Entwicklung einer ereignisorientierten Architektur für Business Process Management-Szenarien auf Basis von Apache Kafka

Bachelorarbeit

Hochschule der Medien Stuttgart
Fachbereich Information und Kommunikation
Studiengang Wirtschaftsinformatik und digitale Medien
(8. Semester)

Erstbetreuer: Herr Prof. Dr. David Klotz
Zweitbetreuerin: Frau Viktoria Heinzel

Sommersemester 2021

Vorgelegt von: Philipp Kurrle
Matrikelnummer: 33027

Stuttgart, 24.08.2020

Ehrenwörtliche Erklärung

Name:	Kurrle	Vorname:	Philipp
Matrikel-Nr.:	33027	Studiengang:	WI7

Hiermit versichere ich, *Philipp Kurrle* ehrenwörtlich, dass ich die vorliegende *Bachelorarbeit* mit dem Titel: *„Entwicklung einer ereignisorientierten Architektur für Business Process Management-Szenarien auf Basis von Apache Kafka“* selbständig und ohne fremde Hilfe verfasst und keine anderen als die angegebenen Hilfsmittel benutzt habe. Die Stellen der Arbeit, die dem Wortlaut oder dem Sinn nach anderen Werken entnommen wurden, sind in jedem Fall unter Angabe der Quelle kenntlich gemacht. Die Arbeit ist noch nicht veröffentlicht oder in anderer Form als Prüfungsleistung vorgelegt worden.

Ich habe die Bedeutung der ehrenwörtlichen Versicherung und die prüfungsrechtlichen Folgen (§ 26 Abs. 2 der Bachelor-SPO (6-Semester), § 24 Abs. 2 Bachelor-SPO (7-Semester), § 23 Abs. 2 Master-SPO (3 Semester) bzw. §19 Abs. 2 Master-SPO (4 Semester und berufsbegleitend) der HdM) einer unrichtigen oder unvollständigen ehrenwörtlichen Versicherung zur Kenntnis genommen.

.....24.08.2020.....
Ort, Datum

……..
Unterschrift

Kurzfassung

Im Kontext der Digitalisierung und der dadurch zunehmenden Veränderung von Geschäftsprozessen entstehen auch neue Anforderungen an betriebliche Anwendungssysteme. Hierbei entstehen insbesondere Probleme durch unagile, starre Prozessintegration und fehlende Standardisierung. Lösungsansätze bietet der Architekturstil Event-Driven Architecture und die Softwaretechnologie Complex Event Processing. Dazu wird folgende Forschungsfrage gestellt: *Wie kann eine ereignisorientierte Softwarearchitektur für BPM-Szenarien aussehen, die sowohl Interoperabilität, Agilität und Qualität bietet?* Als Forschungsmethode wird Design Science Research verwendet. Um die Forschungsfrage zu beantworten wird zuerst eine neue Anwendungsarchitektur entworfen, anschließend werden Teile dieser Architektur prototypisch implementiert. Die Architektur und der Prototyp werden anhand sechs qualitativer Kriterien evaluiert. Die Evaluation zeigt, dass Event-Driven Architecture die definierten Probleme der lösen kann und in der Lage ist die konzeptionelle Lücke zwischen Geschäftsprozessen und Anwendungssystemen zu schließen. Die entworfene ereignisgesteuerte Architektur bietet sowohl die nötige Interoperabilität als auch Agilität.

Abstract

In the context of digitalization and the increasing change of business processes, new requirements for business application systems arise. Problems arise in particular from unagile, rigid process integration and lack of standardization. Event-Driven Architecture and Complex Event Processing offer solutions. This thesis aims to answer the following research question is asked: *How should an event-driven software architecture that provides interoperability, agility and quality be designed for BPM scenarios?* Design Science Research is used as the research method. In order to answer the research question, a new application architecture is first designed partly implemented. The architecture and the prototype are evaluated against six qualitative criteria. The evaluation shows that Event-Driven Architecture can solve the defined problems and is able to close the conceptual gap between business processes and the application system. The designed event-driven architecture provides both the necessary interoperability and agility.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Abkürzungsverzeichnis

BPM	Business Process Management
CEP	Complex Event Processing
DP	Design Prinicple
DSR	Design Science Research
EDA	Event-Driven Architecture
EPA	Event Processing Agents
EPL	Event Processing Language
EPN	Event Processing Network
GQM	Goal-Question-Metric

1 Einleitung

Durch digitale Vernetzung und die dadurch resultierende Informationsverfügbarkeit können benötigte Informationen bedarfsgerecht am richtigen Ort bereitgestellt werden. Diese Vernetzung ermöglicht effizientere und flexiblere Prozesse (Emmerich et al., 2015, S. 14). Allerdings entstehen in konventionellen Systemen durch ungeeignete sowie mangelhafte Informationsverfügbarkeit Verschwendungen wie unnötige Materialbewegungen, Liegezeiten und Latenzen (Heger et al., 2020, S. 711). Das liegt daran, dass konventionelle betriebliche Anwendungssysteme oft nicht die Flexibilität bieten, um die fachliche Agilität abzubilden. Darüber hinaus wachsen die internen und externen Anforderungen an Agilität stetig. Dies geschieht unter anderem durch neue Möglichkeiten oder verändertes Kundenverhalten (Avery, 2019, S. 3; Bruns & Dunkel, 2010, S. 3).

Durch historisches Wachstum und eine fehlende unternehmensweite IT-Strategie entstehen komplexe, heterogene Anwendungssysteme (Aier & Dogan, 2005, S. 607–608; Österle, 1996, S. 4–5). Durch eine zunehmende Dezentralisierung der Unternehmen sowie eine höhere Verfügbarkeit und Leistungsfähigkeit Anwendungssystemen wird diese Heterogenität verstärkt. Die Folge ist eine mangelnde Interoperabilität der Anwendungssysteme (Müller, 2005, S. 39). Da konventionellen Anwendungsarchitekturen meist eine ablauforientierte Sicht zugrunde liegt, sind diese Systeme zu unflexibel, um Geschäftsprozesse zu verstehen. Sie können nicht zeitnah auf Fehler reagieren. Dies kann immense Kosten verursachen (Bruns & Dunkel, 2010, S. 8).

Dadurch ergeben sich drei Kernprobleme konventioneller Anwendungsarchitekturen: Fehlende Agilität, Interoperabilität sowie eine niedrige Qualität durch Fehler in Prozessen. Lösungen könnten der Architekturstil Event-Driven Architecture (EDA) und die Softwaretechnologie Complex Event Processing (CEP) bieten. Sie repräsentieren eine neue Art von betrieblicher Softwarearchitektur, bei der Ereignisse in das Zentrum dieser rücken. EDA in Verbindung mit CEP bietet zahlreiche neue Möglichkeiten im *Business Process Management* (BPM) (Lundberg, 2006, S. 55–56).

Trotz dieser Tatsache finden EDA und CEP selten Anwendung in diesem Bereich. Während in Anwendungsgebieten wie Smart Home, Fahrzeugüberwachung, Aktienhandel oder Business Activity Monitoring diese Technologien bereits Anwendung finden (Hedtstück, 2017, S. 5–9), gibt es trotz ausführlicher Recherche noch keine nennenswerten Anwendungen im BPM-Umfeld. Dies scheint verwunderlich, da gerade mangelnde Agilität durch den Einsatz von CEP gelöst werden könnte (Bruns & Dunkel, 2010, S. 5). Laut Bruns und Dunkel (2010) sind

die Gründe für die fehlende Verbreitung wenig erforscht, sie nennen potenzielle Gründe, von denen zwei trotz ihres Alters auch heute noch beobachtbar sind. Der erste möglichen Grund sind unvollständige sowie fehlende Standards. Dazu gehören vor allem fehlende technische Standards für wichtige EDA-Komponenten. Als weiterer Grund wird die Abwesenheit einer etablierten Entwicklungsmethodik genannt. Es existieren nur wenige ausgereifte Methodiken für die Entwicklung von ereignisgesteuerten Systemen. Insbesondere Richtlinien, Entwurfsmuster, Best-Practise-Beispiele sowie wiederverwendbare Komponenten für EDA existieren oftmals nur in rudimentärer Form (Bruns & Dunkel, 2010, S. 234–235). Unternehmen mit ereignisgesteuerten Architekturen verarbeiten Millionen von Ereignissen pro Sekunde (Churchward, 2008). Hier kann bereits ein kleiner Fehler in der Architektur großen Schaden anrichten (Bruns & Dunkel, 2010, S. 18). Aus dieser Beobachtung der momentanen Situation lassen sich insbesondere die drei Kriterien der fehlenden Standardisierung, Flexibilität sowie Qualität hervorheben, welche eine Architektur erfüllen sollte.

1.1 Ziele und Forschungsfrage

Ziel der Arbeit ist es herauszufinden, wie eine Softwarearchitektur konzipiert und implementiert werden muss, um einen Lösungsansatz die Probleme der fehlenden Interoperabilität, Agilität sowie Qualität in BPM-Szenarien zu bieten. Hierbei soll dem ursprünglichen Konzept der ereignisgesteuerten Geschäftsprozesse auf Basis einer modernen Architektur wieder neues Leben eingehaucht werden. In einem ersten Schritt sollen hierbei Anforderungen für eine solche Architektur definiert werden. In nachfolgenden Schritten wird mithilfe der Plattform *Apache Kafka* ein Prototyp implementiert, welcher es ermöglicht, ereignisgesteuerte Prozesse zu beschreiben, zu erstellen und diese zu validieren. Der Prototyp einer Softwarearchitektur für ereignisgesteuerte Geschäftsprozesse wird mit den folgenden Schwerpunkten entwickelt, um die definierten Kriterien zu erfüllen:

1. Definition einer Beschreibungssprache für Geschäftsprozesse.
2. Erzeugung von Kafka-Komponenten anhand der Beschreibungssprache.
3. Validierung der Prozesse und Ereignisse anhand der definierten Logik. Hierbei liegt der Fokus auf der Vermeidung von inkonsistenten Prozessinstanzen.

Anhand des Prototyps kann getestet werden, inwiefern die Gesamtarchitektur in der Lage ist die zuvor definierten Probleme zu lösen. Hierfür werden passende Metriken definiert. Anhand der Ergebnisse soll die folgende Forschungsfrage beantwortet werden:

Forschungsfrage
Wie kann eine ereignisorientierte Softwarearchitektur für BPM-Szenarien aussehen, die sowohl Interoperabilität, Agilität und Qualität bietet?

2 Theoretische Grundlagen

Im Folgen werden die theoretischen Grundlagen für die Technologien und Konzepte erläutert, welche für die Erfüllung der Forschungsfrage relevant sind. Darüber hinaus werden Begriffe definiert und abgegrenzt.

2.1 Ereignisorientierte Softwarearchitekturen

Für den Begriff der Architektur haben sich in der Wirtschaftsinformatik eine Vielzahl von Ableitungen wie beispielsweise Softwarearchitektur, IT-Architektur oder Systemarchitektur entwickelt. Es existieren für die zentralen Begriffe bereits etablierte Definitionen (Rennenkampff et al., 2015, S. 54).

Eine Definition stammt aus dem IEEE 1471:2000 Standard:

> "*Architecture is the fundamental organization of a system embodied in its components, their relationships to each other and to the environment and the principles guiding its design and evolution.*" (ISO/IEC/IEEE 42010: Defining „architecture", o. J.).

Diese Definition umfasst nach Vogel (2009) die wichtigsten Aspekte einer Software-Architektur: (Vogel, 2009, S. 66)

- Die Software-Struktur eines Systems
- Die Software-Bausteine eines Systems
- Die Eigenschaften der Software-Bausteine
- Die Beziehungen zwischen den Software-Bausteinen

Für die Entwicklung einer Architektur können verschiedene Ansätze gewählt werden, welche von Vogel (2009) näher ausgeführt werden.

Komponentenorientierung: Komponenten sind wiederverwendbar und in sich geschlossene Bausteine einer Architektur. Komponentenorientierung entstand aus der Problematik, dass Objekte in der objektorientierten Programmierung zwar das Modularitätsprinzip umsetzten, aber als wiederverwendbare Einheiten oft zu klein sind (Vogel, 2009, S. 161–162). Eine Komponente ist „eine Kompositionseinheit mit vertraglich spezifizierten Schnittstellen, die nur explizite Abhängigkeiten zu ihrem Kontext hat. Eine Software-Komponente kann unabhängig eingesetzt werden und sie kann durch Dritte komponiert werden" (Vogel, 2009, S. 162).

Meta-Architekturen/Reflexion: Durch Meta-Programmierung erreichen Software-Systeme durch eine zusätzliche Abstraktionsebene mehr Flexibilität und Kontrolle. In der Regel wird

zwischen dem Programm als ausführbare Anweisungen und den Daten, mit dem das Programm arbeitet, unterschieden. Während der Laufzeit werden normalerweise nur die Daten geändert und nicht das Programm selbst. Das Paradigma der Meta-Programmierung (auch Reflexion oder Introspektion genannt) macht dies allerdings möglich und erlaubt dem Programm den Zugriff auf sich selbst (Vogel, 2009, S. 164). Damit hat das Programm zur Laufzeit Informationen über Typinformationen, Klassen, Variablen und Methoden sowie die Vererbungshierarchie. Mit Reflexion ist es möglich, dynamisch Methoden aufzurufen, Klassen zu instanziieren (Vogel, 2009, S. 164).

Ereignisse in Unternehmensnetzwerken

Ereignisse spielen eine zentrale Rolle in Unternehmen und sind entscheidend bei der Steuerung von Geschäftsprozessen. Ein Geschäftsprozess ist „eine Folge von Schritten zur Erreichung eines angestrebten Arbeitsergebnisses und dient der Leistungserbringung in einem Unternehmen“ (Bruns & Dunkel, 2010, S. 14). Ein Geschäftsprozess lässt sich als „ereignisgesteuert“ bezeichnen, wenn eine Verarbeitungsfolge durch ein Ereignis ausgelöst wird (Bruns & Dunkel, 2010, S. 14). Geschäftsprozesse von Unternehmen beruhen auf dem Austausch von Ereignissen sowohl intern als auch in der Zusammenarbeit mit anderen Unternehmen. Demnach sind ereignisgesteuerte Unternehmen in der Realität weit verbreitet (Bruns & Dunkel, 2010, S. 6). Ein Ereignis kann alles sein, was passiert oder von dem erwartet wird, dass es passiert (Luckham & Schulte, 2011, S. 5).

Ereignisgesteuerte Unternehmen sind nach Bruns und Dunkel (2010) dadurch gekennzeichnet, dass

- sie Ereignisse erkennen, analysieren und auf sie reagieren,
- sie durch den Austausch von Ereignissen kommunizieren,
- sie neue Ereignisse erzeugen und diese versenden,
- ihre Prozessschritte bzw. Aktivitäten durch Ereignisse ausgelöst werden (Bruns & Dunkel, 2010, S. 20–21).

Event-Driven Architecture

Event-Driven Architectures (EDA) repräsentieren einen Architekturstil von Unternehmensanwendungen, bei dem Ereignisse im Fokus der Softwarearchitektur stehen (Bruns & Dunkel, 2010, S. 4). Der Architekturstil ermöglicht eine realitätsnahe Abbildung von ereignisgesteuerten Geschäftsprozessen (Bruns & Dunkel, 2010, S. 225). Ein Softwaremodul ist ereignisgesteuert, wenn Funktionen durch das Eintreffen eines Ereignisses ausgelöst werden (Luckham & Schulte, 2011, S. 15).

Ereignisgesteuerte Architekturen erfuhren in der Vergangenheit wachsende Aufmerksamkeit in verschiedensten Bereichen (Hamzah, 2020, S. 273–304; Shaheen et al., 2019, S. 149–154). Ereignisgesteuerte Geschäftsprozesse unterscheiden sich von traditionellen, ablauforientierten Prozessen. Diese Unterschiede haben direkten Einfluss auf die Architekturen von Anwendungssystemen. Die Implementierung von ereignisgesteuerten Geschäftsprozesse ist oftmals nur möglich, wenn dies auch die darunter liegende Architektur unterstützt (Bruns & Dunkel, 2010, S. 43; Schulte & Inc, 2003). Bei ereignisgesteuerten Prozessen löst ein Ereignis, welches eine Zustandsveränderung kennzeichnet, auf eine Funktion wie beispielsweise „Artikel fertigen“ aus (Scheer, 2002, S. 18).

Die Architektur besteht aus stark entkoppelten Komponenten, welche die Ereignisse asynchron empfangen und verarbeiten (Richards & Media, o. J., S. 19). Bei einer Event-Driven Architecture lassen sich drei logische Strukturierungsschichten unterscheiden, welche in Abb. 1 dargestellt sind (Bruns & Dunkel, 2010, S. 43).

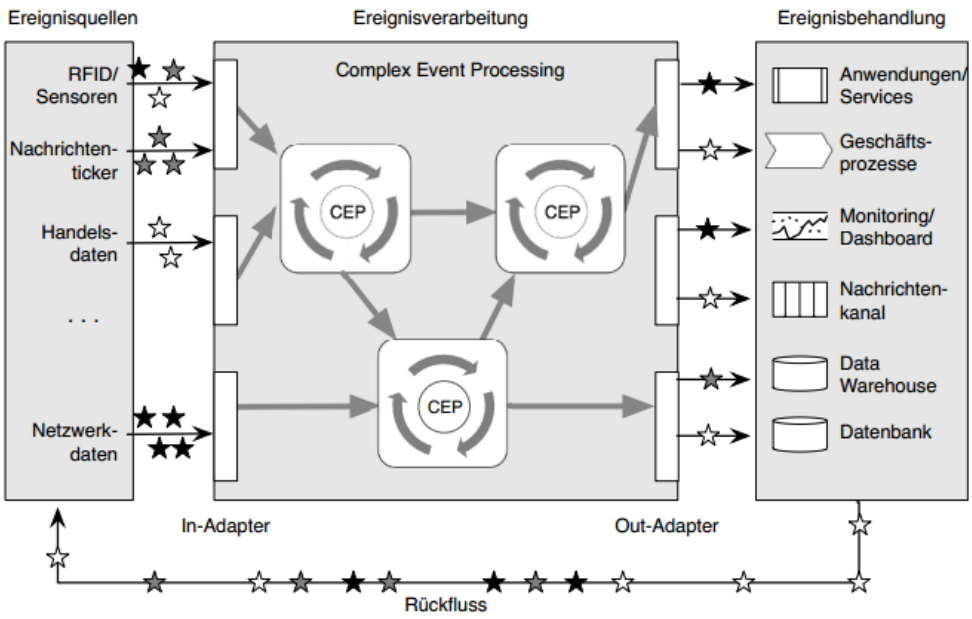

Abb. 1: EDA-Schichten
(Quelle: Eigene Darstellung)

1. **Ereignisquellen**: Jedes Ereignis muss eine Quelle besitzen. Diese Ereignisquellen sind in der Realität sehr heterogen. Beispiele für Quellen sind Sensoren, Feeds, Softwaremodule oder Web Services (Bruns & Dunkel, 2010, S. 61–62).

2. **Ereignisverarbeitung:** Ströme von Ereignisdaten aus mehreren Ereignisquellen dienen der Ereignisverarbeitung (*event processing*) als Eingabe. In ereignisgesteuerten Architekturen sind Ereignisströme in der Regel anhand des Auftrittszeitpunktes geordnet. Die eigentliche Verarbeitung des Stroms und dessen Ereignissen erfolgt durch *Complex Event Processing*, auf welches später noch eingegangen wird (Bruns & Dunkel, 2010, S. 62–63)

3. **Ereignisbehandlung:** Die eigentliche Ereignisbehandlung (*event handling*), also die Reaktion auf eintreffende Ereignisse, findet im *Backend* der Unternehmensanwendungen statt. Eine Softwarekomponente aus der Ereignisbehandlungsschicht kann selbst wiederum Ereignisse genieren und somit eine Quelle sein (Bruns & Dunkel, 2010, S. 63–64).

