



Fachhochschule Münster

Fachbereich Elektrotechnik und Informatik

Masterarbeit

**zur Erlangung des akademischen Grades
Master of Science (M.Sc.)
im Studiengang Informatik**

Migration monolithischer Softwaresysteme in eine Microservice-Architektur unter Berücksichtigung der Variabilität am Beispiel einer Anwendung aus dem Lotterieumfeld

Autor: Niklas Tasler

Matrikel-Nr.: 782130

Abgabedatum: 15. Februar 2022

Erstprüfer: Prof. Dr. rer. nat. Patrick Stalljohann

Zweitprüfer: Dr. rer. nat. Eric Schreiber

Inhaltsverzeichnis

Abbildungsverzeichnis	iii
Tabellenverzeichnis	v
Listingverzeichnis	vi
1 Einleitung	1
1.1 Motivation	1
1.2 Zielsetzung	2
1.3 Aufbau der Arbeit	3
2 Hintergrund	5
2.1 Monolithische Architekturen	5
2.2 Microservices	7
2.3 Containerisierung	9
2.3.1 Docker	10
2.3.2 Kubernetes	11
3 Stand der Forschung	13
4 Migration	15
4.1 Dekompositionsstrategien	16
4.1.1 Statische Analyse	16
4.1.2 Dynamische Analyse	18
4.1.3 Kombination aus statischer und dynamischer Analyse	20
4.1.4 Modellgetriebene Analyse	21
4.1.5 Domain-driven Design	22
4.2 Prozessstrategien	24
4.2.1 Neuentwicklung	24
4.2.2 Erweiterungen	25
4.2.3 Strangler-Muster	26
5 Variabilität	28
5.1 Feature Flags	28
5.2 Code-Duplikationen	30
5.3 Konfigurierbare Instanzen	31
5.4 Shared Libraries	32
5.5 Shared Service	33
5.6 Sidecar	34
5.7 Software-Produktlinien	35
6 Konzept	40
6.1 Bewertung der Dekompositionsstrategien	40
6.1.1 Anwendbarkeit	40

6.1.2	Tool-Unterstützung	41
6.1.3	Komplexität des Legacy-Systems	44
6.1.4	Qualität und Wert der Codebasis	45
6.2	Bewertung der Prozessstrategien	48
6.2.1	Schnittstellenstabilität	48
6.2.2	Komplexität des Legacy-Systems	50
6.2.3	Kritikalität des Legacy-Systems	51
6.2.4	Kosten	52
6.3	Bewertung der Variabilitätsstrategien	56
6.3.1	Abgrenzung	56
6.3.2	Grad der Variabilität	57
6.3.3	Wartbarkeit	59
6.3.4	Performance	61
6.3.5	Verfügbarkeit	62
6.3.6	Kosten	63
6.4	Zusammenfassung	65
7	Fallstudie EJP-CC	66
7.1	Bestehendes System	66
7.2	Bewertung der Dekomposition	68
7.2.1	Anwendbarkeit	68
7.2.2	Tool-Unterstützung	68
7.2.3	Komplexität	69
7.2.4	Qualität	69
7.2.5	Zusammenfassung	71
7.3	Bewertung des Prozesses	71
7.3.1	Schnittstellenstabilität	71
7.3.2	Komplexität	72
7.3.3	Kritikalität	72
7.3.4	Kosten	73
7.3.5	Zusammenfassung	74
7.4	Analyse und Dekomposition	74
7.4.1	Domain-driven Design	75
7.4.2	Service Cutter	77
7.4.3	Identifizierte Microservices	80
7.5	Umsetzung	81
7.6	Bewertung der Variabilität	85
7.6.1	Kombinationsdateien	85
7.6.2	Abrechnungen	88
7.7	Konzept für das Hinzufügen weiterer Spielvarianten	91
7.8	Ergebnisse	94
8	Diskussion	97
9	Fazit und Ausblick	103
	Literaturverzeichnis	105
A	Anhang	117

