjects.get(0));
        } catch (SQSDValidationException e) {
            throw new ValidationFailedException("Validation of SQSD <" + listOf-
SchemaPaths.get(0) + "> failed due " + e.getMessage());
        }
    }

    @Override
    protected List<Class<?>> getSupportedMessageClasses() {
        return List.of(TableResult.class);
    }
}
```

7 Evaluierung / Bewertung

Die Evaluierung im Unternehmen startete in einer frühen Entwicklungsphase, bei der noch kein System produktiv im Einsatz war und endete zwei Wochen nachdem die ersten Systeme den produktiven Betrieb gestartet hatten und erste Datenstrecken in der BI aufgenommen wurden.

Für den Go-live wurden von der BI zwei Datenquellen von zwei unterschiedlichen Teams angebunden. Der Effekt von CDCs auf die Zusammenarbeit zwischen der BI und den Produktteams lässt sich aufgrund der Tatsache gut bewerten, dass Team A die Implementierung der CDCs in ihre Pipeline frühzeitig forcierte, während das andere Team B aus Zeitgründen die Implementierung der CDCs auf ein Datum nach der Liveschaltung ihres Systems verschoben hat. In beiden Fällen handelt es sich um JSON Nachrichten, die über AWS SQS konsumiert werden.

Darüber hinaus werden die Fähigkeiten des CDCT-Frameworks in Bezug auf die gestellten Anforderungen A1 bis A5 bewertet.

7.1 Bewertung der Anforderung A1

Die Anforderung A1, sich an bestehenden CDC Konzepten zu bedienen, wurde mit der Docker CDC-Variante erfüllt. Auch wenn die Produktteams für Docker Images die Elastic Container Registry (ECR) verwenden, war die Integration der GitHub Container Registry dank einheitlicher Verwendung von GitHub Repositorien mit geringem Aufwand möglich und stieß uneingeschränkt auf Akzeptanz. Damit ist die Grundvoraussetzung von CDCs, dass Producer die Tests verwenden, erfüllt.

7.2 Bewertung der Anforderung A2

Anforderung A2, Tolerant Reader, ist aus der Architekturvorgabe entstanden, nur die Daten abzuziehen, die für die eigene Arbeit nötig sind. Es ist möglich, diese Fähigkeit im CDCT-Framework für bestimmte Datenformate zu aktivieren, darunter in JSON-Nachrichten und durch explizite SQL-Abfragen. Während der Evaluierung wurde dies jedoch nicht umgesetzt, da sich die Anforderungen an die BI mit der Architekturvorgabe widersprechen. So wurde seitens der Fachbereiche gefordert, immer alle Datenfelder der Quelle abzuziehen und für die Adhoc Analyse bereitzuhalten, um diese schnell analysieren zu können. Aufgrund der DSGVO muss aber sichergestellt werden, dass neue, der BI unbekannte Felder, keine personenbezogenen Informationen enthalten, da diese „PII-Felder" sonst verschlüsselt werden müssen. Für die gezielte Verschlüsselung müssen diese Felder der BI jedoch bekannt sein. Folglich müssten neue, unbekannte Felder standardmäßig verschlüsselt werden, dadurch sind diese Felder für die Analysten, die nur verschlüsselte Daten bei PII-Feldern sehen dürfen, jedoch nicht mehr analysierbar. Dazu kommt, dass aufgrund dessen die Tolerant Reader Fähigkeit im ETL Framework noch nicht implementiert wurde. Daher wurden beide während der Evaluierungen erstellten CDCs nicht tolerant gegenüber Änderungen definiert, um eine Information vom Producer an den Consumer (BI) bei neuen Feldern zu erzwingen.

Dass sich Quellen nach der Implementierung einer Datenstrecke bis zur Liveschaltung des Systems ändern, war abzusehen. Team B hatte den CDC nicht implementiert, sodass es bei der Liveschaltung der Datenstrecken zu erheblichen Verzögerungen kam, da unterwartete Felder in der BI ergänzt und vom Datenschutzbeauftragten abgenommen werden mussten. Diese Probleme traten im Zusammenspiel mit Team A nicht auf. Abschließend lässt sich schlussfolgern, dass ein Großteil der Verzögerungen durch den konsequenten Einsatz der mit diesem Framework erstellten CDCs hätte verhindert werden können.