Herausforderungen im Umgang mit Ereignissen

Die Menge an Ereignissen innerhalb von Unternehmen steigt kontinuierlich an. Folgende Faktoren tragen im Wesentlichen dazu bei, dass Unternehmen mit einem steig wachsenden Strom von Ereignissen umgehen müssen (Bruns & Dunkel, 2010, S. 15–16):

1. **Fachliche Komplexität:** Geschäftsprozesse eines Unternehmens können durch eine fortschreitende Arbeitsteilung abteilungs- als auch standortübergreifend sein. Dadurch entsteht eine hohe unternehmensübergreifende Kopplung von Prozessen, was zur einer erhöhten fachlichen Komplexität führt.

2. **Technische Komplexität:** Die meisten größeren Unternehmen besitzen eine komplexe und heterogene IT-Infrastruktur, welche historisch gewachsen ist. Es wird eine Vielzahl an Programmiersprachen, Technologien, Datenformaten und Schnittstellen verwendet.

3. **Kommunikation mit der physikalischen Umwelt:** Durch den großräumigen Einsatz von Sensoren entsteht eine starke Verbindung zwischen den Unternehmenswendungen und der Außenwelt. Dadurch wird die Ereignismenge erhöht.

4. **Datenmenge:** Unternehmen müssen die zunehmende Anzahl an Ereignissen systematisch identifizieren und zeit- sowie ortsabhängig bereitstellen. Eine

Herausforderung ist es hierbei, die anwachsende Menge an Ereignisdaten in die laufenden operativen Geschäftsprozesse zu integrieren.

Vorteile der Ereignisorientierung

Die Anwendung von Ereignisorientierung bietet mehrere Vorteile:

Agilität: Da bei einer ereignisgesteuerten Architektur die ereignisverarbeitenenden Komponenten weitgehend entkoppelt sind, kann sich das System schnell an veränderndes Umfeld anpassen (Avery, 2019; Bruns & Dunkel, 2010, S. 73).

Kapselung: Da jede Komponente nur genau einem Zweck dient (*single-purpose*), kann sich das System schnell an Veränderungen anpassen. Veränderungen der Umwelt betreffen meist nur eine beschränkte Anzahl an Komponenten, welche schnell identifiziert und verändert werden können (Avery, 2019).

Skalierbarkeit: Durch das *Publish/Subscribe*-Prinzip und die Entkopplung können ereignisgesteuerte Systeme physikalisch verteilt und somit skaliert werden. Jede ereignisverarbeitende Komponente kann einzeln skaliert werden, was eine sehr feinkörnige Skalierung erlaubt (Bruns & Dunkel, 2010, S. 74).

Evolutionär: Ereignisgesteuerte Systeme und Ereignisse können sich über einen längeren Zeitraum verändern (Avery, 2019).

Anwendungsgebiete

Ereignisse spielen in vielen Unternehmensprozessen eine signifikante Rolle. Allerdings ergibt nicht im jeden Kontext eine exklusive EDA-Lösung Sinn. Stark strukturierte Geschäftsprozesse ohne Echtzeitanforderungen lassen sich auch durch konventionelle Architekturen umsetzen (Bruns & Dunkel, 2010, S. 39). Es existieren jedoch einige Kernanforderungen an Systeme, für welche EDA-Lösungen besonders gut geeignet sind. Wenn eine komplexe Fachlogik wie beispielsweise die komplexe Aggregation von Daten oder das Erkennen von Ereignismustern benötigt wird, eignet sich EDA (Bruns & Dunkel, 2010, S. 39). Des Weiteren können mithilfe der Technologie große Datenvolumina mit einer geringen Latenzzeit verarbeitet werden. Somit sind Reaktionen in Echtzeit möglich. Darüber hinaus bieten EDAs Skalierbarkeit und Agilität und können somit auf steigende Datenaufkommen reagieren und einfach geändert und gewartet werden (Bruns & Dunkel, 2010, S. 39).

Beispielhafte Anwendungsgebiete sind daher Kontroll- und Überwachungssysteme, Sensornetzwerke, Business Intelligence Anwendungen und Wertpapierhandel (Bruns &

Dunkel, 2010, S. 40 ff.). Prominente Anwender der von EDA mittels Apache Kafka sind Unternehmen wie Twitter, Netflix sowie LinkedIn, welche mit der Technologie Nutzerdaten verarbeiten (Gour, 2018).

Complex Event Processing

Während EDA einen ereignisorientierten Entwurfsstil für Anwendungsarchitektur beschreibt, handelt es sich bei *Complex Event Processing* (CEP) um eine Technologie zur dynamischen Verarbeitung von Ereignissen in Echtzeit. CEP kann als Kernbestandteil der EDA angesehen werden, beide verwenden jedoch das Grundkonzept der Ereignisorientierung (Bruns & Dunkel, 2010, S. 4). Mit EDA ist es möglich, Ereignisse aus unterschiedlichen Quellen nach Ereignismustern zu verarbeiten. Nach dem Erkennen eines Musters kann eine entsprechende Aktion wie die Generierung eines neuen Ereignisses oder der Aufruf eines Dienstes ausgeführt werden (Bruns & Dunkel, 2010, S. 44).

CEP besteht aus drei Komponenten:

- **Ereignisregeln:** Definieren einen Bedingungsteil, der ein Ereignismuster beschreibt, sowie einen Aktionsteil, welcher ausgeführt wird, wenn der Bedingungsteil erfüllt ist. Ereignisregeln können mittels einer *Event Processing Language* (EPL) definiert werden (Bruns & Dunkel, 2010, S. 12).
- **Event Processing Agents (EPAs):** Überwachen Ereignisströme und führen zuvor definierte Ereignisregeln aus (Bruns & Dunkel, 2010, S. 13).
- **Event Processing Networks (EPN):** Sind EPAs, welche in einem Netzwerk zusammenarbeiten (Bruns & Dunkel, 2010, S. 14).

2.2 Apache Kafka

Apache Kafka ist eine *Open-Source* Software der Apache Software Foundation, welche insbesondere der Verarbeitung von Ereignisströmen dient. Kafka ist ein verteiltes *messaging system*, welches eine schnelle, skalierbare Architektur mittels eines P*ublish/ Subscibe*-Prinzips bietet (Narkhede et al., 2019, S. 1–4).

Ursprünglich war Apache Kafka eine interne Entwicklung von LinkedIn. Ende 2010 wurde der Quellcode von Kafka veröffentlicht und gehört seitdem zur Apache Foundation. Im Jahr 2014 gründeten die Entwickler von LinkedIn das Unternehmen Confluent, welches sich bis heute auf die Entwicklung von Apache Kafka fokussiert (Narkhede et al., 2019, S. 14–15).

Publish/Subscribe-Architektur

Publish/Subscribe ist ein Architekturstil, welcher eine Alternative zu klassischen *Client/Server* Architekturen darstellt (Shaw & Garlan, 1996). Der Austausch von Nachrichten in ereignisgesteuerten Architekturen erfolgt in der Regel in Form von Ereignissen. Die *Publish/Subscribe*-Architektur besitzt Komponenten, welche Ereignisse erzeugen (*Publisher*). Diese leiten Ereignisse an einen Vermittler (*Broker*) weiter, ohne zu wissen, wann und wie die Weiterverarbeitung erfolgt. Stattdessen wird die Nachricht klassifiziert und der Empfänger (*Subscriber*) empfängt eine bestimmte Klasse an Nachrichten. Damit werden Aufrufe nicht direkt unter den Komponenten versendet, sondern Ereignisse werden durch einen Vermittler, den *Broker*, weitergeleitet (Vogel, 2009, S. 228–229). In Abb. 2 ist dieser Prozess beispielhaft dargestellt.

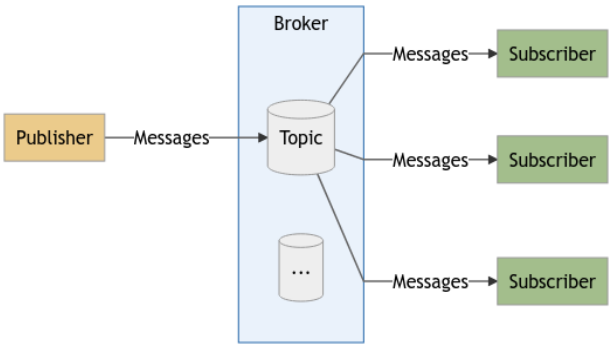

Abb. 2: Beispiel eines Publish/Subscribes-Systems beschrieben von Vogel, 2009, S. 244 (Quelle: Eigene Darstellung)

Die Architektur widmet sich dem Problem, dass eine Reihe von Klienten über Ereignisse informiert werden müssen. Manchmal müssen viele Klienten aktiv über ein Ereignis informiert werden, in anderen Fällen ist nur ein Klient am Ereignis interessiert. Deshalb müssen der Produzent und Konsument eines Ereignisses voneinander entkoppelt werden, um beispielsweise getrennte Veränderbarkeit zu gewährleisten (Vogel, 2009, S. 229). Die Kommunikation zwischen *Publisher* und *Subscriber* erfolgt asynchron. Die ereigniserzeugende Komponente setzt ihre Verarbeitung direkt fort, sobald sie eine Nachricht gesendet hat. Diese Asynchronität bildet die Basis für die hohe Interoperabilität von EDA (Bruns & Dunkel, 2010, S. 31).

Kafka Terminologie

Apache Kafka ist um einige wenige Schlüsselbergriffe herum organisiert, welche im Folgenden erläutert werden.

Die *Publisher* und *Subscriber* werden von Kafka *Producer* und *Consumer* genannt und folgen dem üblichen *Publish/Subscribe*-Prinzip. *Producer* sind Komponenten, welche Nachrichten auf ein Kafka *Topic* veröffentlichen. *Consumer* sind die Komponenten, welche Nachrichten von den *Topics* beziehen. *Consumer* sind meist innerhalb einer *consumer group* organisiert.

Kafka wird üblicherweise auf einem oder mehreren Servern als *Cluster* betrieben, welcher mit den anderen Komponenten wie in Abb. 3 dargestellt verbunden ist (Apache Kafka, o. J.).

Die atomare Dateneinheit in Kafka sind *Records*. Diese halten einen *key*, *timestamp* sowie einen Wert. Records werden in sogenannten *Topics* organisiert, welche eine Menge an *Records* enthalten (auch *Stream* genannt). Ein *Topic* ist die Kategorie unter welcher ein *Record* veröffentlicht wird. Ein *Topic* ist in mehrere Partitionen aufgeteilt, welche eine Parallelisierung erlauben, indem Daten aus einem *Topic* auf mehrere *Broker* verteilt werden. Jede Partition besteht aus einer unveränderlichen Sequenz aus *Records*, welche kontinuierlich erweitert wird. Dies wird in Kafka auch *commit-log* genannt. Jedem *Record* wird eine fortlaufende Nummer, das *offset*, zugeordnet, welches jeden Record innerhalb einer Partition eindeutig identifiziert (Apache Kafka, o. J.). Der *Broker* verwaltet die Speicherung der *Records* in *Topics*.

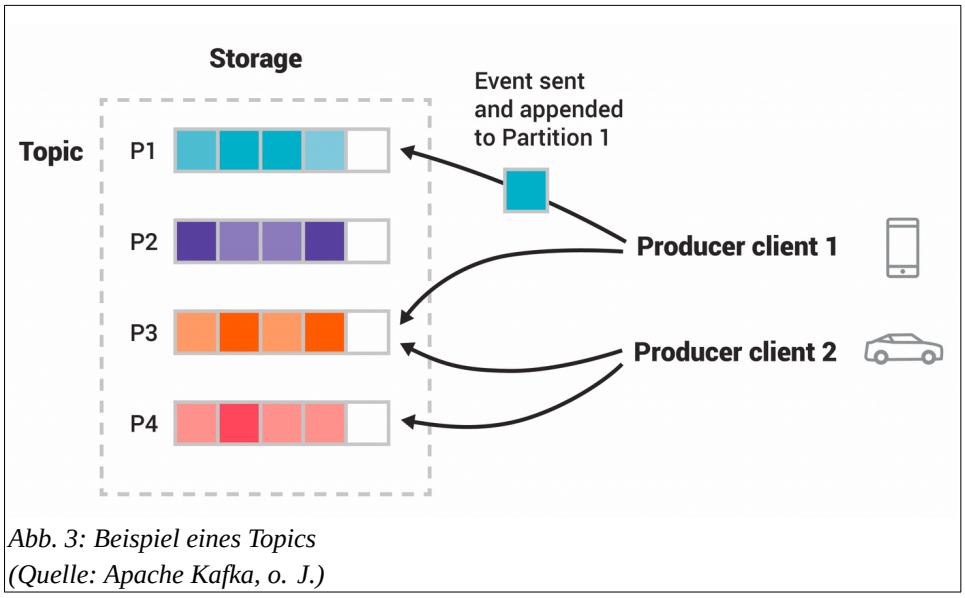

Abb. 3: Beispiel eines Topics
(Quelle: Apache Kafka, o. J.)

In Abb. 3 ist ein Beispiel eines *Topics* mit den vier Partitionen P1-P4 zu sehen. Zwei unterschiedliche *Producer* erzeugen Ereignissen unabhängig voneinander und schreiben diese in unterschiedliche Partitionen der *Topics*.

2.3 Agilität

In der Wirtschaftsinformatik existieren keine allgemein akzeptieren Definitionen von Agilität. (Rennenkampff et al., 2015, S. 39). Deshalb ist eine Abgrenzung nötig.

Agilität wird als Eigenschaft einer Organisation gesehen, Veränderungen schnell umzusetzen (Sambamurthy et al., 2003, S. 245; Terry Anthony Byrd, 2000, S. 170–172). Hierbei können die Veränderungen in kapazitive oder funktionale Veränderungen unterschieden werden.

Kapazitive Veränderungen führen dazu, dass die Menge der erbrachten Leistung angepasst werden muss (Rennenkampff et al., 2015, S. 48). Funktionale Änderungen hingegen führen zu inhaltlichen Anpassungen (Nissen & Mladin, 2009, S. 42–43). Bei der Agilität einer Anwendung lässt sich grundsätzlich zwischen kapazitiver Agilität und funktionaler Agilität unterscheiden. Das Ziel der hohen kapazitiven Agilität ist die Fähigkeit einer Anwendung, sich schnell an veränderte kapazitive Anforderungen anzupassen (Rennenkampff et al., 2015, S. 155–156). Funktionale Agilität erfordert, dass die Anwendung in der Lage ist, Änderungen an ihren Elementen schnell und effizient zu umzusetzen (Dern & Jung, 2009, S. 155). Die Änderungen sollten einfach und prompt umgesetzt werden können. Die steigende Zahl an Anforderungen an eine Anwendung sowie eine alternde und wachsende Anwendungslandschaft stellen Unternehmen vor Herausforderungen. Um eine funktionale Agilität zu besitzen, muss das System sowohl eine hohe Konsistenz als auch eine geringe Komplexität aufweisen (Rennenkampff et al., 2015, S. 156).

Geringe Komplexität

Hohe Komplexität gehört zu den zentralen Verhinderern von Agilität. Monolithische und historisch gewachsene Anwendungssysteme sind schwer wartbar und änderbar und können daher veränderte Geschäftsprozesse nicht mehr abbilden. Hinzu kommen viele weitere Faktoren wie der Einsatz von alten Technologien, für welche nur mit hohem Aufwand qualifizierte Mitarbeiter zu finden sind (Rennenkampff et al., 2015, S. 156).

Komplexität kann anhand der Systemtheorie beschrieben werden. In dieser wird unter einem System “eine Menge von Komponenten verstanden, die miteinander in Beziehung stehen” (Ferstl & Sinz, 2013). Es scheint intuitiv, dass ein System mit wenigen Komponenten eine geringere Komplexität als ein System mit vielen Komponenten aufweist. Die Komplexität eines Systems kann durch drei Eigenschaften bestimmt werden: die Anzahl der Elemente eines Systems, deren Abhängigkeiten zwischen ihnen und der Unterschiedlichkeit der Elemente (Dern & Jung, 2009, S. 670 ff. Schneberger & McLean, 2003, S. 217). Daraus lassen sich nach Rennkampf (2015) die zwei Unterziele “Geringe Abhängigkeiten” und “Hohe Homogenität” ableiten, wobei die Anzahl der Systemelemente in das Subziel der “Geringen Abhängigkeiten” einfließt (Rennenkampff et al., 2015, S. 157).

Geringe Abhängigkeiten: Abhängigkeiten entstehen durch die Verbindung von Teilen eines Systems durch Schnittstellen. Sie führen dazu, dass die Änderung an einem Teil des Systems andere Teile beeinflussen. Eine maximale Abhängigkeit liegt vor, wenn Anwendungssysteme paarweise miteinander verbunden sind. Eine minimale Abhängigkeit liegt dann vor, wenn keine Verbindungen existieren (Dern & Jung, 2009, S. 670–671). Je höher die

Abhängigkeiten sind, desto schwieriger wird die Umsetzung von Änderungen, da für jede Änderung die Auswirkung auf weitere Komponenten beachtet werden muss. Sind kaum Abhängigkeiten vorhanden, können Änderungen isoliert und dadurch schnell, mit geringen Kosten und risikoarm durchgeführt werden (Rennenkampff et al., 2015, S. 158).

Um geringe Abhängigkeiten zwischen den Komponenten zu erreichen, muss das System eine geringe Vernetzung aufweisen und die vernetzten Systeme müssen adäquat gekoppelt sein. Eine adäquate Kopplung erfordert, dass die Module einer Ebene untereinander möglichst gering vernetzt sind (lose Kopplung), während sie intern stark vernetzt sind (hohe Kohäsion) (Vogel, 2009, S. 133–134). Das Prinzip der adäquaten Kopplung beschreiben Aier und Dogan (2005) als eine "Verschiebung von Abhängigkeiten in Module." (Aier & Dogan, 2005, S. 117). Somit entsteht der Vorteil, dass "sofern Schnittstellen unverändert bleiben, einzelne Module verändert oder ausgetauscht werden [können], ohne Änderungen an den übrigen vornehmen zu müssen." (Aier & Dogan, 2005, S. 616–618).

Hohe Konsistenz

Eine hohe Konsistenz des Systems wird durch eine hohe Modularität und Redundanzfreiheit der Anwendung erreicht. Durch eine modulare Struktur und die Vermeidung von Redundanzen innerhalb der Anwendungslandschaft können bestehende Daten und Funktionen wiederverwendet werden. Ein System mit hoher Konsistenz zeichnet sich auch durch einen geringen Kopplungsgrad der einzelnen Komponenten aus. Beides führt zu einer hohen funktionalen Agilität (Rennenkampff et al., 2015, S. 165).

Hohe Modularität: Modularität bedeutet, dass die Aufgaben der Anwendung in granulare, gekapselte Module aufgeteilt werden (Aier & Dogan, 2005, S. 616–617; Gronau et al., 2007, S. 3; Vogel, 2009, S. 145). Modularität basiert auf dem Prinzip "Seperation of Concerns", welches besagt, dass große umfangreiche Aufgaben in kleinere und dadurch simple Aufgaben aufgeteilt werden sollen (Vogel, 2009, S. 137). Hierbei kann zwischen hoher fachlicher und technischer Modularität unterschieden werden. Eine hohe fachliche Modularität erfordert, dass eine eindeutige Anwendung anhand fachlicher Kriterien strukturiert ist. Die Struktur des Unternehmens sollte sich hierbei in der Anwendung widerspiegeln. Je stärker zwei Geschäftsprozesse miteinander gekoppelt sind, desto enger sollte auch die Kopplung der Anwendungskomponenten ausfallen (Rennenkampff et al., 2015, S. 166–170). Bei hoher technischer Modularität ist die Anwendung nach technischen Kriterien strukturiert. Hierbei sollte jedes Modul einen klaren, eindeutigen technischen Zweck haben (Rennenkampff et al., 2015, S. 166–170). Hier unterscheiden Engels et al. (2008) zwischen den vier Komponentenarten Interaktionskomponente,

Prozesskomponente, Funktionskomponente und Bestandskomponente. Damit technische Modularität gewährleistet ist, sollte eine Komponente eindeutig einer dieser vier Arten zuordenbar sein (Engels & Voß, 2008, S. 161 ff.).