Abbildungsverzeichnis

2.1	Monolithische Softwarearchitektur mit zugehörigen Teams	5
2.2	Microservice-Architektur mit zugehörigen Teams	8
2.3	Docker-Architektur	10
2.4	Kubernetes-Architektur	12
4.1	Abgewandeltes Hufeisenmodell	15
4.2	Auf Basis von Trace-Logs ermittelte Dekompositionsoptionen	19
4.3	Partitionierung auf Basis des k-Means-Algorithmus	21
4.4	Migrationsprozess im Rahmen einer Neuentwicklung	25
4.5	Monolith mit ergänzenden Microservices	26
4.6	Anwendung des Strangler-Musters zur Überführung eines Legacy-Systems in eine Microservice-Architektur	27
5.1	Verarbeitung mehrerer Varianten durch einen Microservice	29
5.2	Verarbeitung mehrerer Varianten durch separate Microservices	30
5.3	Microservice-Instanzen mit verschiedenen Konfigurationen	32
5.4	Microservices mit gemeinsam genutzter Bibliothek	32
5.5	Verwendung eines gemeinsamen Services durch zwei Microservices	34
5.6	Verwendung des Sidecar-Musters durch zwei Microservices	35
5.7	Feature-Modell eines Webshops	36
5.8	Überblick über die Entwicklung von Software-Produktlinien	37
5.9	Feature-Modell mit zugehöriger ABS-Modellierung	38
6.1	Bewertungskriterien für die Auswahl einer Dekompositionsstrategie	47
6.2	Strangler-Muster im Rahmen einer Modernisierung, welche die externe Schnittstelle betrifft	49
6.3	Technische Schulden und Architekturerosion	53
6.4	Relative Kosten verschiedener Softwareentwicklungsphasen	54
6.5	Kostenanteil einzelner Wartungsaktivitäten	55
6.6	Bewertungskriterien für die Auswahl einer Prozessstrategie	56
6.7	Zusammenhang zwischen dem Grad der Variabilität und der Komplexität einzelner Codebasen unter Anwendung verschiedener Variabilitätsstrategien	59
6.8	Single-Tenant und Multi-Tenant Applikationen	63
6.9	Bewertungskriterien für die Auswahl einer Variabilitätsstrategie	64
6.10	Bewertungskriterien für die Auswahl von Dekompositions-, Prozess- und Variabilitätsstrategien	65
7.1	Architektur des bestehenden EJP-CC	66
7.2	Bewertung der Codebasis durch Structure101 und Sonargraph	70
7.3	Ausschnitt der Ergebnisse des Event Stormings	75
7.4	Ergebnisse des Event Stormings mit identifizierten Subdomänen	76

7.5	Ermittelte Microservices unter Verwendung des Leung-Algorithmus	79
7.6	Ermittelte Microservices unter Verwendung des Markov-Algorithmus	80
7.7	Architektur des EJP-CC auf Basis von Microservices	81
7.8	Microservice-Instanzen für die Verarbeitung von Eurojackpot- sowie Lotto 6aus49 Kombinationsdateien	92
7.9	Separate Microservices für die Verarbeitung von Eurojackpot- sowie Lotto 6aus49 Abrechnungen	94
8.1	Bewertungskriterien für die Auswahl von Dekompositions-, Prozess- und Variabilitätsstrategien nach Erkenntnisgewinn der Fallstudie	101
A.1	Kopplungskriterien des Tools Service Cutter	117
A.2	Zyklische Abhängigkeitsgruppe in bestehender Codebasis	120

Tabellenverzeichnis

6.1	Dekompositionsstrategien und benötigte Eingabedaten	42
A.1	Dekompositionsstrategien und vorhandene Tool-Unterstützung	118
A.2	Einschränkungen von Dekompositionsstrategien mit Tool-Unterstützung . .	119

Listingverzeichnis

5.1	Anfrageverarbeitung auf Basis von Feature Flags	29
5.2	Java Bean mit externer Konfiguration	31
7.1	Ausschnitt der Modellierung der bestehenden Entitäten mit Context Mapper	77
7.2	Modellierung eines Use-Cases mithilfe von Context Mapper	78
7.3	Konfiguration einer Gateway-Route	83
7.4	Verarbeitung eines Domänen-Events durch einen Microservice	84

1 Einleitung

1.1 Motivation

In der Vergangenheit waren klassische Monolithen das vorherrschende Paradigma in der Entwicklung von Softwareanwendungen. Dabei wird die gesamte Anwendung in einem Prozess bereitgestellt [1]. Da dieser Prozess alle Funktionalitäten enthält, führen auch kleine Änderungen am Quelltext dazu, dass die gesamte Anwendung neu kompiliert und bereitgestellt werden muss [2].