7.3 Bewertungen der Anforderungen A3 bis A5

Die Fähigkeiten des Frameworks bezüglich der Anforderungen A3 und A4 für verschiedene Datenformate und Quellen sowie A5, der Validierung, wurden in Tabelle 4 verdeutlicht. Eine

Erweiterung der Fähigkeiten für zukünftige Formate und Quellen sind dank der Abstraktionsebene des Frameworks mit etwas Einarbeitung möglich. Während der Evaluierung wurden zwei Quellen mit CDCs versorgt, dessen eingehenden JSON-Nachrichten nach den Bedürfnissen der BI geprüft werden konnten. Die Schemadefinitionen der BI unterscheiden sich dabei von jenen, die Schnittstellen selbst zu Verfügung stellen. Beispielsweise enthalten Schema der Schnittstellen häufig erlaubte Ausprägungen für Felder, wenn diese ein Enum wie die Versandmethode darstellen. Da diese Felder in der BI üblicherweise in der Datenbank in Feldern wie VARCHAR gespeichert werden, kann die BI das Schema flexibler gestalten, indem nur die Feldlänge im CDC geprüft wird.

7.4 Wesentliche Erkenntnisse

Die Evaluierung insbesondere in Bezug auf A3 bis A5 hat zwei wesentliche Erkenntnisse hervorgebracht.

Die BI konnte mithilfe der Dokumentation des Quellteams vor der Implementierung mit der Bereitstellung von CDCs beginnen. Bei der Implementierung des CDCs bei Team A sind Ungereimtheiten zwischen der Dokumentation und der eigentlichen Implementierung aufgefallen und konnten daher frühzeitig geklärt oder korrigiert werden. Da Team B die CDCs nicht implementiert hatte, hätten Integrationstests stattfinden müssen. Die von Jyri Lehvä et al. im Kapitel 2.4 beschrieben positiven Auswirkungen von CDCT bei Microservices, lassen sich daher auch auf die Schnittstellen zur BI übertragen.

Die zweite Erkenntnis ist die Notwendigkeit, die bereitgestellten CDCs auch selbst zu testen. Eine Prüfung, ob valide Nachrichten korrekt validiert werden, reicht nicht aus. Beispielsweise gab es in einem Fall ein gültiges JSON-Schema, bei dem die Direktive *additionalProperties: false* an der falschen Stelle platziert wurde, sodass das Schema selbst gültig war, die Direktive aber nicht evaluiert wurde. Eine Änderung in der Quelle führte so trotz ausgeführten CDC zu einem Abbruch im ETL-Prozess, da keine Spalte für das neue Datenfeld vorhanden war.

7.5 Einschränkungen

Das Absichern von Tabellen durch weitere BI Teams ließ sich nicht evaluieren, da derzeit nur ein einziges BI Team aktiv ist. Auch für die ersten Berichte konnten keine CDCs durch den Fachbereich bereitgestellt werden, da diese von der BI Abteilung entwickelt wurden. Somit ist die zu sichernde SQL-Abfrage nur dem BI Team bekannt. Auch wenn das CDCT-Framework der BI ermöglicht, sich selbst Test-Images zu erzeugen, sind diese nicht Consumer-Driven im eigentlichen Sinne. Weiter ist der zukünftige Einsatz des Frameworks im Fachbereich fragwürdig, da diese keine auf Entwickler zugeschnitten Produkte wie GitHub Repositorien und Docker nutzen. Die Zielgruppe für die SQSD-Bibliothek wird sich mutmaßlich auch wegen der benötigten Kenntnisse beispielsweise zu Datentypen der jeweiligen Datenbank und regulären Ausdrücken auf Data Analysten und BI-Teams beschränken.

8 Fazit

Die Evaluierung hat gezeigt, dass sich die positiven Effekte von CDCT auch in die BI übertragen lassen. CDCs können mit dem entwickelten Framework in kurzer Zeit bereitgestellt werden, sodass bereits bei der technischen Anbindung oder gar der Entwicklung der Schnittstelle ein direkter Austausch zwischen den betroffenen Teams stattfindet. Dadurch können Fehler und technische Details frühzeitig aufgeklärt werden. Obwohl die CDCs innerhalb der BI während der Evaluierung nicht Consumer-Driven im eigentlichen Sinne waren, können diese sinnvoll in die eigene Testpipeline aufgenommen werden. Der Wunsch, CDCs auch durch die Kunden der BI Abteilung erstellen zu lassen, ließ sich mit der entwickelten Lösung aufgrund des notwendigen technischen Know-hows der verwendeten Technologien nicht vollständig verwirklichen.