Redundanzfreiheit: Das Ziel der Redundanzfreiheit ist erfüllt, wenn Daten oder Funktionen nicht mehr als einmal innerhalb der Anwendung existieren. Werden Daten oder Funktionen an mehreren Stellen benötigt, sollten diese trotzdem nur an einer Stelle vorhanden sein und werden wiederverwendet. Durch Redundanzfreiheit können Änderungen schnell und einheitlich durchgeführt werden und sind anschließend an allen Stellen, an welchen diese benötigt werden, verfügbar (Rennenkampff et al., 2015, S. 171–172). Redundanzfreiheit trägt somit entscheidend zur Agilität der Anwendungslandschaft bei (Durst, 2007, S. 109–110; Ross et al., 2006, S. 37–39).

Hierbei wird zwischen fachlicher Redundanzfreiheit und Datenredundanzfreiheit unterschieden. Bei der fachlichen Redundanzfreiheit müssen fachliche Funktionen nur an genau einer Stelle der Anwendung implementiert sein. Wenn diese an einer anderen Stelle benötigt werden, dürfen sie nicht neu implementiert werden. Analog dazu verlangt die Datenredundanzfreiheit, dass Datenbestände jeweils von einem einzigen Teil der Anwendung verwaltet werden. Alle anderen Teile, welche diesen Datenbestand benötigen, müssen auf den verwaltenden Teil zurückgreifen (Rennenkampff et al., 2015, S. 173).

2.4 Interoperabilität

Nach Rennkampff et al. (2015) sind zwei der häufigsten Gestaltungsziele für agile Anwendungslandschaften Standardisierung und Interoperabilität (Aier & Dogan, 2005, S. 620; Andresen et al., 2005, S. 73; Durst, 2007, S. 108–109; Gronau et al., 2007, S. 4; Ross et al., 2006, S. 74–76). Interoperabilität ist die Fähigkeit eines Systems, mit anderen Systemen zu interagieren (Losavio et al., 2004, S. 213). Sie erfordert hohe Kompatibilität der Komponenten und Systeme innerhalb der Anwendungslandschaft. Dies wird vor allem durch die Festlegung von Standards erreicht (Gronau et al., 2007, S. 4). „Allgemein versteht man unter Standardisierung eine Vereinheitlichung von Gütern, von Produktionsmethoden oder anderem.” (Wiese, 1990, S. 3–4). Dabei kann sich die Standardisierung auf viele Bereiche erstrecken. In der vorliegenden Arbeit bezieht sich der Begriff Standardisierung ausschließlich auf betriebliche Anwendungssysteme. Standardisierung wird als Grundlage gesehen, um diese miteinander zu vernetzen. Müller (2005) unterscheidet hierbei in mehrere Standardisierungskonzepte:

1. Standardisierung von Anwendungssystemkomponenten: Dabei werden die Datenmodelle, Funktionalitäten und die Präsentationslogik standardisiert, indem die gleichen Strukturmerkmale, wie beispielsweise die Verwendung der gleichen Datentypen, verwendet werden (Müller, 2005). Des Weiteren sollten möglichst wenig unterschiedliche Technologien genutzt werden. Beim Einsatz mehrerer Technologien sollte eine Technologie dominant sein. Eine hohe Anzahl an Technologien steigert die Komplexität der Anwendungslandschaft (Ross et al., 2006, S. 74–76). Der Einsatz weniger Technologien ermöglicht die Portabilität des Systems sowie Kompatibilität. Daten können leichter zwischen Anwendungssystem ausgetauscht sowie wiederverwendet werden. Des Weiteren ist eine positive Auswirkung auf Entwicklerressourcen beobachtbar (Rennenkampff et al., 2015, S. 164).

2. Schnittstellenstandardisierung: Das Konzept setzt bei der Kommunikation von Anwendungssystemen an. Es werden einheitliche Regeln für den Informationsaustausch zwischen zwei Anwendungssystemen benötigt. Diese Regeln können sich auf alle drei Systemebenen (Präsentationsschicht, Logikschicht, Datenhaltungsschicht) beziehen und stellen eine relationale Beziehung zwischen den betroffen Elementen her (Müller, 2005, S. 60–61). Dadurch wird die Komplexität der Anwendung reduziert und Entwicklungsressourcen lassen sich bei Bedarf leicht zwischen Projekten verschieben. Des Weiteren sind Daten der Anwendung verfügbar und damit wiederverwendbar (Rennenkampff et al., 2015, S. 165). Komponenten, welche die Kommunikation und Koordination zwischen Komponenten regeln, werden *Konnektoren* genannt (Vogel, 2009, S. 199).

Gute Softwarearchitekturen reduzieren ihre Komplexität, indem eine sinnvolle Aufteilung in Teilsysteme erfolgt. Jede Komponente steht für einen abgegrenzten Bereich zur Verfügung und stellt der Außenwelt eine Menge von Diensten zu Verfügung, welche über die definierte Schnittstellen genutzt werden können (Dunkel & Holitschke, 2013, S. 11 ff.).

Hohe Homogenität: Damit ein System eine hohe Homogenität besitzt, müssen Anforderungen an die Interoperabilität, Konnektivität, Standardisierung und Portierbarkeit erfüllt sein. Um das zu erreichen, sollte ein System einheitliche Technologien und Schnittstellen verwenden (Rennenkampff et al., 2015, S. 163).

3 Forschungsmethode

Die in dieser Arbeit angewendete Forschungsmethode folgt dem gestaltungsorientierten Forschungsansatz, welcher dem englischen *Design-Science-Research*-Ansatz (DSR) entspricht (Rennenkampff et al., 2015, S. 17). Diese Begriffe werden im weiteren Verlauf der Arbeit synonym verwendet.

Design Science Research ist ein Forschungsparadigma, bei welchem der Designende Fragen beantwortet für Probleme beantwortet, indem er Artefakte erschafft und mithilfe dieser neue wissenschaftliche Erkenntnisse generiert werden können. Die entworfenen Artefakte sind sowohl nützlich als auch grundlegend für das Verständnis des Problems (Hevner & Chatterjee, 2010, S. 10). Design ist nach Peffers et al. (2007) die Schaffung einer anwendbaren Lösung für ein Problem (Peffers et al., 2007, S. 47). Nach Hevner et al. (2010) ist „Design“ sowohl ein Prozess (“eine Menge von Aktivitäten”) als auch ein Produkt (“Artefakt”) (Hevner & Chatterjee, 2010, S. 78). Der gestaltungsorientierte Forschungsansatz als methodischer Rahmen erscheint für die Beantwortung der Forschungsfrage gut geeignet. Der fehlende Einsatz ereignisgesteuerter Architekturen im BPM-Umfeld ist ein relevantes Problem aus der Unternehmenspraxis, zum anderen entspricht das Konzipieren und prototypische Umsetzen einer Softwarearchitektur einer Designtätigkeit.

Der Begriff des Artefaktes ist absichtlich weit gefasst und kann verschiedenste Formen annehmen. In dieser Arbeit ist das Artefakt eine Softwarearchitektur, von welcher Teile als Prototyp praktisch umgesetzt werden. Im Kontext dieser Arbeit soll eine ereignisgesteuerte Architektur entwickelt werden, welche im BPM-Umfeld anwendbar sein soll. Bislang wurden verschiedene Ansätze für die Durchführung von DSR vorgeschlagen. Ein prominentes Beispiel liefert Peffers et al. (2007): Dieser Ansatz umfasst sechs Prozessschritte, welche in die übergeordneten Phasen “Build” (Errichten) und “Evaluate” (Evaluieren) eingeteilt werden können, wie in Abb. 4 zu sehen ist (Peffers et al., 2007, S. 54). Der Hauptteil der Arbeit ist in diese Phasen aufgeteilt (vgl. Kapitel 4 „Errichten“ und Kapitel 5 „Evaluation“).

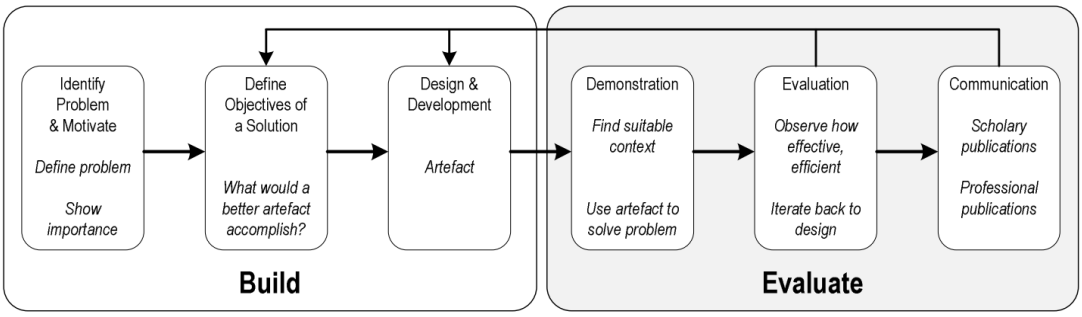

Abb. 4: Design Science Research Prozess nach (Peffers et al., 2007, S. 45–77)
(Quelle: Sonnenberg & vom Brocke, 2012, S. 72)

Aktivität 1 - Problemidentifizierung und Motivation: Ein spezifisches Forschungsproblem sowie der Wert einer potenziellen Lösung werden definiert. Die Problemdefinition wird für die Entwicklung eines Artefakts verwendet. Für die Komplexitätsreduktion kann es sinnvoll sein, das Problem in Teilprobleme aufzuteilen. Für die Problemlösung können explizite Methoden wie das Sammeln von Systemanforderungen oder eine implizite Methode wie das Programmieren und/oder Datenanalyse genutzt werden (Peffers et al., 2007, S. 52).

Aktivität 2 - Lösungsziele definieren: Die Ziele einer Lösung werden von der Problemdefinition abgeleitet. Diese werden im Kontext des Möglichen und Durchführbaren abgeleitet. Ziele können quantitativ oder qualitativ sein. Ziele sollten aus der Problemspezifikation abgeleitet werden und basieren somit auf dem vorherigen Schritt. Für die Definition der Probleme werden Kenntnisse über bisherige Lösungen sowie deren Wirksamkeit vorausgesetzt (Peffers et al., 2007, S. 55).

Aktivität 3 - Konzeption und Implementation: Dieser Schritt beinhaltet die Erstellung des Artefakts, welches der DSR-Methode zugrunde liegt. Ein Artefakt kann potenziell Modelle, Methoden oder Konstrukte enthalten. Grundsätzlich kann das DSR-Artefakt alles sein, was zur Lösung der Forschungsfrage beiträgt. Dieser Schritt umfasst die Definition der Funktionalität sowie Architektur des Artefakts und anschließend die Erstellung dieser (Peffers et al., 2007, S. 55).

Aktivität 4 - Demonstration: Die Nutzung des zuvor erstellen Artefakts wird für eines oder mehrere Probleme demonstriert. Für die Demonstration wird effektives Wissen über das Artefakt benötigt (Peffers et al., 2007, S. 55). In dieser Arbeit werden sowohl die Funktionen des Prototyps demonstriert, als auch die Möglichkeiten, welche die Gesamtarchitektur für die definierten Probleme bietet.

Aktivität 5 - Evaluation: Es wird beobachtet und evaluiert, wie gut das entwickelte Artefakt eine Lösung für das in Aktivität 1 definierte Problem bietet. Wissen über relevante Metriken sowie Analysemethoden wird vorausgesetzt. Abhängig von der Art des Problems kann die

Evaluation in verschiedenen Formen stattfinden. Es kann ein Vergleich der Funktionalität des Artefakts und anderen Lösungen in Erwägung gezogen werden. Des Weiteren können quantifizierte Parameter genutzt werden, um die Leistung des Artefakts zu messen (Peffers et al., 2007, S. 56).

Hevner et al. (2004) schlagen fünf verschiedene Evaluationsmethoden vor: beobachtende Methoden, analytische Methoden, Experimente, Testen des Artefakts und beschreibende Methoden (Hevner et al., 2004, S. 86). Für die Evaluation dieser Arbeit werden beschreibende Methoden verwendet, indem anhand informierter Argumente bewertet wird, ob das Artefakt eine Problemlösung bietet. Des Weiteren wird die Architektur des Prototyps im Zusammenhang mit der Gesamtarchitektur untersucht.

Aktivität 6 – Kommunikation: Das Problem sowie das Artefakt und sein Nutzen werden nach außen kommuniziert (Peffers et al., 2007, S. 56).

Diese Arbeit stellt in diesem Sinne die Verwirklichung der sechs Aktivitäten dar.

Hevner et al. (2004) beschreiben in einem konzeptuellen Rahmen Richtlinien für die erfolgreiche Anwendung von DSR an Informationssystemen. Diese wurden in dieser Arbeit ebenfalls angewendet und sind in Abb. 5 beschrieben (Hevner et al., 2004, S. 83).

Table 2.1 Design Science Research Guidelines

Guideline	Description
Guideline 1: Design as an Artifact	Design science research must produce a viable artifact in the form of a construct, a model, a method, or an instantiation
Guideline 2: Problem relevance	The objective of design science research is to develop technology-based solutions to important and relevant business problems
Guideline 3: Design evaluation	The utility, quality, and efficacy of a design artifact must be rigorously demonstrated via well-executed evaluation methods
Guideline 4: Research contributions	Effective design science research must provide clear and verifiable contributions in the areas of the design artifact, design foundations, and/or design methodologies
Guideline 5: Research rigor	Design science research relies upon the application of rigorous methods in both the construction and evaluation of the design artifact
Guideline 6: Design as a search process	The search for an effective artifact requires utilizing available means to reach desired ends while satisfying laws in the problem environment
Guideline 7: Communication of research	Design science research must be presented effectively to both technology-oriented and management-oriented audiences

Abb. 5: DSR-Richtlinien
(Quelle: Hevner et al., 2004, S. 83)

3.1 Design Principles

Design Principles (DP) werden als zentraler Teil der gestaltungsorientierten Forschung gesehen (Gregor & Hevner, 2013, S. 348). *Design Principles* werden charakterisiert als „Prinzipien der Form und Funktion“ sowie „Prinzipien der Implementation“ eines Artefakts (Gregor & Jones, 2007, S. 8). Sie werden genutzt um die Lücke zwischen Forschenden und Anwender zu schließen und erlauben eine präskriptive Forschung an Systemen. Sie werden also genutzt, um Wissen über das Artefakt zu erfassen (Sein et al., 2011, S. 37–56). Koppenhagen et al. (2012) schlagen vor, Design Principles zu generieren, indem Anforderungen an die Lösung gruppiert werden und daraus dann Kernanforderungen zu erstellen welche dann DPs sein können (Koppenhagen et al., 2012, S. 6).

3.2 Evaluierung von Softwarearchitekturen

Ziel dieser Arbeit ist es eine ereignisgesteuerte Softwarearchitektur zu konzipieren und teilweise umzusetzen, welche Geschäftsprozesse mittels Ereignissen entkoppelt. Um diese Architektur zu evaluieren existieren verschiedene Methoden. Die meisten dieser Methoden sind szenario-basiert und testen an bereits produktiven Systemen (Ali Babar et al., 2014, S. 5–11). Diese eignen Sie sich nicht für diese Arbeit, da nur ein Teil der Architektur prototypisch implementiert wird.

Die Evaluation von Artefakten welche Informationssysteme darstellen, ist oft eng verbunden mit Qualitätskriterien. Qualität kann in verschiedenen Variablen bewertet werden. Wenn es sich bei dem Artefakt um ein Produkt wie eine Softwarearchitektur handelt, kann ein Qualitätsmodell wie ISO 9126 genutzt werden (Pries-Heje et al., 2008, S. 8). ISO 9126 ist ein internationaler Standard für die Evaluation von Software, welches die Qualitätsmerkmale einer Software messbar macht (Pries-Heje et al., 2008, S. 8).

Das ISO 9126 Modell zur Bestimmung von Softwarequalität gliedert sich in sechs Qualitätsattribute: Funktionalität, Zuverlässigkeit, Benutzbarkeit, Effizienz, Wartbarkeit und Übertragbarkeit. Jedes dieser Attribute wird wiederum in weitere Subattribute untergliedert, welche in Abb. 6 abgebildet sind (Jeon et al., 2011, S. 207). Im Rahmen dieser Arbeit werden nicht alle Kriterien des Standards beachtet. Des Weiteren wird der Standard sinnvoll für die Evaluierung von Softwarearchitekturen angepasst.

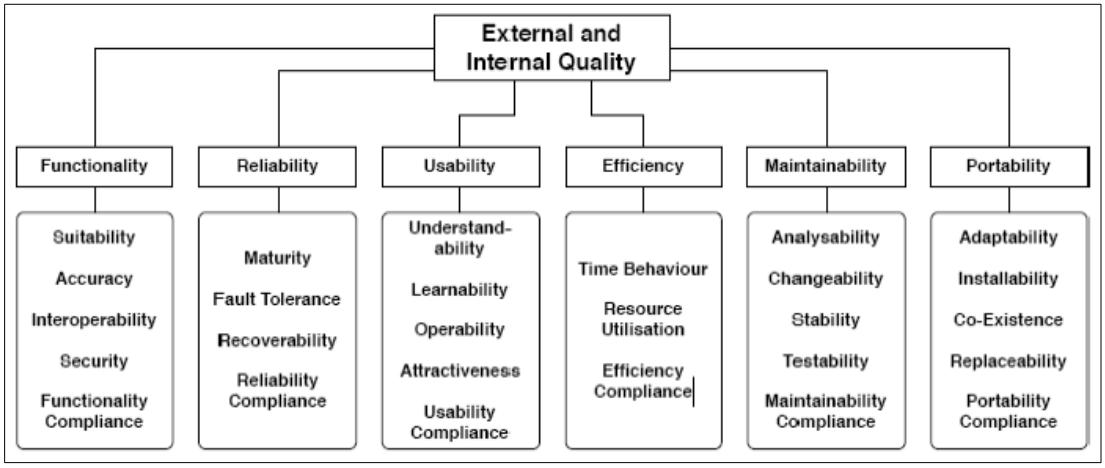

Abb. 6: Qualitätsattribute des ISO 9126 Standards
(Quelle: Jeon et al., 2011, S. 207)

Nach der ISO 9126-1 (ISO/IEC, 1998) ist Qualität definiert als eine Menge an Funktionen und Charakteristiken eines Produktes oder Services, welche in der Lage sein müssen, definierte oder angedeutete Bedürfnisse zu erfüllen. Die Qualitätsattribute sowie deren Subqualitätsattribute sind im Standard nicht konkret definiert. Deshalb müssen diese für die Evaluierung einer Softwarearchitektur konkret definiert werden (Losavio et al., 2004, S. 212). Für den Umfang dieser Arbeit ist es nicht sinnvoll, die Architektur auf alle sechs Qualitätsattribute sowie deren Subattribute zu evaluieren. Deshalb wird anhand der Problemidentifikation eine Auswahl getroffen.

Um die definierten Qualitätsattribute messbar zu machen wurde in dieser Arbeit der “Goal-Question-Metric”-Ansatz (GQM) gewählt, welcher einer der meist verbreitenden Ansätze darstellt und sich in drei Ebenen gliedert: (Basili et al., 2002, S. 3) .

1. **Konzeptionelle Ebene (Ziel):** Ein Ziel, welches meist einem Qualitätskriterium entspricht, wird für ein Objekt festgelegt.
2. **Operative Ebene (Frage):** Es werden Fragen gestellt, um zu definieren, wie ein Ziel erreicht werden kann.
3. **Quantitative Ebene (Metrik):** Eine Menge an Daten wird mit jeder Frage verbunden mit dem Ziel diese zu beantworten. Je nach Frage können die Daten objektiv oder subjektiv erhoben und bewertet werden.

Ziele, Fragen und Metriken können in einer hierarchischen Struktur dargestellt werden.