In den letzten Jahren erfreuen sich sogenannte Microservices steigender Beliebtheit. Dabei handelt es sich um eine neuere Architektur, bei der das gesamte System aus mehreren lose gekoppelten Services aufgebaut ist. Jeder Service soll dabei genau eine Aufgabe eigenständig erfüllen [3]. Diese Art der Softwarearchitektur verspricht unter anderem eine bessere Skalierbarkeit und schnellere Releases, da sich einzelne Services aufgrund der geringeren Komplexität leichter anpassen und bereitstellen lassen [4], [5], [6], [7].

Im Jahr 2011 wurde der Begriff Microservice erstmals auf einem Softwarearchitekturworkshop in Venedig verwendet [3]. Seit 2014 gewinnt der neue Architekturstil stetig an Bedeutung und Popularität [8], [9]. Angetrieben durch Unternehmen wie Netflix [10], SoundCloud [11], LinkedIn [12] und Amazon [13] stehen heute viele weitere Unternehmen vor der Entscheidung, ob und wie sie ihre monolithischen Anwendungen in eine Microservice-Architektur überführen sollen.

Dabei ist zu beachten, dass die Dekomposition einer monolithischen Anwendung in Microservices einen herausfordernden und komplexen Prozess darstellt [1], [14], [15]. Gleichzeitig ist die Dekomposition in der Praxis oft ein manueller Prozess [16], [17], [18], [19], welcher häufig unstrukturiert vorgenommen wird [20]. Dies stellt ein Problem dar, da eine schlechte Dekomposition dazu führt, dass die Vorteile der Microservice-Architektur nicht ausgenutzt werden können [21, S. 73 ff.].

Vor der Herausforderung einer erfolgreichen Migration steht auch WestLotto mit der Anwendung *Eurojackpot-Control-Center* (EJP-CC), welche im Rahmen dieser Arbeit betrachtet wird. Bei dem EJP-CC handelt es sich um eine klassische monolithische Software, die für

die Gesamtkoordination der Eurojackpot¹-Ziehung verantwortlich ist und unter anderem folgende Aufgaben übernimmt:

1. Sammlung und Auswertung der Tipps aller teilnehmenden Lotteriegesellschaften
2. Gewinnermittlung und Abrechnung
3. Erstellung von Reports

Im Rahmen einer anstehenden Modernisierung der Software ist für das migrierte System eine Microservice-Architektur vorgesehen.

Neben dieser technischen Änderung ist ein wichtiger Aspekt der Migration die Variabilität der Architektur. Während das EJP-CC dem Namen entsprechend aktuell ausschließlich die Verwaltung des Spiels Eurojackpot übernimmt, ist es denkbar, dass in Zukunft weitere Spielarten in sehr ähnlicher Form verarbeitet werden. Dabei ist davon auszugehen, dass ein Großteil der Funktionalität von allen Spielarten geteilt wird und zwischen der Verarbeitung verschiedener Spielarten nur wenige Unterschiede bestehen. Eine Herausforderung ist daher, Variabilitäten zu berücksichtigen und Strategien für die Nutzung der Software in weiteren Kontexten zu ermitteln.

Die Umsetzung von Variabilität ist in der Praxis häufig erforderlich [22], [23], es existieren aber insbesondere im Kontext einer Microservice-Architektur keine Richtlinien und systematischen Vorgehensweisen für die Ermittlung geeigneter Strategien.

1.2 Zielsetzung

Ziel der Arbeit ist, eine geeignete Migrationsstrategie für das EJP-CC zu entwickeln. Zu den relevanten Aspekten der Migration gehören dabei insbesondere

- Dekompositionsstrategien (Analyse und Aufteilung des Monolithen mithilfe verschiedener Techniken)
- Prozessstrategien (Vorgehensweise bei der Migration)

Die vorhandenen Methodiken und Tools werden evaluiert, wodurch eine geeignete Auswahl für die Migration des EJP-CC getroffen werden kann.

Neben der Entwicklung der Microservice-Architektur ist mit Blick auf die Zukunft insbesondere die Erweiterbarkeit ein wichtiger Aspekt. Dafür gilt es ein Konzept zu entwickeln, welches die Auswahl einer geeigneten Strategie ermöglicht, um Gemeinsamkeiten und Variabilitäten in einer Microservice-Architektur zu verwalten. Die Zielarchitektur soll somit erweiterbar

¹<https://www.eurojackpot.de/>

und flexibel sein, ohne die Vorteile von Microservices wie Wartbarkeit und Skalierbarkeit zu vernachlässigen.