Auch wenn Jyri Lehvä et al. in seiner Fallstudie beschreibt, dass CDCT die Integrationstests gar ersetzen könne, muss berücksichtigt werden, dass auch CDCs fehlerhaft definiert werden

können. Die CDCs selbst sollten ebenfalls ausreichend vom Consumer getestet werden. Durch Integrationstests hätten weitere Probleme erkannt werden können und sollten daher nicht gänzlich aus der Testpyramide verschwinden.

8.1 Ausblick

Das konzipierte CDCT-Framework hat sich in der BI etabliert. In naher Zukunft steht die Erweiterung der möglichen Datenquellen (z.B. Kafka) und Formate (z.B. XML) bevor, welche nicht für die Evaluierung implementiert, aber in der Softwarearchitektur berücksichtigt wurden (vgl. Tabelle 4).

Technisch könnte das CDCT-Framework um eine zentrale Schema Registry ergänzt werden, damit der aktuelle Stand auch den ETL-Prozessen zur Verfügung steht. Um das Tolerant Reader Pattern in die ETL-Prozesse zu tragen, könnte der CDC auf Basis des JSON-Schemas zusätzliche Attribute erlauben, während der ETL-Prozess das Schema als Filter nutzt, um alle unbekannten Felder aus der Nachricht zu entfernen.

Weiter sind in der jetzigen Projektstruktur das Framework als auch die CDCs im selben Repository. CDCT-Framework Code und CDCs könnten getrennt werden, sodass bei mehreren BI Teams jedes ihre CDCs im eigenen Repository verwalten kann.

Die eigens entwickelte Validierungsbibliothek für SQL-Abfragen bzw. Datenbanktabellen und Views „SQSD“ reicht für die denormalisierten BigQuery Tabellen aus. Eine Weiterentwicklung für die Unterstützung des JSON-Datentyps, der beispielsweise in BigQuery oder MySQL möglich ist, könnte in Betracht gezogen werden. Das Herausnehmen der SQSD-Bibliothek in ein eigenständiges Artefakt würde außerdem eine unabhängige Weiterentwicklung ermöglichen.

Dass Fachbereiche der BI Abteilung CDCs bereitstellen können, stellt die größte offene Herausforderung dar. Welche Konzepte oder Frameworks bei Anwendern ohne technische Vorkenntnisse auf Akzeptanz stoßen würden, wäre Gegenstand weiterer Evaluierungen. Ein möglicher Kandidat könnte das Robot-Framework sein, dessen Testfälle sich angelehnt an natürlicher Sprache schreiben lassen.

Literaturverzeichnis

Armbrust et al. Lakehouse: A New Generation of Open Platforms that Unify Data Warehousing and Advanced Analytics.

Azvine, Ben/Cui, Zhan/Nauck, Detlef (2005). Towards real-time business intelligence. BT Technology Journal (Vol 23 No 3), 214–225.

Baeldung (2020). Consumer Driven Contracts with Pact | Baeldung. Online verfügbar unter https://www.baeldung.com/pact-junit-consumer-driven-contracts (abgerufen am 17.04.2022).

Collier, Ken (2012). Agile Analytics: A Value-Driven Approach to Business Intelligence and Data Warehousing.

Dehghani, Zhamak (2020). Data Mesh Principles and Logical Architecture. Online verfügbar unter https://martinfowler.com/articles/data-mesh-principles.html (abgerufen am 19.12.2021).

Dehghani, Zhamak (2021). How to Move Beyond a Monolithic Data Lake to a Distributed Data Mesh. Online verfügbar unter https://martinfowler.com/articles/data-monolith-to-mesh.html (abgerufen am 19.12.2021).

Drexl, Josef/Hilty, Reto M./Desaunettes, Luc/Greiner, Franziska/Kim, Daria/Richter, Heiko/Surblytė, Gintarė/Wiedemann, Klaus. DATA OWNERSHIP AND ACCESS TO DATA. POSITION STATEMENT OF THE MAX PLANCK INSTITUTE FOR INNOVATION AND COMPETITION OF 16 AUGUST 2016 ON THE CURRENT EUROPEAN DEBATE. Max Planck Institute for Innovation and Competition Research Paper (No. 16-10).