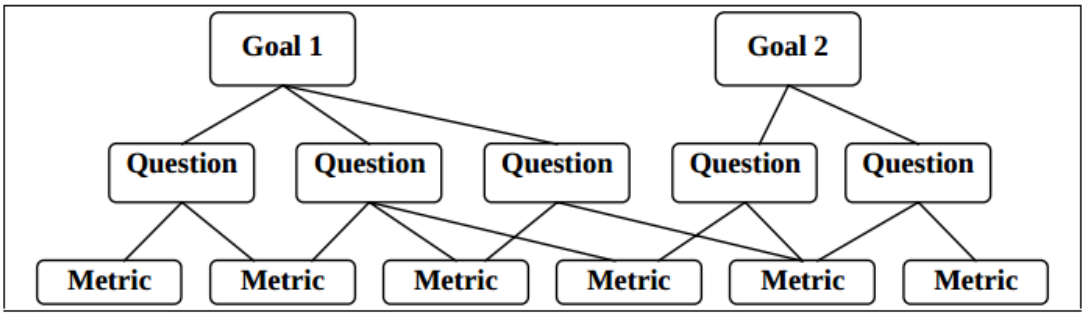

Abb. 7: Goal-Question-Metric Ansatz
(Quelle: Basili et al., o. J., S. 3)

In dieser Arbeit werden die Qualitätsattribute des ISO 1926 Standards mit dem GQM-Ansatz verbunden. Die Qualitätsattribute stellen gleichzeitig die Lösungsattribute dar und werden im zweiten Schritt des DSR Prozesses (vgl. 4.1 Problemidentifizierung) definiert.

4 Errichtung

Dieses Kapitel beinhaltet die Schritte des DSR-Prozess, welche unter „Design“ zusammengefasst werden (vgl. Kapitel 3 Forschungsmethode). Zunächst wird das Problem, welches diese Arbeit behandelt, identifiziert. Daraus ergeben sich *Design Principles*, welche später für die Evaluation des Artefakts genutzt werden. Die DPs leiten sich damit aus den Anforderungen an das Gesamtsystem ab und orientieren sich am ISO 1926 Standard. Daraufhin folgt die Entwicklung der ereignisgesteuerten Architektur sowie des Prototyps.

4.1 Problemidentifizierung

Konventionelle betriebliche IT-Systeme bieten häufig nicht die erforderliche Flexibilität, um die fachliche Agilität abzubilden, da oft eine ablauforientierte Sicht zugrunde liegt (Avery, 2019, S. 3; Bruns & Dunkel, 2010, S. 3). Grund für die Etablierung von unflexibler Software ist die sukzessive und flächendeckende Einführung solcher Systeme seit Anfang der 90er Jahre (Bungard, 2005, S. 14–21). In der Vergangenheit wurden feste Unternehmensprozesse in Software „gepresst“ (Bayer, 2016). Vor dem Hintergrund der Digitalisierung und den daraus resultierenden Veränderungen von Unternehmensprozessen stellen sich besondere Herausforderung an die Agilität von Unternehmen (Bruns & Dunkel, 2010, S. 3). Agilität ist entscheidend für die Wettbewerbsfähigkeit eines Unternehmens, da jede Unternehmensanwendung einer permanenten Veränderung ausgesetzt ist. Es wird also eine Anwendungsarchitektur benötigt, welche sich anpassen kann und modifizierbar ist. Anpassbarkeit und Modifizierbarkeit beeinflussen wesentliche Möglichkeiten eines Unternehmens, effizient auf neue Marktbedingungen zu reagieren (Bruns & Dunkel, 2010, S. 27). Die DPs für diese beiden Eigenschaften werden in dieser Arbeit wie folgt definiert:

DP1: Anpassbarkeit
Ist das Artefakt in der Lage, sich an Veränderungen im Umfeld anzupassen?

DP2: Modifizierbarkeit
Sind die Komponenten des Artefakts ohne hohen Aufwand modifizierbar?

Häufig sind starre Implementierungen von Geschäftsprozessen ein Problem, da traditionell der Fokus auf Zuständen und nicht auf Ereignissen liegt. Ereignisse spielen allerdings eine zentrale Rolle, da Prozesse in der realen Welt durch Ereignisse gesteuert werden (Bruns & Dunkel, 2010, S. 28). Veränderungen dieser eng gekoppelten Systeme sind mit großem Aufwand und Risiko verbunden. Je höher die Abhängigkeiten in einem System sind, desto schwieriger wird die Umsetzung von Änderungen, da für jede Änderung die Auswirkung auf weitere Komponenten beachtet werden muss (Rennenkampff et al., 2015, S. 158). Es ist also nicht möglich, neue Prozessschritte auszutauschen oder bestehende Prozessschritte zu verändern. Um das zu ändern, müssen die Komponenten der neuen Architektur sowohl veränderbar als auch austauschbar sein.

DP3: Austauschbarkeit
Sind die Komponenten des Artefakts austauschbar?

Lösungsansätze bieten Event-Driven Architecture und Complex Event Processing. Sie repräsentieren einen neuen Stil von betrieblichen Softwarearchitekturen, bei dem Ereignisse in das Zentrum der Softwarearchitektur rücken (Bruns & Dunkel, 2010, S. 4). Da Geschäftsprozesse auch ereignisgesteuert sind, ermöglichen diese Technologien eine realitätsnahe Abbildung der Geschäftsprozesse eines Unternehmens (Bruns & Dunkel, 2010, S. 24).

Bei der Anwendung von ereignisgesteuerten *Publish/Subscribe-Systemen* kommt es nicht selten vor, dass viele Ereignisse verarbeitet werden. Wenn Fehler den Ablauf von Geschäftsprozessen stören, können hohe Kosten entstehen (Bruns & Dunkel, 2010, S. 18). Da konventionellen Anwendungsarchitekturen meist eine ablauforientierte Sicht zugrunde liegt, sind diese Systeme zu unflexibel, um Geschäftsprozesse zu verstehen. Sie können nicht zeitnah auf Fehler reagieren. (Bruns & Dunkel, 2010, S. 8). Deshalb sollte eine neue Architektur eine adäquate Fehlererkennung sowie Fehlerbehandlung besitzen. Fehlertoleranz ist die Fähigkeit eines Systems, trotz eintretender Fehler weiter zu funktionieren (Losavio et al., 2004, S. 214).

DP4: Fehlertoleranz
Besitzt das Artefakt eine adäquate Fehlerkennung und Fehlerbehandlung? Ist es in der Lage, ein Level an Performanz zu halten, auch wenn Fehler oder Verstöße gegen Regeln vorliegen?

Trotz der Verbreitung von ereignisgesteuerten Architekturen sind trotz ausführlicher Recherche kaum relevante Anwendung im BPM-Umfeld anzutreffen (vgl. Kap. 2.1.5. Anwendungsgebiete). Dies scheint verwunderlich, dem die zuvor genannten Probleme durch den Einsatz von EDA und CEP gelöst werden könnten.

Bruns und Dunkel (2010) nennen mehre Gründe, welche bereits einleitend erwähnt wurden. Der erste mögliche Grund sind unvollständige sowie fehlende Standards. Dazu gehören vor allem fehlende technische Standards für wichtige EDA-Komponenten wie eine *Event Processing Language* zur Beschreibung von Ereignistypen und -regeln (Bruns & Dunkel, 2010, S. 234–245). Als weiterer Grund wird die Abwesenheit einer etablierten Entwicklungsmethodik genannt. Es existieren nur wenige ausgereifte Methodiken für die Entwicklung von ereignisgesteuerten Anwendungssystemen. Insbesondere Richtlinien, Entwurfsmuster, Best-Practice Beispiele, wiederverwendbare Komponenten oder Vorgehensmodelle für EDA existieren oftmals nur in rudimentärer Form (Bruns & Dunkel, 2010, S. 234–245). Unternehmen besitzen durch historisches Wachstum oft komplexe und heterogene Anwendungsarchitekturen. Ein unternehmensweiter IT-Bebauungsplan und eine IT-Strategie fehlen häufig (Aier & Dogan, 2005, S. 607–608; Österle, 1996, S. 4–5). Durch eine zunehmende Dezentralisierung der Unternehmen sowie höherer Verfügbarkeit und Übertragungsrate von Anwendungssystemen wird diese Heterogenität verstärkt. Hieraus resultiert eine mangelnde Interoperabilität der Anwendungssysteme (Müller, 2005, S. 39). Darüber hinaus sind konventionelle IT-Systeme durch ihre Rigidität und evolutionäres Wachstum oft nicht in der Lage, komplexe und interorganisationale Prozesse abzubilden (Bruns & Dunkel, 2010, S. 3). Es fehlen Schnittstellen, um mit anderen Informationssystemen zu kommunizieren. Eine neue Architektur sollte deshalb ein hohes Level an Interoperabilität bieten, wofür einer Standardisierung von Anwendungssystemen benötigt wird (Picot et al., 2007, S. 176).

DP5: Interoperabilität
Ist das Artefakt in der Lage, mit anderen Systemen zu interagieren? Nutzt es dafür gängige Standards?

Darüber hinaus werden bei der Entwicklung der Gesamtarchitektur funktionale Anforderungen definiert. Diese Anforderungen sollten von der Softwarearchitektur adäquat

erfüllt werden (Losavio et al., 2004, S. 212). Hierbei wird überprüft, ob die Software die benötigten Aufgaben erfüllen kann (Chua & Dyson, 2004, S. 186).

DP6: Tauglichkeit
Erfüllt das Artefakt alle funktionalen Anforderungen?

4.2 Entwicklung der Architektur

Die Problemidentifizierung zeigt, dass insbesondere die Aspekte der fehlenden Interoperabilität, Agilität und Fehlertoleranz bei konventionellen IT-Systemen Probleme darstellen. Für die Softwarearchitektur bietet sich daher eine ereignisgesteuerte Architektur an, welche eine standardisierte Schnittstelle darstellt und mittels Ereignissen Komponenten entkoppelt. Des Weiteren kann die Architektur durch CEP ergänzt werden, um Ereignisse zu validieren und Fehler zu erkennen.

Dadurch ergibt sich eine Architektur, welche die Komponenten mittels des *Publish/Subscribe*-Prinzips lose miteinander koppelt. Bei ereignisgesteuerten Architekturen werden Geschäftsprozesse aus sogenannten *Services* oder *Microservices* zusammengesetzt. Diese Geschäftsprozesskomponenten setzen genau eine Geschäftsfunktionalität um. Hierbei wird auch von *Kapselung* gesprochen (Bruns & Dunkel, 2010, S. 35). Diese Prinzip wird in Abb. 8 verdeutlicht.

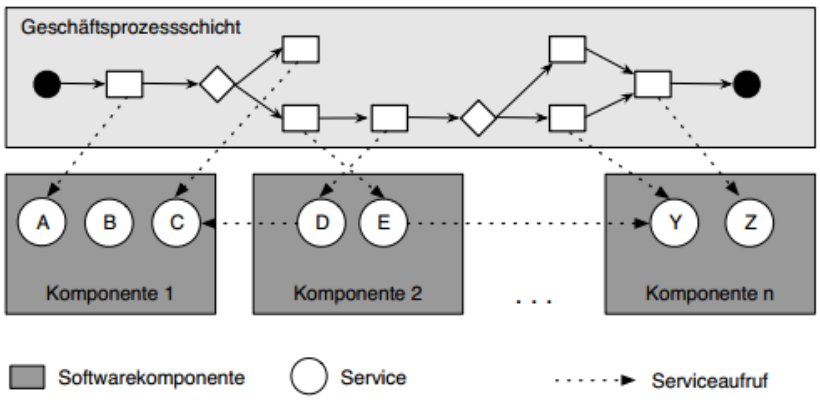

Abb. 8: Service-Orientierung nach nach Dunkel, 2008 (Quelle: (Bruns & Dunkel, 2010, S. 35)

In einem produktiven System würde jeder Prozessschritt, beispielsweise das Versenden eine E-Mail, von einer einzelnen Geschäftsprozesskomponente umgesetzt werden. Durch die ereignisgesteuerte Natur des Gesamtsystems kann jede Geschäftsprozesskomponente durch ein Ereignis ausgelöst werden und wiederum neue Ereignisse erzeugen (Bruns & Dunkel, 2010, S. 20–21). Somit lassen sich ereignisgesteuerte Geschäftsprozesse, bei

welchen sich in der Regel Prozessschritte und Ereignisse abwechseln, direkt in der Architektur abbilden. Damit eine Ereignisbehandlung stattfinden kann, wird dem System eine „Validator“-Komponente hinzugefügt. Diese prüft eintreffende Ereignisse auf Inkonsistenzen und Fehler. Anschließend erfolgt eine Aufteilung von korrekten und fehlerhaften Ereignissen in unterschiedliche *Topics*.

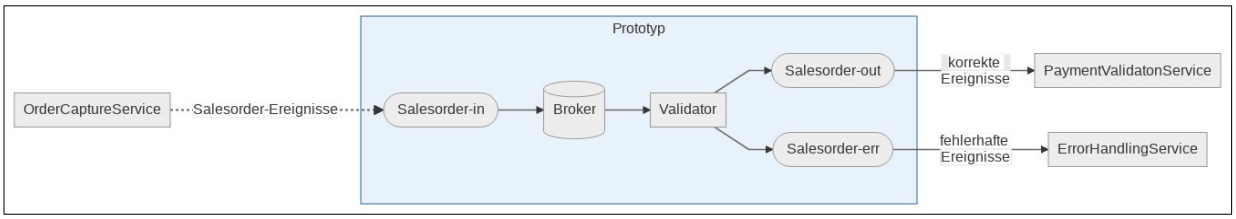

Abb. 9: Abgrenzung Prototyp und Gesamtsystem (eigene Darstellung)

In Abb. 9 und 10 ist ein beispielhafter Ausschnitt eines Geschäftsprozess zu sehen, wie er in der Architektur umgesetzt würde. Jeder Prozessschritt wird durch einen *Microservice* umgesetzt. Im Beispiel erfasst der *Microservice OrderCaptureService* Bestellungs-Ereignisse, welche anschließend vom Prototyp validiert werden. Korrekte und fehlerhafte Ereignisse werden in unterschiedliche Topics gespeichert, welche weitere Geschäftsprozesskomponenten wie einen *PaymentValidationService* oder einen *ErrorHandlingService* auslösen können.

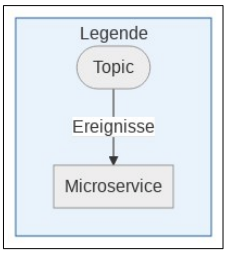

Abb. 10: Legende zu Abb.8

Als *Gesamtsystem* werden alle Komponenten inklusive Geschäftsprozesskomponenten bezeichnet. Der *Prototyp* umfasst nur die programmierte Anwendung. Das Artefakt des gestaltungsorientierten Forschungsansatz ist das Gesamtsystem.

4.3 Beschreibung des Prototyps

Der entwickelte Ansatz erweitert das *Publish/Subscribe-Prinzip* um eine Validator-Komponente. Der entscheidende Vorteil dieses Architekturansatzes ist, dass Fehler während der Laufzeit erkannt werden und somit nicht wiederholt werden können. Folgende funktionale Anforderungen an das Artefakt wurden anhand der Problemdefinition sowie anhand Überlegungen zur Architektur definiert. Sie werden in natürlicher Sprache nach den Regeln von Rupp (2009, S. 128 ff.) formuliert:

A1 Beschreibungssprache: Zur Setup-Phase muss das System dem Anwender die Möglichkeit bieten, ereignisgesteuerte Geschäftsprozesse mit einer Beschreibungssprache zu modellieren. Der Aufbau der Anwendungssprache sollte dem Aufbau von ereignisgesteuerten Geschäftsprozessen entsprechen.

Ereignisgesteuerte Geschäftsprozesse enthalten als grundlegende Bausteine Ereignisse und Funktionen (Scheer, 2002, S. 18). Im Umfang des Prototyps sollen nur die Ereignisse sowie deren Eigenschaften umgesetzt werden. Eine Implementierung von Geschäftsprozesskomponenten ist nicht vorgesehen. Die Beschreibungssprache soll den Grundstein für eine *Event Processing Language* legen.

Ein Ereignis sollte sowohl einen Namen, sowie eine oder mehre Eigenschaften haben können. Beispielsweise könnte das Ereignis „Bestellung eingegangen“ die Eigenschaften „Name“ und „Datum“ besitzen. Hierbei ist zu beachten, dass Eigenschaften verschiedene Datentypen haben können, weshalb eine entsprechende Typisierung vorzusehen ist.

A2 Syntaxprüfung: Zur Setup-Phase muss das System Syntaxfehler der Beschreibungssprache erkennen und dem Anwender zurückmelden. Für den Umfang des Prototyps ist eine Ausgabe von Syntaxfehlern in der Entwicklerkonsole ausreichend. Da der Prototyp über keine Benutzerschnittstelle verfügt und ausschließlich über die Entwicklerkonsole gesteuert wird, ist auch hier eine Ausgabe im Terminal ausreichend.

A3 Erstellung von Kafka Komponenten: Zur Laufzeit muss das System anhand der Beschreibungssprache Ereignisse und *Topics* erzeugen. Komponenten wie *Producer* und *Consumer* sollen vorerst nicht automatisch erstellt werden.

A4 Validierung: Zur Laufzeit muss das System eintreffende Ereignisse nach festgelegten Kriterien wie die Zugehörigkeit zu einer Prozessinstanz validieren und diese entsprechend in *Outbound-Topics* oder *Error-Topics* speichern.

Somit soll das System in Lage sein, trotz fehlerhaften Ereignissen weiter zu operieren. Des Weiteren soll durch eine Speicherung von Ereignissen in einem weiteren *Topic* dafür gesorgt werden, dass keine weiteren Prozessschritte durch dieses fehlerhafte Ereignis ausgelöst werden.

A5 Änderungen im Geschäftsprozess: Während der Laufzeit soll das System in der Lage sein, Änderungen der Beschreibungssprache zu

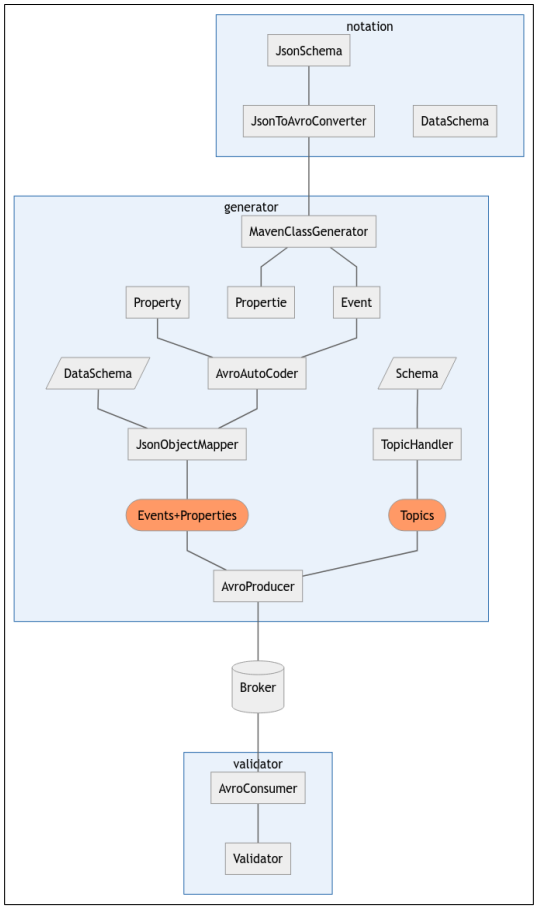

Abb. 11: Übersicht der Module und Komponenten des Prototyps (Quelle: Eigene Darstellung)

verarbeiten und die Kafka Komponenten aus der Anforderung A2 anzupassen. Werden beispielsweise weitere Prozessschritte hinzugefügt oder Eigenschaften verändert, sollten während der Laufzeit die entsprechenden Komponenten angepasst werden.

A6 Einbindung weiterer Komponenten: Es muss möglich sein, Komponenten wie Geschäftsprozesskomponenten einzubinden. Hierfür werden entsprechende Schnittstellen und eine Einhaltung von Standards benötigt. Die Implementierung des Prototyps wurde in einem mehrstufigen Prozess umgesetzt. Dafür wurde die Anwendung in mehrere Module eingeteilt, welche nachfolgend erläutert werden. In Abb. 11 sind alle Module und Komponenten des Prototyps dargestellt.