Da die gesamte Migration der Software die Bearbeitungsdauer dieser Arbeit übersteigt, werden auf Basis des erarbeiteten Konzeptes prototypisch die ermittelten Microservices implementiert, um die angestrebte Architektur zu evaluieren.

In der Arbeit sollen die folgenden zentralen Forschungsfragen beantwortet werden:

1. Welche Möglichkeiten und Technologien existieren, um die Migrationen eines Monolithen in eine Microservice-Architektur zu unterstützen?
2. Welche Techniken existieren, um Gemeinsamkeiten und Variabilitäten innerhalb einer Microservice-Architektur zu verwalten?
3. Welche der identifizierten Möglichkeiten können im Falle der EJP-CC Migration sinnvoll eingesetzt werden?
4. Wie kann der Entscheidungsprozess für die Auswahl der identifizierten Technologien unterstützt werden?

1.3 Aufbau der Arbeit

Die vorliegende Arbeit gestaltet sich wie folgt:

Kapitel 2 beschreibt zunächst die theoretischen Grundlagen, die für das weitere Verständnis der Arbeit essenziell sind.

In Kapitel 3 wird ein Überblick über den aktuellen Forschungsstand gegeben, welcher den Wissensstand in Bezug auf Microservice-Migrationen und die Verwaltung von Variabilitäten in einer Microservice-Architektur zusammenfasst.

Anschließend werden in Kapitel 4 die bestehenden Möglichkeiten der Migration eines monolithischen Systems näher betrachtet. Die Analyse unterscheidet dabei zwischen Dekompositionsstrategien, welche sich auf die Identifizierung von Microservices konzentrieren, und Prozessstrategien, welche den organisatorischen Prozess der Implementierung beschreiben.

Kapitel 5 beschäftigt sich mit den Möglichkeiten der Variabilität. Dabei wird insbesondere nach Verfahren gesucht, Gemeinsamkeiten und Unterschiede in einer Microservice-Architektur zu verwalten, um so mehrere Varianten eines Produktes zu unterstützen.

Auf Basis der gesammelten Ergebnisse aus Kapitel 4 und 5 wird in Kapitel 6 ein Konzept entwickelt, welches die geeignete Auswahl von Technologien und Vorgehensweisen bei einer Migration unterstützt.

Das Konzept dient als Basis für die geeignete Auswahl von Technologien und Methodiken, die im Rahmen der Migration des EJP-CC angewendet werden. Die Anwendung des Konzeptes und die Implementierung einer prototypischen Microservice-Architektur werden in Kapitel 7 dargestellt und im Rahmen einer Diskussion in Kapitel 8 ausgewertet, bevor die Ergebnisse in Kapitel 9 zusammengefasst werden und ein Ausblick gegeben wird.

2 Hintergrund

2.1 Monolithische Architekturen

Lange Zeit wurden Softwaresysteme als Monolithe konzipiert. Diese monolithischen Softwarearchitekturen zeichnen sich dadurch aus, dass die gesamte Funktionalität in einem Prozess bereitgestellt wird. Der Monolith besteht dabei typischerweise aus drei Schichten: der Präsentationsschicht, der Geschäftslogikschicht und der Persistenz- beziehungsweise Datenschicht [24].

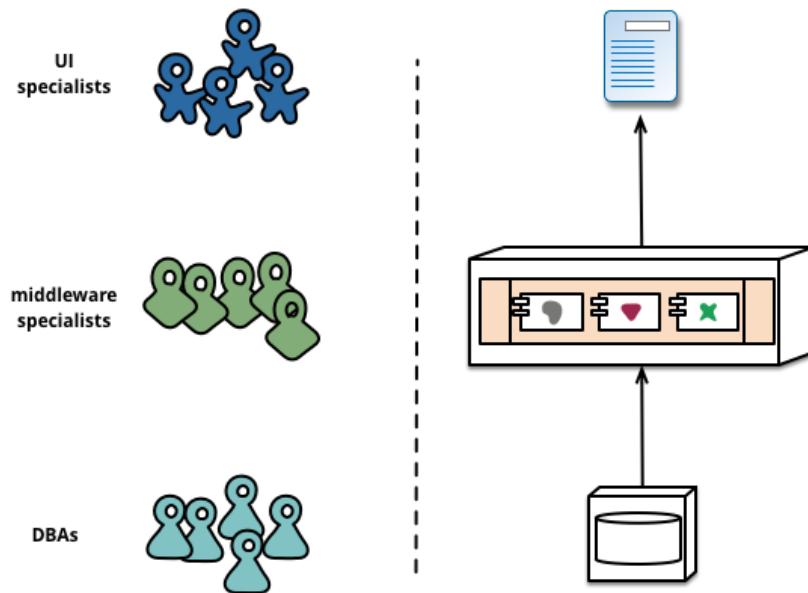


Abbildung 2.1: Monolithische Softwarearchitektur mit zugehörigen Teams [3]