Ernest Teniente, Carles Farre, Toni Urpi, Carlos Beltran, and David Ganan (University of Catalunya, Spain. SVT: Schema Validation Tool for Microsoft SQL-Server.

Gabriel, Roland/Pastwa/Alexander/Gluchowski, Peter (2009). Data Warehouse & Data Mining.

Jung, Reinhard/Winter, Robert (2000). Data Warehousing 2000. Methoden, Anwendungen, Strategien. Heidelberg, Physica-Verlag.

Klein, Linda (2020). „Als Payment-Dienstleister übernehmen wir die komplette Zahlungsabwicklung“. Online verfügbar unter https://www.otto.de/newsroom/de/technologie/otto-baut-eigene-payment-gesellschaft-auf (abgerufen am 26.03.2022).

Kotwal (Thoughtworks), Priyanka (2020). PaymentHub Contract Testing Approach. Online verfügbar unter https://confluence.otto.de/x/ygVwGQ.

Lehvä, Jyri/Mäkitalo, Niko/Mikkonen, Tommi (2019). Consumer-Driven Contract Tests for Microservices: A Case Study. In: Xavier Franch/Tomi Männistö/Silverio Martínez-Fernández (Hg.). Product-Focused Software Process Improvement. Cham, Springer International Publishing, 497–512.

Leipert, Ralph (2021). Die relevanten Layer für eine erfolgreiche BI (Business Intelligence). Überblick - Business Intelligence Layer Architektur. Online verfügbar unter http://www.business-intelligence24.com/business-intelligence-definition/business-intelligence-layer-architektur (abgerufen am 19.12.2021).

Luber, Stefan (2018). Was ist ein Data Lake? Online verfügbar unter https://www.bigdata-insider.de/was-ist-ein-data-lake-a-686778/ (abgerufen am 27.12.2021).

MicroStrategy Inc. (2021). Testing Report and Document Execution in a Projec. Online verfügbar unter https://www2.microstrategy.com/producthelp/current/IntegrityManager/html/Testing_report_and_document_execution_in_a_project.htm (abgerufen am 15.05.2022).

Murillo, Mauricio (2016). Agile Data Warehousing and Business Intelligence in Action | Thoughtworks. Online verfügbar unter https://www.thoughtworks.com/insights/blog/agile-data-warehousing-and-business-intelligence-action (abgerufen am 06.02.2022).

OTTO MA-EC E-Commerce Innovation & -Plattform (2018). Deepsea common event format.

Pact Foundation (2021). Pact Docs | Introduction. Online verfügbar unter https://docs.pact.io/ (abgerufen am 19.12.2021).

Robinson, Ian (2006). Consumer-Driven Contracts: A Service Evolution Pattern. Online verfügbar unter https://www.martinfowler.com/articles/consumerDrivenContracts.html (abgerufen am 19.12.2021).

Stamm, Christian (2013). Continuous Everything: Fast Feedback Driven Development | OTTO Tech | Blog. Online verfügbar unter https://www.otto.de/jobs/technology/techblog/artikel/continuous-everything-fast-feedback-driven-development_2013-11-14.php (abgerufen am 21.12.2021).

Vernon, Vaughn (2017). Domain-Driven Design kompakt. Heidelberg, dpunkt.verlag.

Vollerthun, Tom (2021). CDCs für otto.de – Continuous Everything (jetzt mit Cloud™). Online verfügbar unter https://www.otto.de/jobs/technology/techblog/artikel/cdcs-fuer-otto.de-continuous-everything-jetzt-mit-cloud.php (abgerufen am 19.12.2021).

Westermann, Utz/Azad, Mahmoud Reza Rahbar (2021). Evaporating a data lake: Otto Group’s lessons learned migrating a Hadoop infrastructure to GCP | Google Cloud Blog. Online verfügbar unter https://cloud.google.com/blog/topics/customers/otto-groups-lessons-learned-migrating-a-hadoop-infrastructure-to-gcp (abgerufen am 22.12.2021).

Erklärung zur selbstständigen Bearbeitung einer Abschlussarbeit

Hiermit versichere ich, dass ich die vorliegende Arbeit ohne fremde Hilfe selbständig verfasst und nur die angegebenen Hilfsmittel benutzt habe. Wörtlich oder dem Sinn nach aus anderen Werken entnommene Stellen sind unter Angabe der Quellen kenntlich gemacht.