Modul Notation

Ziel des Moduls ist die Definition einer *Domain Specific Langage* (DSL) in Form einer Beschreibungssprache für Geschäftsprozesse. Ziel der Beschreibungssprache ist, das Modellieren ereignisgesteuerter Prozessabläufe zu ermöglichen. Da die Beschreibungssprache maschinenlesbar sein muss, wird diese vorerst nur in Textform umgesetzt. Die Beschreibungssprache wird ähnlich zu bereits bekannten Modellierungssprachen wie der Ereignisgesteuerten Prozesskette (EPK) sein. Diese zeichnen sich dadurch aus, dass auf ein Ereignis immer eine Funktion folgt (Scheer, 2002, S. 18).

Die Modellierung soll zwei Zwecken dienen. Zum einen sollen auf Basis der Konfiguration möglichst viele Apache-Kafka Komponenten generiert werden und zum anderen soll die Abfolge der Prozessschritte zur Laufzeit kontrolliert werden können, um die Korrektheit eintreffender Ereignisse zu überprüfen. Hierbei ist eine grafische Modellierung, wie es beispielsweise bei EPK üblich ist, nicht sinnvoll. Deshalb soll eine Beschreibungssprache in Textform genutzt werden. Als spätere Weitereinwicklung des Prototyps ist die Entwicklung einer grafischen Benutzeroberfläche zur Modellierung der Geschäftsprozesse denkbar. Im Rahmen dieses Prototyps stehen bei der Beschreibungssprache vor allem pragmatische Aspekte im Vordergrund. Diese folgt daher einem simplen Aufbau. Allerdings soll die Notation so konzipiert sein, dass komplexere Elemente später noch hinzugefügt werden können. Denkbar sind hier unter anderem logische Operatoren, mit welchen die Modellierung komplexerer Geschäftsprozesse möglich wäre.

Implementation Modul 1

Für die Syntax der Beschreibungssprache wurde die *JavaScript Object Notation* (JSON) gewählt. Dies ist ein kompaktes Datenformat, welches in der *Client-Server* Kommunikation weit verbreitet ist (*JSON*, o. J.). JSON besitzt standardmäßig nur wenige Syntaxregeln. Daher

ist es nötig, zusätzliche Regeln mittels eines sogenannten Schemas für den Aufbau festzulegen. Die *Confluent-Platform* von Kafka bietet bereits solch ein Schema namens *Avro Schema (Apache Software Foundation, 2012)*.

Um das Architekturprinzip *Seperation of Concerns* zu gewährleisten wird das Schema in zwei Teile aufgeteilt: Das *JSON-Schema* legt nur den Namen der Ereignisse sowie die Namen der Eigenschaften jedes Ereignisses fest. Das *Datenschema* hält die Daten von Ereignissen, welche beim Start der Anwendung erstellt werden. Später können natürlich weitere Ereignisse erstellt werden. Diese Aufteilung ist aus technischer Sicht notwendig, da die Erstellung der Ereignisse und die Überführung der Daten in Objekte in zwei unterschiedlichen Schritten stattfinden. Da das *JSON-Schema* sowohl dem JSON-Standard als auch den Eigenschaften des *Avro-Schemas* entsprechen muss, würde eine Zusammenlegung der Schemas bei der Verarbeitung einen höheren Aufwand bedeuten. In Abb. 12 und 13 sind Beispiele der Schemas zu sehen.

Abb. 12: JSON-Schema
(Quelle: Eigene Darstellung)

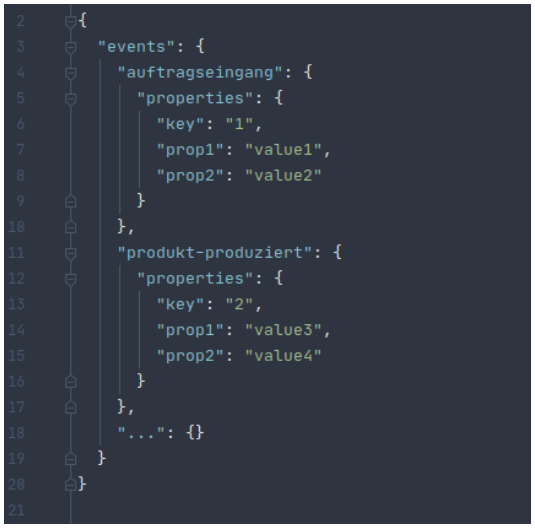

Abb. 13: Datenschema
(Quelle: Eigene Darstellung)

Mit der Beschreibungssprache lassen sich noch keine Kafka Komponenten generieren. Hierfür muss das *JSON-Schema* zu einem *Avro-Schema*, welches Kafka versteht, konvertiert werden. Die zentrale Funktion der *Confluent Schema Registry* ist die *Schema-Evolution*. Sollte ein bereits vorhandenes Schema verändert werden, findet eine automatische Versionierung statt. Damit wird eine Anpassbarkeit des zugrundeliegenden Speichermechanismus von Apache Kafka ermöglicht.

Im Prototyp ist es damit möglich, während der Laufzeit Prozessänderungen vorzunehmen. Sollten also neue Prozessschritte hinzukommen oder wegfallen, können Kafka-

Komponenten erstellt oder verändert werden, ohne dass das System neu gestartet werden muss. Um aus der bisherigen Beschreibungssprache ein *Avro-Schema* zu erzeugen, wird die Scala Bibliothek *avro4s* verwendet. Das entsprechende Modul erhält als Input das *JSON-Schema und* transformiert dieses in ein *Avro-Schema*.

Modul 2: Generator

Ziel dieses Moduls ist die Implementierung eines Generators, welcher anhand der zuvor beschriebenen Beschreibungssprache entsprechende Kafka-Komponenten erzeugt. Mit jeder neuen Prozessdefinition ändern sich unter anderem die Ereignisse. Deshalb müssen insbesondere *Topics* und Ereignisse geändert oder erstellt werden.

Damit Kafka die Ereignisse verarbeiten kann, müssen diese zuerst in Java-Objekte umgewandelt werden. Dafür muss das *Avro-Schema* zuerst in Java-Klassen umgewandelt werden, um daraus später Ereignisobjekte zu generieren. Die Umwandlung ist mit dem Build Management Tool *Apache Maven* möglich.

Aus dem Schema aus Abb. 12 würden die Java Klassen “Event” und “Property” erstellt werden, welche entsprechende Ereignisse und deren Eigenschaften repräsentieren. Eine Herausforderung hierbei ist, dass sich die generierten Klassen je nach Prozess deutlich unterscheiden. Die Anzahl der Ereignisse sowie die Anzahl der Eigenschaften eines Ereignisses ändern sich mit jeder neuen Prozessbeschreibung. Beispielsweise könnte das Ereignis “Auftragseingang” die Eigenschaften Auftragsnummer, Name und Datum haben, während das Ereignis “Rechnung beglichen” zusätzlich die Eigenschaft Rechnungsnummer besitzen könnte. Diese Variabilität der “Event” sowie “Property”-Klassen macht eine Verarbeitung mit einem statischen Java-Programm unmöglich.

Deshalb wird hier das Programmierprinzip Introspektion (engl. Reflection) verwendet. Damit kann das Programm zur Laufzeit seine eigene Struktur modifizieren (vgl. Kap. 2.1 Ereignisorientierte Softwarearchitekturen). In diesem Fall passt sich der Programmcode bei der Überführung der Daten in die Java-Objekte an die generieren Klassen während der Laufzeit an. Dies hat den Vorteil, dass der Programmcode nicht manuell angepasst werden muss und somit flexibel auf sich veränderte Klassenstrukturen reagieren kann. Die Klassen *Reflection* und *AvroAutoCoder* übernehmen diese Funktionalität.

Anschließend übernimmt der *JsonObjectMapper* das Mapping der Daten im Datenschema auf Java-Objekte, indem das *Datenschema* geparsed wird. Diese Objekte werden im folgenden *Avro-Objekte* genannt. Die generierten *Avro-Objekte* können von einem Kafka-*Producer* verarbeitet und in die entsprechenden *Topics* gespeichert werden. Hierfür werden

zur Laufzeit für jedes Ereignis drei *Topics* erstellt. Zum einen wird *Inbound-Topic* erstellt, in welchem alle bisher noch nicht validierten Ereignisse gespeichert werden. Zum anderen werden für jedes Ereignis ein *Outbound-* sowie *Error-Topic* erstellt, in welchem später korrekt oder inkorrekt validierte Ereignisse gespeichert werden.

Die Erstellung der Topics übernimmt der *TopicHandler*. Die Namen der Topics werden aus der Konfiguration entnommen und jeweils durch ein “-in”, “-out” oder “-err” erweitert. Damit ist es weiteren Komponenten möglich, diese *Topics* nur mit der Kenntnis der Schemas zu verwenden.

Modul 3: Validator

Im Validator steckt die Kernfunktionalität der Architektur. Aufgabe des Validators ist es, während der Laufzeit alle eintreffenden Ereignisse nach festgelegten Kriterien zu prüfen. Dies könnte beispielsweise die korrekte Zugehörigkeit eines Ereignisses zu einer Prozessinstanz sein. Abhängig von dem Ausgang der Validation werden die Ereignisse in unterschiedliche Topics gespeichert. Wenn beispielsweise die Konfiguration eine Verkettung aus “EreignisA” und “EreignisB” vorsieht, zur Laufzeit jedoch ein “Ereignis_B” eintritt, ohne dass im Zeitraum zuvor „EreignisA“ verarbeitet wurde, wird ein Laufzeitfehler erzeugt. In diesem Fall wird das fehlerhafte Ereignis in ein „Error-Topic“ statt in ein “Outbound-Topic“ für korrekte Ereignisse gespeichert. Da Komponenten, welche Prozessschritte implementieren, nur letztere Topics abonnieren, ist somit eine Verarbeitung von fehlerhaften Ereignissen unmöglich. Des Weiteren kann das restliche System trotz fehlerhafter Ereignisse weiter funktionieren. Fehlerhafte Ereignisse aus *Error-Topics* könnten von weiteren Komponenten verarbeitet werden und beispielsweise einer Fehlerverarbeitung zugeführt werden.

Ein *Kafka-Consumer* bezieht alle zuvor erstellen Topics und empfängt damit alle Ereignisse. Anschließend werden die Ereignisse einer logischen Prüfung unterzogen und abhängig vom Ausgang der Tests in das *Outbound-* oder *Error-Topic* gespeichert. Dies ist möglich, da jedes Ereignis in der Konfiguration eine ID erhält, welche die Position im Prozess festlegt. Zur Laufzeit verfolgt der Validator alle eintreffenden Ereignisse und versucht diese einer bestehenden Prozessinstanz zuzuordnen. Sollte es sich um ein Ereignis mit der ID „1“ handeln, wird eine neue Prozessinstanz erstellt. Alle bisher erstellen Prozessinstanzen werden in einer Mapping-Tabelle gespeichert.

4.4 Demonstration

Im Folgenden wird zunächst die Funktionalität des Prototyps anhand eines simplen Beispiels demonstriert. Danach wird in diesem Kontext auf Lösungsfähigkeit der Gesamtarchitektur eingegangen. Der Demonstration zugrunde liegt ein zweistufiger Prozess, in welchem zuerst ein Bestellungsereignis („salesorder“) und daraufhin ein Zahlungs-Ereignis („payment-validated“) erzeugt werden. Für die Visualisierung des Prototyps wurde das *Kafka-Development-Environment* genutzt.

Konfiguration

Zuerst muss das Artefakt konfiguriert werden. Hierzu benötigt der Prototyp das JSON-Schema sowie optional das Datenschema. Die Ereignisse (*Events*) des ereignisgesteuerten Prozesses sowie die Attribute (*Properties*) werden definiert. Im Datenschema können bereits initial Ereignisse erstellt werden, welche beim Start des Projekts erzeugt und validiert werden. Ein beispielhaftes *JSON-Schema* ist in Abb. 11 zu sehen. Für die Demonstration werden alle Ereignisse manuell mittels der Entwicklerkonsole erzeugt.

Laufzeit

Um Ereignisse nutzbar zu machen, müssen diese in mehreren Schritten in Java-Objekte überführt werden. Dies geschieht wie zuvor beschrieben mittels einer Serialisierung des *JSON-Schemas* zu einem *Avro-Schema*. Anschließend werden aus diesem Schema Java-Klassen mittels *Maven* erstellt. Aus diesen Klassen werden während der Laufzeit Objekte generiert, welche mit den Daten des Datenschemas befüllt werden.

Für jedes Ereignis werden die entsprechenden Topics vom *TopicHandler* erstellt. In diesem Fall werden sechs Topics für die zwei konfigurierten Ereignisse erstellt. Diese sind in Abb. 14 zu sehen.

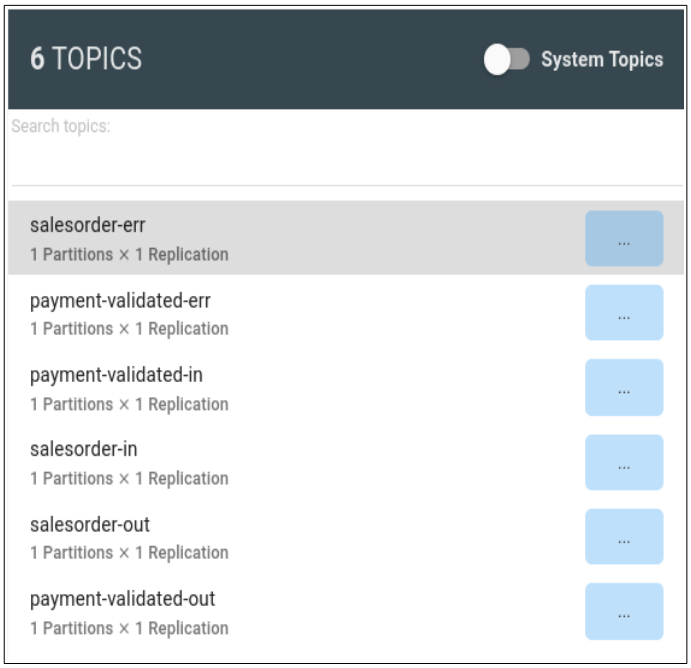

Abb. 14: Erstellte Topics

Alle eingehenden Ereignisse werden von einem Kafka-Producer in die entsprechenden *Inbound-Topics* produziert. Danach werden diese validiert und entsprechend dem Resultat in ein *Outbound-Topic* oder ein *Error-Topic* gespeichert.

Fehlerverarbeitung

Wird die korrekte Syntax des *JSON-Schemas* nicht eingehalten, so werden Fehler erzeugt. Die verwendeten Bibliotheken verfügen bereits über eine vollständige Validierung des *JSON-Scheams* und geben entsprechende Fehlermeldungen aus. Für die Demonstration werden zwei Durchläufe simuliert. Zuerst werden korrekte Ereignisse erzeugt und validiert. Hierbei handelt es sich um ein salesorder“- sowie ein „payment-validated“-Ereignis. Hierbei tritt kein Fehler auf und beide Ereignisse sind sowohl im *Inbound-Topic* und im *Outbound-Topic* zu finden.

Im zweiten Durchlauf wird absichtlich ein fehlerhaftes Ereignis erzeugt. Dies wird erreicht, indem nur ein „payment-validated“-Ereignis erzeugt wird, ohne dass ein „salesorder“-Ereignis validiert wurde. Somit wird ein Ereignis erzeugt, welches keiner Prozessinstanz zugeordnet werden kann. Neue Prozessinstanzen werden erzeugt, wenn das erste Ereignis eines Prozesses verarbeitet wird. In Abb. 15 und 16 die „payment-validated“-Ereignisse am Ende des ersten sowie des zweiten Durchlaufs zu sehen.

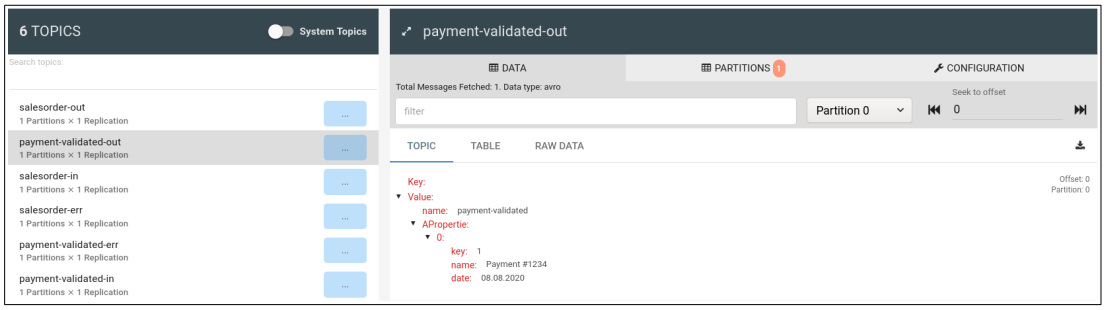

Abb. 15: Erster Durchlauf: Das Ereignis befindet sich im Outbound-Topic

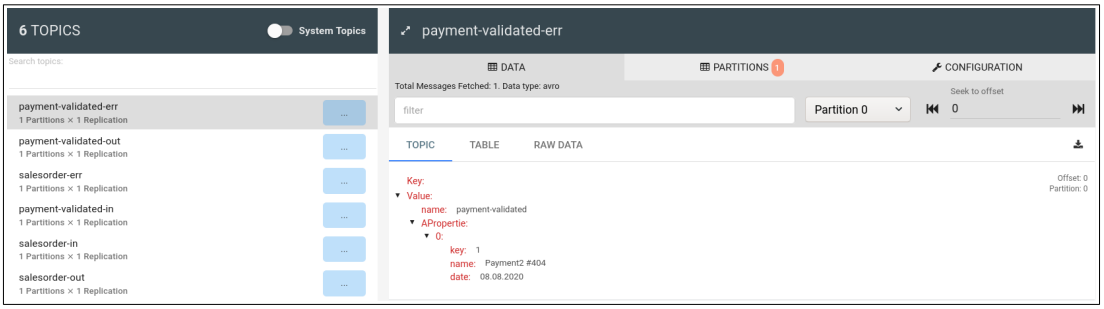

Abb. 16: Zweiter Durchlauf: Das Ereignis befindet sich im Error-Topic

Demonstration des Gesamtsystems

In der entworfenen Architektur wird jeder Prozessschritt durch eine Geschäftsprozesskomponente umgesetzt. Diese Komponenten kommunizieren nur über Ereignisse mittels des *Kafka-Clusters* miteinander. Dabei weiß eine Komponente nicht, mit welchen anderen Komponenten sie durch Ereignisse verbunden ist. Die Verbindung der Komponenten entsteht ausschließlich durch das Empfangen und Senden von Ereignisströmen. Dadurch sind die Geschäftsprozesskomponenten und damit die Geschäftsprozessschritte entkoppelt. Sollte ein neuer Prozessschritt beispielsweise am Ende des Geschäftsprozesses hinzugefügt werden, ist dies problemlos möglich. Keine der vorherigen Komponenten muss angepasst werden, die neue Komponente muss lediglich die entsprechenden *Topics* mit den gewünschten Ereignissen mittels eines *Consumers* abonnieren. Auch das Einfügen von Prozessschritten innerhalb von Prozessketten ist möglich. Hier müssen allerdings abhängig vom konkreten Geschäftsprozess nachfolgende Komponenten angepasst werden, sollten diese durch die Veränderungen nun andere Ereignisströme beziehen.

5 Evaluation

In diesem Kapitel wird die im vorherigen Kapitel konzipierte und teilweise realisierte ereignisgesteuerte Architektur kritisch untersucht. Hierbei werden sowohl das Gesamtsystem als auch der Prototyp einbezogen. Die Abgrenzung zwischen Gesamtarchitektur und Prototyp ist in Kapitel 4.2 „Entwicklung der Architektur“ dargestellt. In Tabelle 2 ist eine Übersicht der evaluierten Kriterien zu sehen. Diese sind gegliedert nach dem *Goal-Question-Metric* Ansatz.