Während der Begriff Monolith heute oft negativ gesehen wird, hat diese Softwarearchitektur dennoch einige Vorteile. Da die Anwendung in nur einer Programmiersprache entwickelt wird und nur eine einzige Bereitstellung notwendig ist, kann die Software schnell entwickelt werden [25]. Die Tatsache, dass nur wenige Komponenten vorhanden sind und es sich nicht um ein komplexes, weit verteiltes System handelt, sorgt auch dafür, dass sich monolithische Anwendungen leichter testen lassen [25]. Zudem ist in der Regel nur eine Datenbank vorhanden. Alle Module greifen somit auf den gleichen Stand der Daten zu und profitieren etwa

von der Transaktionssicherheit. Eine Synchronisierung der Daten zwischen verschiedenen Modulen ist somit nicht notwendig.

Demgegenüber stehen einige Nachteile, die eine monolithische Softwarearchitektur mit sich bringt:

Wartbarkeit

Mit fortschreitender Zeit wächst die Komplexität einer monolithischen Software stark. Dies kann dazu führen, dass Änderungen zeitlich aufwändig sind oder in Extremfällen gänzlich vermieden werden, da die Auswirkungen nicht abschätzbar sind [2], [26], [27].

Technologie-Lock-in

Da der Monolith ein einziger Prozess ist, sind Entwickler bezüglich Programmiersprache und Frameworks eingeschränkt [2]. Statt für jedes Teilproblem die beste Kombination aus Technologien auszuwählen, muss eine Universalauswahl getroffen werden, mit der die gesamte Anwendung implementiert wird. Insbesondere der Wechsel auf neue Technologien ist nur schwer umsetzbar [25].

Deployment

Mit wachsender Größe des Systems nimmt typischerweise auch die Startzeit zu. In großen Legacy-Systemen kann das Veröffentlichen einer neuen Anwendungsversion mehrere Stunden dauern [28], was sich nachteilig auf die kontinuierliche Bereitstellung (engl. *Continuous Deployment*) auswirkt [1], [29]. Dieser Umstand wird dadurch erschwert, dass jede Änderung eine erneute Bereitstellung der gesamten Software notwendig macht [1], [2]. Typischerweise führt das dazu, dass die Markteinführungszeit (engl. *time to market*) erhöht wird und neue Anwendungsversionen nur selten veröffentlicht werden.

Skalierbarkeit

Da die gesamte Anwendung in einem Prozess bereitgestellt wird, ist eine Skalierung nur auf Anwendungsebene möglich [1]. Einzelne Module können nicht unabhängig voneinander skaliert werden. Insbesondere vor dem Hintergrund, dass immer mehr Software in der Cloud betrieben wird [30], stellt eine Skalierung der gesamten Anwendung eine unflexible und ressourcenintensive Lösung dar, die somit für höhere Kosten sorgt [31].

Teamstruktur

Die in Abbildung 2.1 dargestellte Aufteilung der Anwendung sorgt in der Regel dafür, dass sich spezialisierte Teams ergeben. Nicht selten findet man in Unternehmen getrennte Verantwortlichkeiten für Datenbank, Backend und Frontend. Diese Art der Organisation ergibt sich nach dem Gesetz von Conway [32]:

“ Any organization that designs a system (defined broadly) will produce a design whose structure is a copy of the organization’s communication structure.

”

Melvin Conway, 1968

Änderungen an der Software sorgen dann oft dafür, dass mehrere Teams involviert sind, die sich koordinieren müssen, was wiederum die Agilität negativ beeinflusst und zu längeren Releasezyklen führt [33, S. 4].

2.2 Microservices

Im Laufe der Zeit wurde mehr Aufmerksamkeit auf die Trennung der Verantwortlichkeiten (engl. *separation of concerns*) gelegt [2]. Daraus entwickelte sich eine neue Art der Softwarearchitektur: die *serviceorientierte Architektur* (SOA). Funktionalitäten werden in einem Service gekapselt und über eine Schnittstelle anderen Komponenten zur Verfügung gestellt [2].

Diese Art der Architektur bringt einige Vorteile wie dynamische Lastverteilung, Modularität und Wiederverwendbarkeit mit sich [2]. Die Anforderungen an diese Services waren allerdings komplex und uneindeutig [2]. Dragoni et al. [2] bezeichnen sie auch als „die erste Generation von Services“. Diese Services sind typischerweise noch relativ grobgranular, kommunizieren in der Regel über einen Enterprise Service Bus [3] und sind nicht voneinander unabhängig [4], [34].

Microservices sind ein relativ neues Konzept, welches aus der SOA entstanden ist. Sie werden auch als „SOA done right“ [35] oder „die zweite Generation von Services“ [2] beschrieben. Microservices stellen eine konkrete Form der serviceorientierten Architektur dar, die versuchen einige der Probleme und Herausforderungen zu lösen [36], [37].

Es existiert keine allgemein anerkannte Definition des Begriffs Microservice [9], aber die Begriffserklärungen von James Lewis und Martin Fowler [3] sowie Sam Newman [33, S. 1] finden am häufigsten Anwendung [9]. Lewis und Fowler beschreiben die Microservice-Architektur wie folgt:

“ In short, the microservice architectural style is an approach to developing a single application as a suite of small services, each running in its own process and communicating with lightweight mechanisms, often an HTTP resource API.

”

James Lewis & Martin Fowler, 2014

Diese Art der Services zeichnet sich durch einige Charakteristiken aus:

Größe

Microservices sind feingranular und konzentrieren sich darauf, genau eine Geschäftsfähigkeit (engl. *business capability*) zu erfüllen [2].

Lose Kopplung

Microservices sollen lose gekoppelt sein. Sie werden unabhängig voneinander bereitgestellt und der Ausfall eines Microservices sorgt nicht dafür, dass das gesamte System ausfällt. Außerdem besitzt jeder Service eine eigene Datenhaltung und verfolgt das „Share nothing“ Prinzip [9]. Dies bedeutet, dass jeder Service seine eigenen Ressourcen besitzt und Aufgaben ohne Beteiligung weiterer Services bearbeiten kann.

Bounded Context

Der Bounded Context ist ein zentrales Muster des Domain-driven Designs [38], [39, S. 335 f.]. Zusammenhängende Funktionalitäten einer Domäne werden in unterschiedlichen Bounded Contexts gruppiert. Jeder Microservice implementiert dann einen Bounded Context.

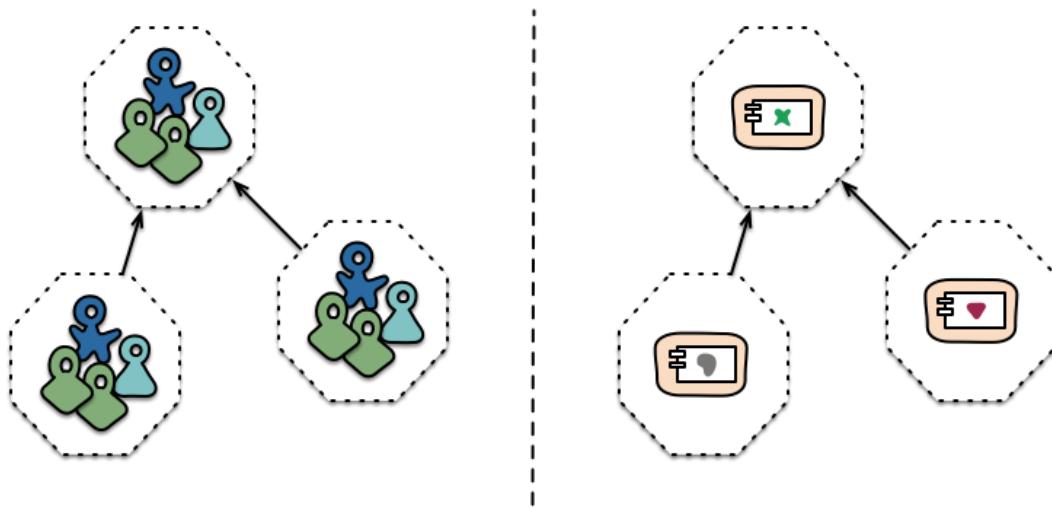


Abbildung 2.2: Microservice-Architektur mit zugehörigen Teams [3]

Abbildung 2.2 zeigt eine Microservice-Architektur bestehend aus drei Microservices mit den zugehörigen Teams. Die Microservice-Architektur berücksichtigt das Gesetz von Conway und fördert crossfunktionale Teams. Jedes Team ist für einen Microservice verantwortlich und betreut diesen von der Entwicklung über das Deployment bis hin zur Wartung in Produktion.