Ziel	**Fragen**	**Metriken**
Tauglichkeit	• Erfüllt das Artefakt die funktionalen Anforderungen?	• Evaluierung jeder einzelnen funktionalen Anforderung (ja/nein)
Fehlertoleranz	• Kann der Artefakt Fehler erkennen? • Kann das Artefakt Fehler verarbeiten?	• Identifizierung von Komponenten zur Fehlererkennung und Fehlerverarbeitung
Interoperabilität	• Ist das Artefakt in der Lage, mit anderen Systemen zu interagieren? • Nutzt das Artefakt Standards, um mit anderen Kompetenten zu kommunizieren?	• Standardisierung der Anwendungskomponenten • Schnittstellenstandardisierung • Konnektoren
Modifizibarkeit	• Kann das Artefakt einfach verändert und modifiziert werden, um zum Beispiel neue Funktionalitäten zu implementieren?	• Geringe Komplexität • Verhinderung von Redundanzen innerhalb des Artefakts
Austauschbarkeit	• Besitzt das Artefakt eine hohe Konsistenz? • Sind die Komponenten des Artefakts entkoppelt und damit austauschbar? • Existieren Redundanzen im System?	• Hohe fachliche Modularität der Komponenten • Hohe technische Modularität der Komponenten • Hohe Kohäsion und lose Kopplung der Komponenten
Anpassbarkeit	• Ist das Artefakt in der Lage kapazitive und funktionale Veränderungen schnell umzusetzen?	• Adaptionsfähigkeit der Komponenten • Hohe kapazitive Agilität • Hohe Skalierbarkeit

Tabelle 1: Übersicht der Ziele, Fragen und Metriken nach dem GQM-Ansatz (Quelle: Eigene Darstellung)

5.1 DP1 Anpassbarkeit

Das Artefakt muss in der Lage sein, sich an eine ständig verändernde Umwelt anzupassen. Dafür müssen sich die Komponenten des Artefakts anpassen können. Des Weiteren muss das System auch eine kapazitive Agilität aufweisen (Rennenkampff et al., 2015, S. 175). Diese beiden Metriken sind in Abb. 17 zu sehen.

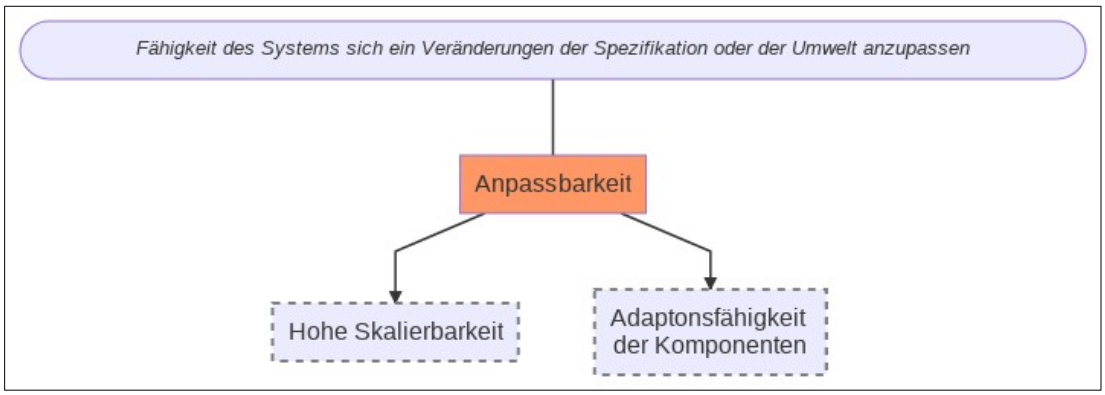

Abb. 17: Zielhierarchie Anpassbarkeit
(Quelle: Eigene Darstellung)

Adaptionsfähigkeit der Komponenten

Je nach definiertem Geschäftsprozess werden unterschiedliche Kafka-Komponenten erzeugt. Damit passt sich diese Teile des Artefakts an die Geschäftsprozesse des Unternehmens an. Dies kann durch die *Schema-Evolution* auch während der Laufzeit geschehen, ohne dass das System neu gestartet werden muss.

Einzelne Geschäftsprozesskomponenten können ebenfalls angepasst werden. Da diese weitgehend entkoppelt sind, ist das ohne erwartbare Wechselwirkungen mit anderen Komponenten möglich. Wenn eine Komponente und damit die Ereignisse verändert werden, findet eine Versionierung des *Topics* statt. Bereits verwendete *Topics* werden nicht überschrieben oder gelöscht, sondern existieren weiter. Somit können keine Ereignisse verloren gehen. Somit ist eine Adaptionsfähigkeit der Komponenten gewährleistet.

Skalierbarkeit

Publish/Subscribe-Architekturen können physikalisch verteilt und somit skaliert werden (Bruns & Dunkel, 2010, S. 74). Im Falle von Kafka können jederzeit neue *Producer* und *Consumer* hinzugefügt werden, ohne dass das System neu gestartet werden muss oder Performanzprobleme zu erwarten sind. Mit Apache Kafka nutzt das Gesamtsystem eine leistungsfähige Architektur, mit welcher durch die feinkörnige Skalierung auch komplexe Geschäftsszenarien umgesetzt werden könnten. Damit eignet sich die Architektur für Geschäftsszenarien, welche sowohl komplex als auch hochfrequent sind, beispielsweise E-Commerce oder Banking.

5.2 DP2: Modifizierbarkeit

Interne und externe Anforderungen ändern sich ständig (Avery, 2019, S. 3; Bruns & Dunkel, 2010, S. 3). Deshalb sollte das Artefakt die entsprechende Modifizierbarkeit besitzen, um eine erforderliche fachliche Agilität abzubilden. Das Artefakt sollte in der Lage sein, einfach verändert und modifiziert werden, um beispielsweise neue Funktionalität zu implementieren. Um diese Agilität zu gewährleisten wird eine geringe Komplexität des Systems benötigt (Rennenkampff et al., 2015, S. 156). Des Weiteren sollten das System Redundanzfreiheit aufweisen. Hierbei wird zwischen funktionaler Redundanzfreiheit und Datenredundanzfreiheit unterschieden (Rennenkampff et al., 2015, S. 173).

Die Design Principles „Interoparbilität“ und „Austauschbarkeit“ sind komplementär und tragen ebenfalls zur Agilität und damit zur Modifizierbarkeit des Gesamtsystems bei (Aier & Dogan, 2005, S. 616–617; Gronau et al., 2007, S. 3; Vogel, 2009, S. 145). Eine Übersicht aller Ziele ist in Abb. 18 zu sehen.

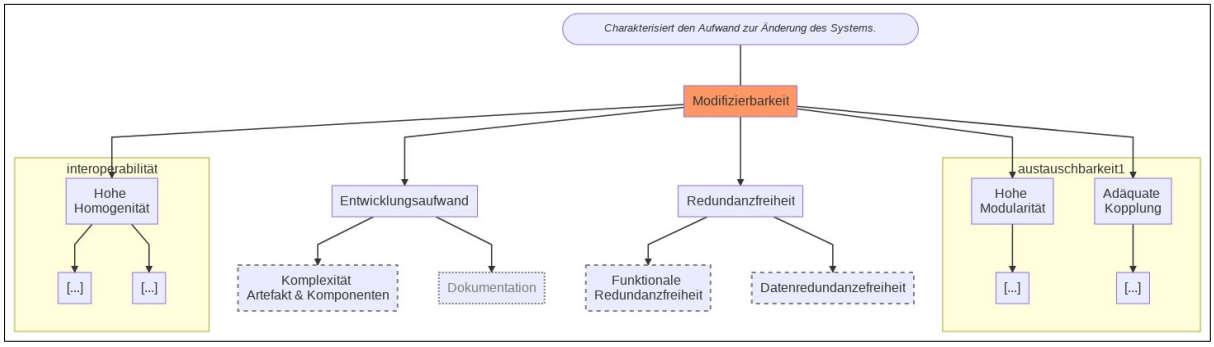

Abb. 18: Zielhierarchie Modifizierbarkeit
(Quelle: Eigene Darstellung)

Entwicklungsaufwand

Die Komplexität eines Systems kann durch drei Eigenschaften bestimmt werden: die Anzahl der Elemente eines Systems, deren Abhängigkeiten untereinander sowie die Unterschiedlichkeit der Elemente (Dern & Jung, 2009, S. 670; Schneberger & McLean, 2003, S. 217).

Die Anzahl der Elemente des Gesamtsystems wird maßgeblich durch die Geschäftsprozesse und deren Komponenten bestimmt. Komplexe, verschachtele Prozesse benötigen viele Geschäftsprozesskomponenten und resultieren daher in einem komplexeren Gesamtsystem. Die Komponenten der Module *Notation*, *Generator* und *Validator* sind allerdings immer gleich. Die Anzahl sowie Komplexität dieser Module und deren Komponenten sind überschaubar und sollte für Unternehmensanwendungen kein Problem darstellen. Die

einzige Ausnahme könnte hier die Komponente des *Validators* bilden, je nach Komplexität des Geschäftsprozesses könnte auch die Validation an Komplexität gewinnen.

Durch die Entkopplung der Geschäftsprozesskomponenten durch Ereignisse lässt sich die Vernetzung und damit die Anzahl der Abhängigkeiten zwischen Komponenten minimieren. Des Weiteren werden Prozessschritte mit Microservices umgesetzt, wobei jeder Service nur genau eine Funktionalität umsetzen sollte (Bruns & Dunkel, 2010, S. 35). Komplexere Aufgaben werden aufgeteilt, was in einer Komplexitätsreduktion resultiert (Vogel, 2009, S. 137). Das verringert den Entwicklungsaufwand im Vergleich zu monolithischen Anwendungssystemen. Des Weiteren werden von einer Komponente erzeugte Ereignisse direkt an einen *Broker* geschickt, welcher die weitere Verbreitung der Ereignisse übernimmt. Somit entfallen komplexere Schnittstellen in jeder Komponente, welche bei einer synchronen Kommunikation nötig wären. Das Artefakt bietet damit die Möglichkeit, Systeme mit einer geringen Komplexität umzusetzen.

Redundanzfreiheit

Die Erfüllung einer fachlichen Redundanzfreiheit sowie Datenredunanzfreiheit ist in weiten Teilen von der konkreten Implementation abhängig. Die Einhaltung des „Seperation of Concerns“-Pinzips und granulare Geschäftsprozesskomponenten verhindern weitestgehend fachliche Redundanzen. Geschäftsprozesskomponenten können sehr einfach wiederverwendet werden. Sollte beispielsweise eine Funktionalität in einem *Microserivce* implementiert werden und diese später in einem weiteren Prozess benötigt werden, kann die Komponente wiederverwendet werden (Rennenkampff et al., 2015, S. 164). Da die Komponenten nur genau eine Aufgabe erfüllen und Kafka als Schnittstelle nutzen ist dies unkompliziert möglich.

Die Datenredundanzfreiheit kann nur an der konkreten Umsetzung des Systems evaluiert werden. Hierbei sollte darauf geachtet werden, dass Daten nur einmal innerhalb der Anwendung existieren (Picot et al., 2007, S. 171–172). Damit ist es mit dem Artefakt möglich, redundanzfreie Systeme umzusetzen. Damit verfügt das Gesamtsystem sowohl über einen geringen Entwicklungsaufwand als auch Redundanzfreiheit. Somit ist DP2 erfüllt.

5.3 DP3 Austauschbarkeit

Das Artefakt muss die erforderliche Flexibilität bieten, um sich in einem stetig verändernden Umfeld anpassen zu können. Deshalb muss es einfach möglich sein, während des Betriebs Teile des Systems auszutauschen und damit die nötige fachliche Agilität abzubilden. Dies soll einen entscheidenden Vorteil im Vergleich zu monolithischen und historisch gewachsen

Systemen darstellen, welche keine veränderten Geschäftsprozesse mehr darstellen können (Rennenkampff et al., 2015, S. 158). Um das DP zu erfüllen, muss das Artefakt sowohl eine hohe Modularität (Aier & Dogan, 2005, S. 616–617; Gronau et al., 2007, S. 3; Vogel, 2009, S. 145) als auch eine adäquate Kopplung besitzen (Aier & Dogan, 2005, S. 616–618; Vogel, 2009, S. 133–134). Eine Hierarchie der Ziele ist in Abb. 19 zu sehen.

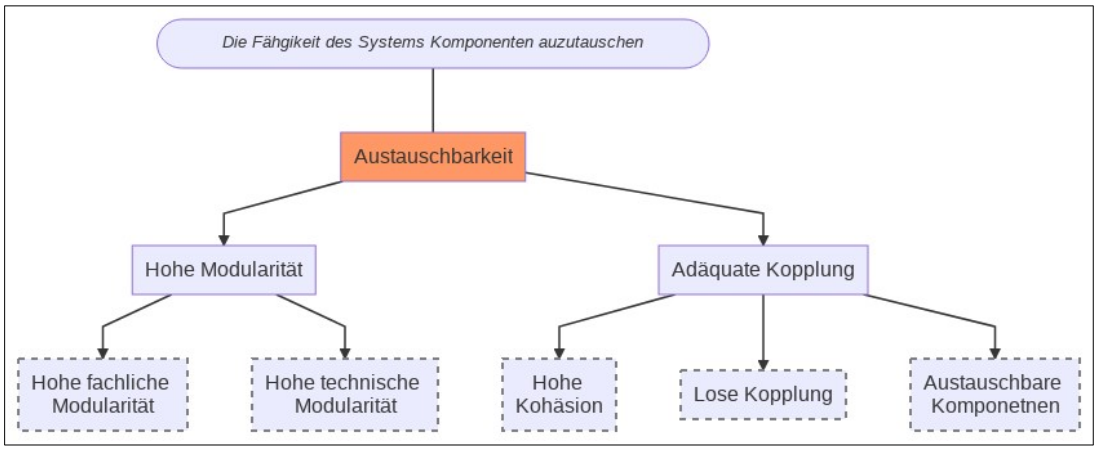

Abb. 19: Zielhierarchie Anpassbarkeit
(Quelle: Eigene Darstellung)

Adäquate Kopplung

Eine adäquate Kopplung erfordert, dass die Module einer Ebene untereinander möglichst wenig vernetzt sind (lose Kopplung), während sie intern stark vernetzt sind (hohe Kohäsion) (Vogel, 2009, S. 133–134). Im Gesamtsystem werden Geschäftsprozesskomponenten durch Ereignisse ausgelöst und können wiederum Ereignisse erzeugen. Dies geschieht über die *Publish-Subscribe*-Architektur von Apache Kafka und sorgt damit für eine lose Entkopplung der Komponenten durch Ereignisse. Geschäftsprozesskomponenten sind in der Regel intern stark vernetzt, besitzen also eine hohe Kohäsion.

Sollte eine Komponente ausfallen oder ausgetauscht werden, könnten andere Komponenten innerhalb ihres Bereiches weiterhin arbeiten.Würde beispielsweise der *PaymentValidationService* ausfallen, so würden vorherige und unabhängig nachfolgende Geschäftsprozesskomponenten wie z. B. der *OrderCapturerService* oder der *ShippingService* weiterhin funktionieren. Bei einem ausgefallenen *Service* wird der entsprechende Prozess nicht fortgeführt, bis die entsprechende Komponente wieder funktionsfähig ist.

Es ist durch den Einsatz des Prototyps problemlos möglich, Teile des Geschäftsprozesses auszutauschen, da die Geschäftsprozesskomponenten sowohl eine lose Kopplung als auch eine hohe Kohäsion aufweisen. Auf diese Weise können Prozesse einfach und risikoarm erweitert und in verteilter Umgebung verarbeitet werden.

Hohe Modularität

Durch die modulare Struktur besitzt das Gesamtsystem eine hohe funktionale Agilität (Rennenkampff et al., 2015, S. 165). Durch die Entkopplung der Komponenten durch Ereignisse und die Beschaffenheit von Microservices werden Aufgaben der Anwendungen in granulare, gekapselte Module aufgeteilt. Des Weiteren besteht ein besonderer Fokus auf dem Prinzip „Separation of Concerns“. Große umfangreiche Aufgaben werden in kleine und damit weniger komplexe Aufgaben aufgeteilt. Somit ist eine hohe technische Modularität gewährleistet, da die Komponenten nur genau eine Aufgabe übernehmen und einem eindeutigen technischen Zweck zuordenbar sind.

Fachliche Modularität erfordert eine eindeutige Strukturierung der Komponenten anhand fachlicher Kriterien (Rennenkampff et al., 2015, S. 166–170). Da jede Geschäftsprozesskomponente genau eine Geschäftsfunktonalität umsetzt, sind die Komponenten anhand fachlicher Kriterien strukturiert. Jede Komponente kann in der Regel anhand der Funktionalität einem Bereich des Unternehmens zugeordnet werden. Bei der Wiederverwendung kann es vorkommen, dass eine Komponente in mehreren Bereichen verwendet wird. Insgesamt ist damit eine hohe technische und eine hohe fachliche Modularität gewährleistet. Das Gesamtsystem verfügt sowohl über eine adäquate Kopplung als auch über eine hohe Modularität, damit ist DP3 erfüllt.

5.4 DP4: Fehlertoleranz

Um fehlertolerant zu sein, muss überprüft werden, ob das Artefakt auch in der Lage ist ein Level an Performanz zu halten, wenn Fehler oder Verstöße gegen Regeln vorliegen (Losavio et al., 2004, S. 214). Wie in Abb. 20 dargestellt liegt hierbei der Fokus auf dem Erkennen von Fehlern sowie der anschließenden Fehlerbehandlung des Systems. Es sollten Mechanismen oder Softwarebestandteile vorhanden sein, welche Fehler verarbeiten können (Chua & Dyson, 2004, S. 187).

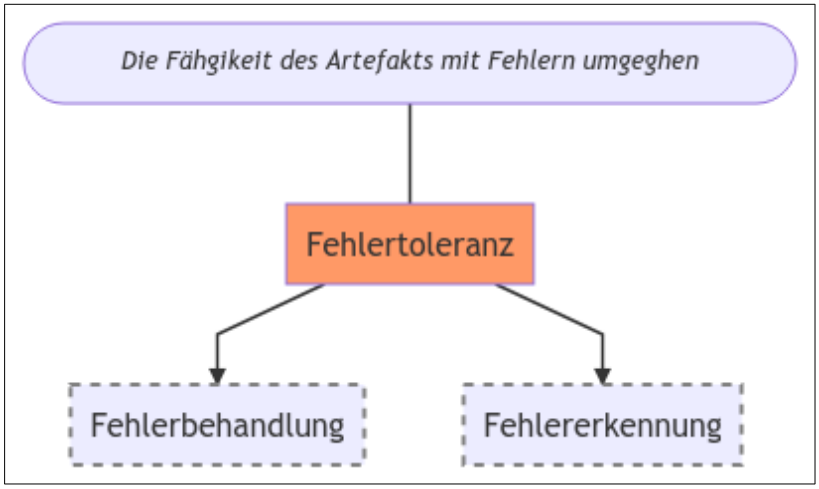

Abb. 20: Zielhierarchie Fehlertoleranz
(Quelle: Eigene Darstellung)

Während der Nutzung des Artefakts können zwei Arten von Fehlern auftreten: Fehler bei der Konfiguration sowie Fehler in den Ereignissen bzw. den Prozessen während der Laufzeit. Eine der Kernfunktionalitäten des Artefakts ist das Erkennen und Behandeln von diesen Fehlern (vgl. Kap. 4.3.5 Modul 3: Validator). Jedes eintreffende Ereignis wird durch den Validator auf potenzielle Fehler geprüft. Eine konkrete Fehlerbehandlung wird nicht vorgegeben, korrekte sowie fehlerhafte Ereignisse werden lediglich in das entsprechenden *Topics* gespeichert. Nutzer können daraufhin entscheiden, wie mit diesen fehlerhaften Ereignissen weiter verfahren wird. Hierfür müssen weitere Komponenten für die entsprechende Fehlerbehandlung implementiert werden, welche Ereignisse aus *Error-Topics* verarbeiten. Der entsprechende Prozess wird aufgehalten bis der Fehler behandelt wurde. Die Fehlerbehandlung ist in diesem Fall individuell und von den Geschäftsprozessen des Unternehmens abhängig. Somit ist eine Reaktion auf fehlerhafte Ereignisse in Echtzeit möglich.