Diese Art der Architektur hat einige Vorteile:

Wartbarkeit

Microservices sind typischerweise feingranular und haben eine überschaubare Codebasis [2]. Da jeder Microservice eine Geschäftsfähigkeit abbildet, beschränken sich Änderungen in der Regel auf nur einen Microservice [5]. Darüber hinaus kann jeder Service unabhängig entwickelt, bereitgestellt und getestet werden [2]. Dies verringert die Kommunikation verschiedener Teams und kann so die Time-to-Market verringern [6].

Skalierbarkeit

Eine Microservice-Anwendung besteht aus mehreren einzelnen Services, die jeweils unabhängig skaliert werden können. Somit kann eine Skalierung bei Bedarf auf die Module beschränkt werden, die mehr Ressourcen benötigen. Dies stellt eine bessere Ressourcennutzung im Gegensatz zum Monolithen dar, bei dem nur die gesamte Anwendung skaliert werden kann [4], [40].

Fehlertoleranz

Eine Anwendung auf Basis der Microservice-Architektur basiert auf mehreren einzelnen Services. Jeder Service trägt einen Teil zur Funktionalität des Gesamtsystems bei. Der Ausfall eines Services hat somit zur Folge, dass nur ein kleiner Teil des Systems ausfällt und weite Teile der Anwendung weiter verwendbar sind [4].

Polyglotte Programmierung

Microservices kommunizieren über Protokolle wie HTTP, welche sich mit verschiedenen Programmiersprachen nutzen lassen. Je nach Problemstellung ist es möglich, unterschiedliche Programmiersprachen, Technologien und Datenbanken einzusetzen [2], [4].

2.3 Containerisierung

Neben Microservices ist auch die Containerisierung in den letzten Jahren immer populärer geworden [41]. Sie wird oft als natürliche Vorgehensweise für das Bereitstellen von Microservices genannt [2]. Eine Containerisierung beschreibt das Bündeln des Softwarecodes inklusive aller benötigten Abhängigkeiten wie Frameworks, Bibliotheken und Konfigurationsdateien in einem einzelnen Paket. Durch das Verpacken der Software in einem Paket wird die Anwendung samt ihrer Abhängigkeiten isoliert vom Rest des Systems ausgeführt.

Auf diese Weise kann etwa das Problem gelöst werden, dass Software sich auf unterschiedlichen Umgebungen nicht gleich verhält. Ein Beispiel dafür ist, dass Software während des Entwicklungsprozesses auf den Rechnern der beteiligten Entwickler lauffähig ist. Wird die Software in den Test übergeben und auf einer anderen Umgebung ausgeführt, kommt es

teils zu Fehlern. In diesen Fällen lässt sich oft nur schwer feststellen, ob es sich um einen Fehler in der Software oder eine falsche Konfiguration der Umgebung handelt. Durch die Isolierung der Anwendung in einem Container wird die Applikation stets in der gleichen Umgebung ausgeführt [6]. Auch das Problem nicht kompatibler Abhängigkeiten einzelner Softwarekomponenten (ugs. *dependency hell*) kann auf diese Weise gelöst werden, indem jede Komponente mit ihren Abhängigkeiten isoliert in einem eigenen Container gekapselt wird [42].

Im Gegensatz zu virtuellen Maschinen haben Container den Vorteil, dass sie performanter sind und weniger Overhead mit sich bringen [43], [44]. Während virtuelle Maschinen die Kopie eines Betriebssystems ausführen und auf einem Hypervisor aufsetzen [45], stellen Container isolierte Prozesse dar, die direkt auf dem Kernel des Host-Betriebssystems lauffähig sind. Die effizientere Ressourcenauslastung führt somit zu geringeren Kosten [45].

2.3.1 Docker

Es existieren verschiedene Produkte, welche eine Containervirtualisierung ermöglichen und vereinfachen. In Bezug auf Containertechnologie hat sich *Docker*¹ als Industriestandard etabliert [6], [41], [46, S. 215]. Dabei handelt es sich um eine Open-Source-Softwareplattform, die das Erstellen und Verwalten von Containern erheblich vereinfacht.