Neben Fehlern in Ereignissen können auch Fehler in der Konfiguration auftreten. Hier erzeugt das Artefakt einen Fehler und verhindert die weitere Ausführung des Programms. Mit einem Fehler in der Konfiguration wäre die Architektur nicht lauffähig, deshalb muss der Fehler vom Nutzer behoben werden, bevor das Programm weiter ausgeführt werden kann. Im Prototyp werden Fehler momentan in einer Ausnahme ausgegeben. Für einen nicht-technischen Nutzer sind diese allerdings nur schwer verständlich. Für eine höhere Nutzerfreundlichkeit müssen hier verständlichere Fehler erzeugt werden. Somit sind Mechanismen oder Softwarebestandteile welche Fehler verarbeiten vorhanden. Damit ist DP4 erfüllt.

5.5 DP5: Interoperabilität

Damit DP5 erfüllt ist, muss das Gesamtsystem in der Lage sein, mit anderen Systemen zu interagieren (Losavio et al., 2004, S. 213). Im Idealfall wird dies durch die Verwendung von gängigen Standards erreicht (Rennenkampff et al., 2015, S. 163). Dafür müssen die Module und Komponenten des Artefakts eine hohe Homogenität aufweisen (Rennenkampff et al., 2015, S. 157). Dies ist, wie in Abb. 21 zu sehen, durch drei Metriken möglich. Das Gesamtsystem sollte standardisierte Anwendungskomponenten besitzen (Müller, 2005). Dazu gehört, dass eine kleine Zahl unterschiedlicher Technologien genutzt wird (Ross et al., 2006, S. 74–76). Des Weiteren sollten standardisierte Schnittstellen vorhanden sein (Müller, 2005, S. 60–61). Es sollten die Konnektoren des Systems identifiziert werden, wobei einheitliche Konnektoren zu Interoperabilität beitragen (Vogel, 2009, S. 199).

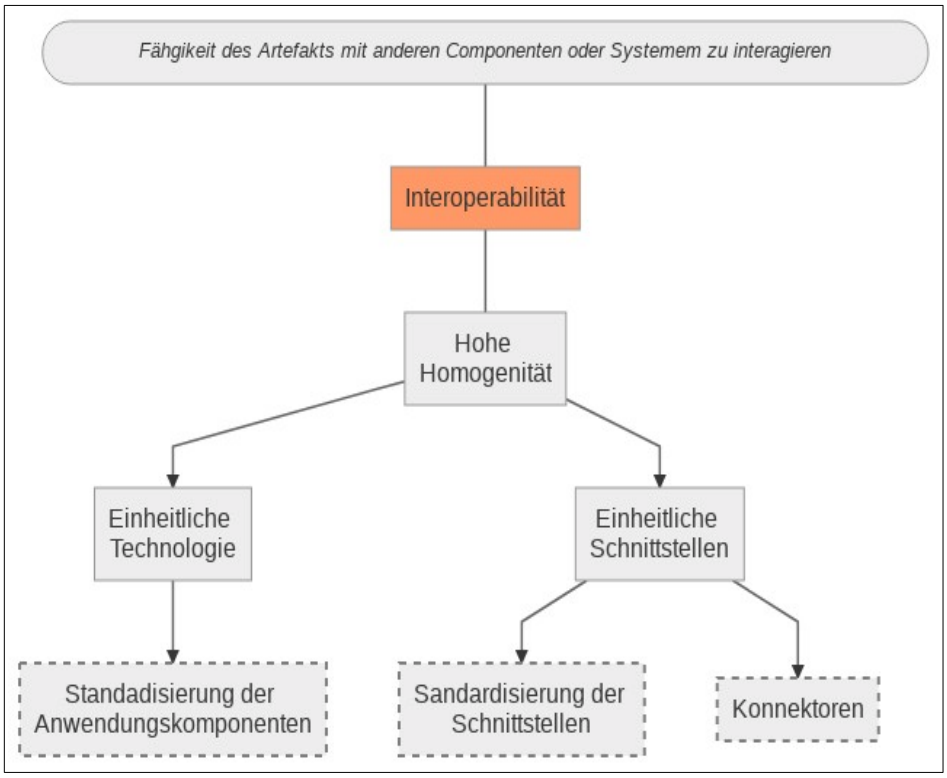

Abb. 21: Zielhierarchie Interoperabilität
(Quelle: Eigene Darstellung)

Konnektoren

Alle Prozessschritte besitzen ein auslösendes Ereignis und können selbst wiederum eine Reihe von Funktionen auslösen (Scheer, 2002, S. 18). Auf diese Weise können Prozesse einfach und risikoarm erweitert werden. Hierfür müssen in das System weitere Komponenten eingebunden werden, welche konkrete Prozessschritte implementieren. Dies sind in der Regel *Services* (Bruns & Dunkel, 2010, S. 35). Im konzipierten System ist dies durch die *Publish/Subscribe*-Architektur von Apache-Kafka möglich. Die entscheidenden Konnektoren hierbei sind die *Broker*, welche für den Austausch der Ereignisse zwischen *Producer* und *Consumer* zuständig sind. Weitere Komponenten können entsprechende *Topics* und deren Ereignisse beziehen und somit die Ereignisse weiter verwenden. Somit ist es einfach, weitere Geschäftsprozesskomponenten einzubinden. Der Prototyp demonstriert, dass Kafka sich als Konnektor für die Entkopplung von Komponenten mittels Ereignissen eignet.

Standardisierung der Anwendungskomponenten

Das Artefakt nutzt Apache Kafka, um mit Ereignissen Geschäftsprozesskomponenten zu entkoppeln. Durch diese weitreichende Entkopplung ist eine Standardisierung dieser Komponenten nur bedingt notwendig. Einzelne Komponenten könnten unterschiedlichen Technologien nutzen, ohne das Gesamtsystem zu beeinflussen (Bruns & Dunkel, 2010, S. 53;

Newman, 2015, S. 20). Solange Komponenten mit Kafka-Ereignissen umgehen können, sind sie problemlos in das System integrierbar. Beispielsweise könnte die bestellungserfassende Komponente in Java geschrieben sein, die zahlungsvalidierende Komponente in Python. Solange diese Komponenten mit Kafka-Ereignissen umgehen können, spielt die interne technische Beschaffenheit dieser eine untergeordnete Rolle.

Der Prototyp benutzt hauptsächlich Technologien im Umfeld der *Confluent Plattform*. Hierzu gehörten Apache Kafka sowie die *Schema Registry* (Confluent, Inc., 2020). Die *Schema Registry* basiert auf *JSONSchema*. Somit basieren alle Komponenten außer der Komponente *JsonToAvroConverter* auf diesen Technologien. Diese nutzt die Scala-Bibliothek *avro4s*.

Schnittstellenstandardisierung

Eine gute Softwarearchitektur zeichnet sich dadurch aus, dass Komplexität durch die Aufteilung in Teilsysteme in Module und Komponenten reduziert wird. Jede Komponente agiert in einem abgegrenzten Bereich und stellt der Außenwelt definierte Schnittstellen zur Verfügung (Dunkel & Holitschke, 2013, S. 11 ff.). Mit Apache Kafka verfügt das Gesamtsystem über eine einheitliche Schnittelle. Die Komponenten interagieren miteinander, indem eine Komponente mittels eines *Consumers* Ereignisse einer anderen Komponente aufruft. Für den Aufruf sind im Gegensatz zu Architekturen mit starker Kopplung von Komponenten keine detaillierten Kenntnisse über die anbietende Komponente notwendig. Die Komponente muss nur die Namen der Topics sowie den *Broker* kennen, in welchem das *Topic* gespeichert ist. Dabei weiß die Komponente, welche die Ereignisse erzeugt, nicht, welche weiteren Komponenten diese verarbeiten. Da durch dieses Prinzip jede Geschäftsprozesskomponente Ereignisse direkt an einen Broker sendet und dieser die weitere Verbreitung der Ereignisse steuert, ist eine asynchrone Kommunikation möglich. Komponenten müssen nicht auf eine Antwort warten, sondern können direkt nach dem Absenden des Ereignisses fortfahren (Bruns & Dunkel, 2010, S. 53).

Die einzige in der Gesamtarchitektur verwendete Schnittstelle ist Apache Kafka. Somit handelt es sich um eine hohe Schnittstellenstandardisierung. Die Verwendung einer Beschreibungssprache könnte eine weitere Schnittstelle darstellen. Weitere Komponenten könnten die Schemas ebenfalls verwenden und erweitern. Mit *JSON* wird hierbei ein übliches Format verwendet, welches viele weitere Systeme verstehen und verwenden können. Alle ereignisauslösenden Komponenten nutzen Apache Kafka als Schnittstelle, allerdings unterscheidet sich die Struktur der Ereignisse. Hier bietet die Beschreibungssprache eine Lösung, da in dieser alle Ereignisse einheitlich definiert werden. Andere Komponenten können die Beschreibungssprache nutzen, um Informationen über

Ereignisse zu erlangen und diese verarbeiten zu können. Somit ist eine Schnittstellenstandardisierung vorhanden. Das Gesamtsystem erfüllt alle drei Metriken und damit DP5.

5.6 DP6: Tauglichkeit

Um die Tauglichkeit des Artefakts zu bestimmen, muss überprüft werden, ob die funktionalen Anforderungen erfüllt werden (Picot et al., 2007) und ob die Software die definierten Aufgaben erfüllen kann (Chua & Dyson, 2004, S. 186). Die Anforderungen an das Artefakt wurden in Teil 4.3 „Beschreibung des Prototyps“ definiert.

A1 Beschreibungssprache: Das Artefakt bietet die Möglichkeit, ereignisgesteuerte Geschäftsprozesse anhand einer Beschreibungssprache zu modellieren. Diese ist in Textform umgesetzt, damit sie weiter verarbeitet werden kann. Mit der Beschreibungssprache ist es möglich, Geschäftsprozesse zu modellieren. Während der Laufzeit werden Aktivitäten durch Geschäftsprozesskomponenten und Ereignisse durch Kafka-Ereignisse umgesetzt. Die Struktur der Beschreibungssprache ist vergleichbar mit bereits bekannten Modellierungssprachen wie EPK. Hierbei wird die Struktur von ereignisgesteuerten Geschäftsprozessen eingehalten, bei welcher abwechselnd Ereignissen und Aktivitäten aufeinander folgen (Scheer, 2002, S. 18). Bei der Modellierung könnten auch logische Operationen wie „Oder“ oder „Und“ zum Einsatz kommen, um komplexe Geschäftsprozesse darstellen zu können. Anhand der Beschreibungssprache werden während der Laufzeit Kafka-Komponenten erstellt. Damit wäre es möglich, Ereignisregeln zu definieren, was die Beschreibungssprache zu einer *Event Processing Langua*ge machen würde.

Im Prototyp sind Geschäftsprozesse in Form eines *JSON-Schemas* modellierbar. Mit diesem sind momentan simple Prozesse möglich. Des Weiteren gibt es noch keine Typisierung, weshalb die Datentypen der Ereignisse bisher nur Text enthalten können. Der Prototyp zeigt aber, dass es technisch problemlos möglich wäre, alle Funktionalitäten wie logische Operatoren oder mehre Datentypen zu implementieren. Das Gesamtsystem erfüllt die Anforderung. Im Prototyp ist die Anforderung ebenfalls teilweise umgesetzt.

A2 Syntaxprüfung: Falls der Anwender Fehler bei der Konfiguration erzeugt, sollten im Gesamtsystem entsprechende Fehlermeldungen ausgegeben werden. Da ein System mit einer fehlerhaften Konfiguration Schaden anrichten könnte, muss hier eine weitere Ausführung vermieden werden. Des Weiteren kann eine Rückmeldung, wie genau der Fehler zustande kam, ausgegeben werden, um die Fehlerbehebung zu beschleunigen. Des Weiteren ist eine Syntaxprüfung für eintretende Ereignisse möglich. Damit könnte

beispielsweise überprüft werden, ob in einem Ereignis festgelegte Pflichtattribute vorhanden sind.

Im Prototyp sind noch keine benutzerfreundlichen Fehlermeldungen implementiert. Diese sind momentan ausschließlich in Form von „Stacktraces“ in der Konsole vorhanden, welche bei der Softwareentwicklung üblich sind. Es mit weiterem Entwicklungsaufwand möglich, Fehlermeldungen für nicht-technische Nutzer einzubauen. Fachliche Fehler im Prozess werden weder in der Setup-Phase noch zur Laufzeit überprüft. Das Artefakt erfüllt die Anforderung. Im Prototyp findet eine Syntaxprüfung der Beschreibungssprache statt, es werden aber keine lesbaren Fehler ausgegeben.

A3 Erstellung von Kafka-Komponenten: Im Gesamtsystem werden je nach Konfiguration die entsprechenden Kafka-Komponenten erstellt. Hierbei werden für jedes Ereignis drei *Topics* erstellt. Des Weiteren kann bereits die Struktur der Ereignisse, welche später verarbeitet werden sollen, festgelegt werden. Bei normalen Geschäftsprozessen ist ein *Consumer* sowie *Producer* ausreichend. Damit können bereits hunderttausende Ereignisse pro Sekunde verarbeitet werden (Kreps, 2014).

Der Prototyp erstellt *Topics*, Ereignisse und deren Eigenschaften. Die Anzahl der *Consumer* und *Producer* ist immer gleich, diese werden nicht automatisch erstellt. Im Prototyp werden alle spezifizierten Komponenten erstellt, somit erfüllt er die Anforderung.

A4 Validierung: Das Gesamtsystem ist zur Laufzeit in der Lage, eintreffende Ereignisse zu validieren. Hierbei können Ereignisse auf verschiedene Weisen validiert werden. Zum einen können Ereignisse im Zusammenhang mit anderen Ereignissen geprüft werden. Damit kann beispielsweise sichergestellt werden, dass die Reihenfolge von Prozessen eingehalten wird. Zum anderen kann auch der Inhalt der Ereignisse, wie der Name oder das Datum, geprüft werden. Somit wird sichergestellt, dass während Laufzeit keine falschen Ereignisse von anderen Komponenten weiter verarbeitet werden können.

Im Prototyp findet momentan eine Validierung der Prozessinstanzen statt. Es wird überprüft, ob vorhergehende Ereignisse vorhanden sind und somit das aktuelle Ereignis einer Prozessinstanz zugeordnet werden kann. Das Gesamtsystem erfüllt somit die Anforderung. Im Prototyp ist die Anforderung teilweise umgesetzt.

A5 Änderungen der Geschäftsprozesse: Bei Änderungen eines Geschäftsprozesses muss das System nicht neu gestartet werden. Es sind zudem auch keine Daten- oder Performanzprobleme erwartbar. Dies ist durch die *Schema Evolution* von Kafka möglich (Confluent, Inc., 2020). Diese ermöglicht, nachdem ein Schema initial erstellt wurde,

weitere Änderungen vorzunehmen und die Kafka Komponenten sowie weitere Daten automatischen anzupassen (ebd.).

Im Prototyp ist die *Schema-Evolution* vorhanden. Allerdings ist eine Veränderung des Geschäftsprozesses momentan noch nicht vollautomatisch möglich und es ist noch ein manueller Eingriff nötig. Dies ist mit zusätzlichem Entwicklungsaufwand allerdings komplett automatisierbar. Des Weiteren nutzt der Prototyp das Programmierprinzip Reflektion, weshalb sich relevante Teile des Codes zur Laufzeit anpassen können. Neue Prozessschritte am Ende eins Geschäftsprozess einzufügen ist im Gesamtsystem problemlos möglich. Sollten neue Prozessschritte innerhalb des Prozesses eingefügt werden oder sich bestehende Schritte ändern, müssen gegebenenfalls nachfolgende Prozessschritte angepasst werden, falls neue oder veränderte Ereignisse entstehen. Wie sich Prozessänderungen auf das Gesamtsystem auswirken, ist in Kapitel 4.4.3 „Demonstration des Gesamtsystems“ beschrieben. Das Gesamtsystem erfüllt die Anforderung. Im Prototyp ist die Anforderung teilweise umgesetzt.

A6 Einbindung weiterer Komponenten: Das Gesamtsystem ist in der Lage, weitere Komponenten einzubinden. Geschäftsprozesskomponenten müssen lediglich in der Lage sein, mit Kafka-Ereignissen umzugehen. Ist dies der Fall, können sie *Outbound-* oder *Error-Topics* abonnieren und die Ereignisse dieser *Topics* weiter verarbeiten.

Somit können Prozessschritte durch Geschäftsprozesskomponenten umgesetzt werden, welche durch Ereignisse ausgelöst werden und gleichzeitig auch Ereignisse erzeugen, welche wiederum weitere Aktivitäten auslösen können. Im Prototyp wurdem bisher noch keine Geschäftsprozesskomponenten eingebunden. Damit sind die in Tabelle 2 dargestellten Anforderungen erfüllt:

Anforderung	**Gesamtarchitektur**	**Prototyp**
A1 Beschreibungssprache	x	x
A2 Syntaxprüfung	x	(x)
A3 Erstellung von Kafka Komponenten	x	x
A4 Validierung	x	(x)
A5 Änderungen zur Laufzeit	x	(x)
A6 Einbindung weiterer Komponenten	x	-

Tabelle 2: Übersicht Erfüllungsgrad der Anforderungen (Quelle: Eigene Darstellung)

Das Artefakt erfüllt alle Anforderungen, somit ist DP6 erfüllt. Der Prototyp erfüllt zu teilen die Anforderungen, demonstriert jedoch, dass diese mit den gewählten Technologien problemlos umsetzbar wären.

5.7 Ergebnis und Interpretation

Im Folgenden werden die Ergebnisse zusammengefasst. Im Mittelpunk der Forschungsfrage stehen die drei Kriterien Interoperabilität, Agilität und Qualität.

Ergebnisse

Das Gesamtsystem kann sich an Veränderungen in der Umwelt anpassen. Durch die Beschreibungssprache und die Nutzung einer *Schema-Evolution* kann sich das Gesamtsystem an verändernde Geschäftsprozesse anpassen. Dies trägt zur Agilität des Gesamtsystems bei. Des Weiteren ist das Gesamtsystem durch die Nutzung von Apache Kafka skalierbar. Damit trägt die Anpassbarkeit zur Agilität des Gesamtsystems bei.

Die Komponenten des Gesamtsystems sind modifizierbar. Die Granularität der Geschäftsprozesskomponenten resultiert in eine geringeren Komplexität und damit einem geringeren Entwicklungsaufwand. Des Weiteren sind bei einer korrekten Umsetzung der Architektur keine Redundanzen im System zu erwarten. Einzelne Komponenten können problemlos angepasst werden, durch lose Kopplung besteht keine Gefahr, andere Komponenten zu beeinflussen. Die Modifizierbarkeit des Gesamtsystems trägt somit zur Agilität bei.

Die entworfene Architektur nutzt das *Publish/Subscribe*-Prinzip, welches eine asynchrone Kommunikation zwischen Komponenten erlaubt. Dies führt zur einer losen Kopplung und hohen Kohäsion der Geschäftsprozesskomponenten. Dadurch wird ein hoher Grad an Austauschbarkeit erreicht. Damit ist der Austausch von einzelnen Geschäftsprozesskomponenten möglich, ohne dass das Gesamtsystem beeinflusst wird. Darüber hinaus können auch Geschäftsprozesskomponenten hinzugefügt oder entfernt werden, was in der Demonstration gezeigt wurde. Dies ermöglicht Unternehmen, Geschäftsprozesse beliebig zu verändern. Dadurch werden typische Probleme starrer Prozessintegration gelöst. Die Austauschbarkeit trägt zu Agilität des Gesamtsystems bei.

Im Kern erweitert die Gesamtarchitektur eine serviceorientierte ereignisgesteuerte Architektur durch die Module Notation, Generator und Validator. Somit stehen neue Möglichkeiten zur Fehlererkennung und Fehlerbehandlung zur Verfügung. Mit CEP können komplexe Ereignismuster erkannt werden. Geschäftsprozesskomponenten können bei der

Verarbeitung von Ereignissen nun davon ausgehen, dass keine fehlerhaften Ereignisse enthalten sind. Die Fehlerbehandlung trägt somit zur Qualität der Geschäftsprozesse bei.

Apache Kafka als einzige Schnittstelle des Gesamtsystems erzeugt eine hohe Schnittstellenstandardisierung. Durch Kafka ist das System in der Lage, mit unterschiedlichen anderen Komponenten und Systemen zusammenzuarbeiten, dabei könnten in Geschäftsprozesskomponenten unterschiedliche Technologien verwendet werden. Im Gesamtsystem lösen Ereignisse Prozessschritte aus, welche mit Geschäftsprozesskomponenten ausgeführt werden. Dies entspricht dem Prinzip von ereignisgesteuerten Geschäftsprozessen. Somit unterstützt die zugrundeliegende Architektur die Struktur der Geschäftsprozesse des Unternehmens. Damit besitzt die Gesamtarchitektur eine hohe Interoperabilität.

Die Evaluation hat ergeben, dass die entworfene ereignisgesteuerte Architektur alle definierten funktionalen Anforderungen erfüllt. Der Prototyp zeigt, dass eine technische Umsetzung mit Apache Kafka möglich ist. Anforderungen, welche im Prototyp bisher noch nicht erfüllt sind, könnten mit zusätzlichem Entwicklungsaufwand implementiert werden. Hierbei sind keine technischen Schwierigkeiten zu erwarten.

Durch die Anpassbarkeit des Gesamtsystems ist eine performante Verarbeitung von vielen Ereignissen und großen Datenmengen in Echtzeit möglich. Da die Architektur durch Skalierung nicht an Performanz verliert, wird eine höhere Effizienz der Prozesse im Vergleich zu konventionellen Architekturen erzeugt. Ereignisse und Geschäftsprozessschritte können schnell abgearbeitet werden, wodurchAnwendungen in komplexen und hochfrequenten Geschäftsszenarien wie E-commerce und Banking möglich sind. Im Kontext der Digitalisierung sich daraus resultierenden zunehmenden Veränderung von Geschäftsprozessen benötigen Unternehmen agile Anwendungssysteme. Dies ist entscheidend für die Wettbewerbsfähigkeit des Unternehmens.

Mit Hilfe des Gesamtsystems könnten Geschäftsprozesse definiert werden und anschließend konstant Änderungen vollzogen werden. Sich verändernde Anforderungen stellen für Unternehmen, welche die Architektur anwenden, kein Problem mehr da. Wie demonstriert können Geschäftsprozesse beliebt angepasst und erweitert werden. Agile Geschäftsprozesse wären nicht mehr mit hohen Implementierungsaufwand verbunden, sondern würden den normalen Arbeitsmodus der Anwendungsarchitektur darstellen. Durch diese Anpassungsfähigkeit würde das Unternehmen eine hohe fachliche als auch technische Agilität aufweisen und könnte sich somit an verändernde Marktbedingungen anpassen.

Das Gesamtsystem besitzt eine Robustheit Fehlern gegenüber. Unternehmen könnten dadurch potenzielle Schäden durch die Verarbeitung fehlerhafte Ereignisse vermeiden. Des Weiteren können Unternehmen in Echtzeit auf fehlerhafter Ereignisse reagieren, was die Reaktionsfähigkeit des Unternehmens erhöht. Durch die ereignisgesteuerte Natur des Systems ist eine transparente Analyse von fehlgeschlagenen Prozessinstanzen möglich. Durch Anwendung von EDA wird somit die Geschäftsprozessoptimierung von Unternehmen erleichtert.

Ziel der Arbeit war es zu erforschen, wie eine ereignisorientierte Softwarearchitektur für BPM-Szenarien aussehen kann, die sowohl Interoperabilität, Agilität als auch Qualität bietet. Die *Design Principles* Anpassbarkeit, Austauschbarkeit und Modifizierbarkeit evaluieren das Kriterium Agilität. Die DPs Fehlertoleranz und Tauglichkeit evaluieren das Kriterium Qualität und das DP Interoperabilität evaluiert das gleichnamige Kriterium. Zusammenfassend konnte die entwickelte Softwarearchitektur alle drei Kriterien erfüllen. Im Prototyp wurden die meisten DPs mit Einschränkungen umgesetzt. Aus der Funktionalität des Prototyps kann geschlossen werden, dass die Erfüllung aller Kriterien technisch mit den ausgewählten Technologien EDA und CEP umsetzbar sind.

Reflexion und Diskussion

Im Rahmen der Arbeit wurde eine ereignisgesteuerte Softwarearchitektur für BPM-Szenarien entwickelt. Die Ergebnisse der Forschung haben gezeigt, dass eine Service-orientierte Architektur, welche durch die der Module Notation, Generator und Validator erweitert wird, sowohl Agilität, Interoperabilität als auch Qualität bieten kann.

Die Ergebnisse der Qualität und Interoperabilität lassen sich mit der Bewertung von EDA von Bruns & Dunkel (2010) erklären, deren Forschung ebenfalls den Einfluss von EDA auf Geschäftsprozesse behandelt. Dem Ergebnis liegt möglicherweise zugrunde, dass ähnliche Kriterien für die Agilität und Interoperabilität wie von Rennenkampff et al. (2015) gewählt wurden.

In der Arbeit gibt es einige Beschränkungen: Die prototypische Implementierung von Teilen der Architektur hat gezeigt, dass es mithilfe einer modernen Technologie wie Apache Kafka möglich ist, die konzipierte ereignisgesteuerte Architektur umzusetzen. Allerdings wurden als Teil Prototyps keine Geschäftsprozesskomponenten umgesetzt, deshalb konnte der Prototyp auch nur eingeschränkt getestet werden. Eine Methodik für die Evaluation einer Softwarearchitektur zu finden, gestaltete sich als schwierig. Die gängigen Methoden betrachten oft bereits implementierte Systeme und beziehen Stakeholder ein. Letztendlich wurde sich für die Anwendung eines Standards entschieden. Hierbei gestaltete sich die

Abgrenzung der *Design Principles* als schwierig, eine Überschneidung konnte nicht ausgeschlossen werden.

Bei der Recherche ergab sich das Bild, dass ein Großteil der wissenschaftlichen Literatur im Themenbereich EDA in einem Zeitraum von 2000-2015 verfasst wurde. Danach fanden sich vermehrt nur noch Anwendungen von EDA auf konkrete Beispiele. Dadurch fiel es schwer, die Aktualität und Relevanz von Literatur zu bewerten.

Eine Empfehlung für weitere Forschung ist daher die konkrete Umsetzung der Architektur samt *Services*, um weitere Erkenntnisse für die Tauglichkeit dieser in der Praxis zu erreichen. Des Weiteren könnte bei der Benutzerfreundlichkeit Beschreibungssprache weiter gearbeitet werden, indem beispielsweise eine Benutzerschnittstelle oder Fehlermeldungen entwickelt werden.

6 Fazit und Ausblick

Ziel der Arbeit war es, eine Softwarearchitektur zu entwickeln, welche die Kriterien Interoperabilität, Agilität und Qualität erfüllt. Durch die Entwicklung der Softwarearchitektur auf der Basis von Apache Kafka und der prototypischen Implementierung dieser lässt sich die Forschungsfrage folgendermaßen beantworten: Die entwickelte ereignisorientierte Softwarearchitektur bietet sowohl Interoperabilität, Agilität als auch Qualität.

Die Ergebnisse haben gezeigt, dass es durch die Anwendung von Event-Driven Architecture möglich ist, Geschäftsprozesse durch Ereignisse zu entkoppeln. Damit sind Softwarearchitekturen umsetzbar, welche sowohl Agilität, Interoperabilität als auch Qualität bieten. Das Konzept der ereignisgesteuerten Geschäftsprozesse auf Basis einer modernen Architektur hat das Potenzial, die konzeptionelle Lücke zwischen Geschäftsprozessen und deren zugrunde liegender Anwendungsarchitektur zu schließen. Geschäftsprozesse lassen sich mittels der Architektur dynamisch erweitern und verändern.

Die Bachelorarbeit hat einen ersten Forschungsbeitrag zur Entwicklung einer ereignisorientierten Softwarearchitektur mittels Apache Kafka beigetragen. Weiterführende Forschung könnte sich mit der Einbindung von Geschäftsprozesskomponenten beschäftigen. Eine konkrete Anwendung der Architektur im Unternehmensumfeld würde weitere Erkenntnisse ermöglichen. Darüber hinaus könnte die Beschreibungssprache weiterentwickelt werden, sodass eine Modellierung komplexer Prozesse möglich wäre als auch und sie eine vollständige EPL darstellt. In weiteren Schritten könnte die Benutzerfreundlichkeit durch entsprechende Schnittstellen erhöht werden.

Quellenverzeichnis

Aier, S., & Dogan, T. (2005). Indikatoren zur Bewertung der Nachhaltigkeit von Unternehmensarchitekturen. In *Wirtschaftsinformatik 2005* (S. 607–626). Springer.

Ali Babar, M., Brown, A. W., & Mistrík, I. (Hrsg.). (2014). *Agile software architecture: Aligning agile processes and software architectures*. Morgan Kaufmann.

Andresen, K., Gronau, N., & Schmid, S. (2005). Ableitung von IT-Strategien durch Bestimmung der notwendigen Wandlungsfähigkeit von Informationssystemarchitekturen. In *Wirtschaftsinformatik 2005* (S. 63–82). Springer.

Apache Kafka. (o. J.). Abgerufen 6. August 2020, von https://kafka.apache.org/documentation/

Apache Software Foundation. (2012). *Apache Avro™ 1.10.0 Specification*. https://avro.apache.org/docs/current/spec.html

Avery, N. (2019, Januar 31). *Journey to Event Driven – Part 1: Why Event-First Thinking Changes Everything | Confluent*. https://www.confluent.io/blog/journey-to-event-driven-part-1-why-event-first-thinking-changes-everything/

Basili, V. R., Caldiera, G., & Rombach, H. D. (2002). *THE GOAL QUESTION METRIC APPROACH*. 10.

Bayer, M. (2016). *Anwender brauchen mehr Flexibilität und Agilität: ERP in Zeiten der Digitalisierung*. https://www.computerwoche.de/a/erp-in-zeiten-der-digitalisierung,3314781

Bruns, R., & Dunkel, J. (2010). *Event-driven architecture: Softwarearchitektur für ereignisgesteuerte Geschäftsprozesse*.

Bungard, W. (2005). Einführung unternehmensweiter Standard-Software-Pakete: Eine gefährliche Gratwanderung zwischen wirtschaftlichem Höhenflug und existenzbedrohendem Absturz. In O. Kohnke & W. Bungard (Hrsg.), *SAP-Einführung mit Change Management* (S. 13–35). Gabler Verlag. https://doi.org/10.1007/978-3-322-90402-7_2

Chua, B. B., & Dyson, L. E. (2004). *Applying the ISO 9126 model to the evaluation of an e-learning system*. 7.

Churchward, G. (2008, Januar 11). *Extreme Agility? Try Event-Driven SOA*. CIO. https://www.cio.com/article/2449490/extreme-agility--try-event-driven-soa.html

Confluent, Inc. (2020, August 18). *Schema Management—Confluent Platform*. https://docs.confluent.io/current/schema-registry/index.html

Dern, G., & Jung, R. (2009). IT-Architektur-Governance auf Basis von Kennzahlen zur Komplexitätsmessung. *Controlling*, *21*(12), 669–672.

Dunkel, J. (2008). *Systemarchitekturen für Verteilte Anwendungen: Client-Server, Multi-Tier, SOA, Event-Driven Architectures, P2P, Grid, Web 2.0*. Hanser.

Dunkel, J., & Holitschke, A. (2013). *Softwarearchitektur für die Praxis*. Springer-Verlag.

Durst, M. (2007). *Wertorientiertes Management von IT-Architekturen*. Deutscher Universitätsverlag.

Emmerich, C., Döbele, M., Bauernhansl, T., Paulus-Rohmer, D., Schatz, A., & Weskamp, M. (2015). *Geschäftsmodell-Innovation durch Industrie 4.0*.

Engels, G., & Voß, M. (2008). *Quasar Enterprise–Anwendungslandschaften serviceorientiert gestalten*. Gesellschaft für Informatik e. V.

Ferstl, O. K., & Sinz, E. J. (2013). *Grundlagen der Wirtschaftsinformatik* (7., aktualisierte Aufl). Oldenbourg.

Gour, R. (2018, September 26). *Apache Kafka Use cases and Applications*. Medium. https://medium.com/@rinu.gour123/apache-kafka-use-cases-and-applications-7a5f379b8451

Gregor, S., & Hevner, A. R. (2013). Positioning and Presenting Design Science Research for Maximum Impact. *MIS Quarterly*, *37*(2), 337–355. JSTOR.

Gregor, S., & Jones, D. (2007). *THE ANATOMY OF A DESIGN THEORY*. 60.

Gronau, N., Lämmer, A., & Andresen, K. (2007). Entwicklung wandlungsfähiger Auftragsabwicklungssysteme. *Wandlungsfähige ERP-Systeme: Entwicklung, Auswahl und Methoden*, *45*.

Hamzah, M. H. I. (2020). AN EXPLORATORY STUDY FOR INVESTIGATING THE ISSUES AND CURRENT PRACTICES OF SERVICE-ORIENTED ARCHITECTURE ADOPTION. *Journal of Information and Communication Technology*, *18*(3), 273–304.

Hedtstück, U. (2017). *Complex Event Processing*. Springer Berlin Heidelberg. https://doi.org/10.1007/978-3-662-53451-9

Heger, S., Valett, L., Thim, H., Technische Hochschule Ingolstadt, Zentrum für angewandte Forschung, Business School, Ingolstadt, Deutschland, Schröder, J., & Gimpel, H. (2020).

Value Stream Model and Notation – Digitale Transformation von Wertströmen. In *WI2020 Zentrale Tracks* (S. 710–724). GITO Verlag. https://doi.org/10.30844/wi_2020_g2-heger

Hevner, A., & Chatterjee, S. (2010). *Design Research in Information Systems* (Bd. 22). Springer US. https://doi.org/10.1007/978-1-4419-5653-8

Hevner, A., March, S. T., Park, J., & Ram, S. (2004). Design Science in Information Systems Research. *2004*, 32.

Jeon, S., Han, M., Lee, E., & Lee, K. (2011). *Quality Attribute Driven Agile Development*. 203–210. https://doi.org/10.1109/SERA.2011.24

JSON. (o. J.). Abgerufen 17. August 2020, von https://www.json.org/json-de.html

Koppenhagen, N., Gaß, O., & Müller, B. (2012). *Design Science Research in Action-Anatomy of Success Critical Activities for Rigor and Relevance*.

Kreps, J. (2014, April 27). *Benchmarking Apache Kafka: 2 Million Writes Per Second (On Three Cheap Machines)*. https://engineering.linkedin.com/kafka/benchmarking-apache-kafka-2-million-writes-second-three-cheap-machines

Losavio, F., Chirinos, L., Matteo, A., Lévy, N., & Ramdane-Cherif, A. (2004). ISO quality standards for measuring architectures. *Journal of Systems and Software*, *72*(2), 209–223. https://doi.org/10.1016/S0164-1212(03)00114-6

Luckham, D., & Schulte, R. (2011). *Event Processing Glossary – Version 2.0*. Event Processing Technical Society.

Lundberg, A. (2006). Leverage complex event processing to improve operational performance. *Business Intelligence Journal*, *11*(1), 55.

Müller, A. (2005). *Wirtschaftlichkeit der Integration*. Deutscher Universitätsverlag. https://doi.org/10.1007/978-3-322-82140-9

Narkhede, N., Shapira, G., & Palino, T. (2019). *Kafka: The Definitive Guide*. 322.

Newman, S. (2015). *Building Microservices*. 473.

Nissen, V., & Mladin, A. (2009). Messung und Management von IT-Agilität. *HMD Praxis der Wirtschaftsinformatik*, *46*(5), 42–51. https://doi.org/10.1007/BF03340398

Österle, H. (1996). Integration: Schlüssel zur Informationsgesellschaft. *Middleware*, 1–23.

Peffers, K., Tuunanen, T., Rothenberger, M. A., & Chatterjee, S. (2007). A Design Science Research Methodology for Information Systems Research. *Journal of Management Information Systems*, *24*(3), 45–77. https://doi.org/10.2753/MIS0742-1222240302

Picot, A., Reichwald, R., & Wigand, R. T. (2007). Die grenzenlose Unternehmung. Information, Organisation und Management. In *Das Summa Summarum des Management* (S. 35–47). Springer.

Pries-Heje, J., Baskerville, R., & Venable, J. R. (2008). *Strategies for Design Science Research Evaluation*. 13.

Rennenkampff, A. von, Nissen, V., & Stelzer, D. (2015). *Management von IT-Agilität: Entwicklung eines Kennzahlensystems zur Messung der Agilität von Anwendungslandschaften*. Univ.-Verl. Ilmenau.

Richards, M., & Media, O. (o. J.). *Software Architecture Patterns*. 55.

Ross, J. W., Weill, P., & Robertson, D. (2006). *Enterprise architecture as strategy: Creating a foundation for business execution*. Harvard business press.

Rupp, C. (Hrsg.). (2009). *Requirements-Engineering und -Management: Professionelle, iterative Anforderungsanalyse für die Praxis* (5., aktualisierte und erw. Aufl). Hanser.

Sambamurthy, V., Bharadwaj, A., & Grover, V. (2003). Shaping agility through digital options: Reconceptualizing the role of information technology in contemporary firms. *MIS quarterly*, 237–263.

Scheer, A.-W. (2002). *ARIS — Vom Geschäftsprozess zum Anwendungssystem*. Springer Berlin Heidelberg. https://doi.org/10.1007/978-3-642-56300-3

Schneberger, S. L., & McLean, E. R. (2003). The complexity cross: Implications for practice. *Communications of the ACM*, *46*(9), 216. https://doi.org/10.1145/903893.903940

Schulte, R. W., & Inc, G. (2003). *The Growing Role of Events in Enterprise Applications*. 5.

Sein, Henfridsson, Purao, Rossi, & Lindgren. (2011). Action Design Research. *MIS Quarterly*, *35*(1), 37. https://doi.org/10.2307/23043488

Shaheen, M., Anees, T., Hussain, N., & Obaid, I. (2019). A Research on SOA in the IT Industry of Pakistan. *Proceedings of the 2019 5th International Conference on Computer and Technology Applications*, 149–154.

Shaw, M., & Garlan, D. (1996). *Software architecture: Perspectives on an emerging discipline*. Prentice Hall.

Sonnenberg, C., & vom Brocke, J. (2012). Evaluation Patterns for Design Science Research Artefacts. In M. Helfert & B. Donnellan (Hrsg.), *Practical Aspects of Design Science* (Bd. 286, S. 71–83). Springer Berlin Heidelberg. https://doi.org/10.1007/978-3-642-33681-2_7

Terry Anthony Byrd, D. E. T. (2000). Measuring the flexibility of information technology infrastructure: Exploratory analysis of a construct. *Journal of management information systems*, *17*(1), 167–208.

Vogel, O. (Hrsg.). (2009). *Software-Architektur: Grundlagen - Konzepte - Praxis* (2. Aufl). Spektrum Akad. Verl.

Wiese, H. (1990). *Netzeffekte und Kompatibilität: Ein theoretischer und simulationsgeleiteter Beitrag zur Absatzpolitik für Netzeffekt-Güter*. C.E. Poeschel.

Anlagen

1. Code des Prototyps

Der Codes des Prototyps ist unter folgender URL abrufbar:

https://github.com/Philipp0205/KafkaOrchestratorAbgabe/