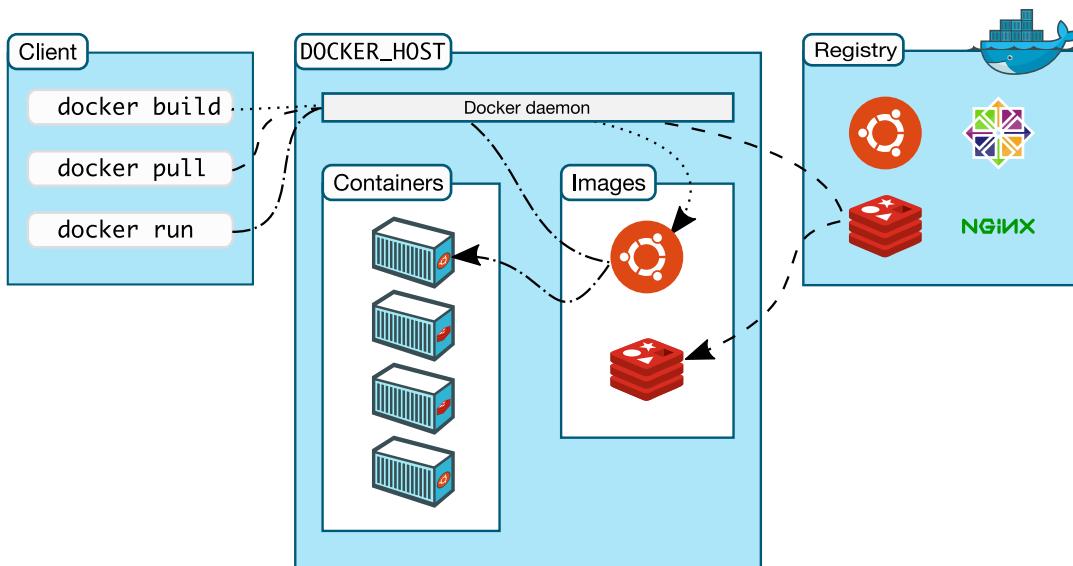


Abbildung 2.3: Docker-Architektur [47]

Die Docker-Technologie besteht dabei aus drei wesentlichen Elementen [47]:

¹<https://www.docker.com/>

Docker Image

Die von Docker erstellten Container nutzen ein isoliertes Dateisystem. Das Dateisystem wird durch das Image bereitgestellt. Dazu muss es alle Komponenten enthalten, die benötigt werden, um die Anwendung auszuführen. Darüber hinaus enthält das Image die Konfiguration des Containers, etwa den Befehl, der beim Start des Containers aufgerufen werden soll und zusätzliche Metadaten [46, S. 216].

Docker Container

Beim Starten eines Images wird ein Container erzeugt. Dies ist ein isolierter Prozess, der die Applikation des Images ausführt. Auf Basis eines Images können mehrere Container erstellt werden. Für die Erstellung von Containern nutzt Docker Features des Linux Kernels: Namespaces und Cgroups [48]. Diese Funktionen sind seit einiger Zeit im Linux Kernel vorhanden [49]. Docker hat die Nutzung allerdings stark vereinfacht und dadurch zur Adaption beigetragen [49].

Docker Registry

Bei einer Docker Registry handelt es sich um ein Repository für Docker-Images. Mit Registries lassen sich bestehende Images anderer Entwickler finden, herunterladen und als Container ausführen. Darüber hinaus können eigene Images erstellt werden, die auf Images von Dritten basieren und entsprechend erweitert werden. Docker bietet mit *Docker Hub*² eine öffentliche Docker Registry, welche mehr als 8 Millionen Images enthält. Neben Docker Hub lassen sich außerdem private Registries nutzen [47].

2.3.2 Kubernetes

In einer Systemlandschaft lassen sich wenige Services mühelos manuell verwalten und als Container bereitstellen. Mit wachsender Komplexität der Systemlandschaft nimmt allerdings auch der administrative Aufwand für die Bereitstellung vieler Services zu.

Um die Bereitstellung von Containern und die Koordination von Releases zu vereinfachen und zu automatisieren, existieren Orchestrationstools wie *Docker Swarm*³, *Apache Mesos*⁴ und *Kubernetes*⁵.

Google arbeitet bereits seit mehr als 15 Jahren mit Linux Containern und hat in diesem Rahmen drei Managementsysteme - Borg, Omega und Kubernetes - für deren Verwaltung entwickelt [50]. Im Jahr 2014 wurde Kubernetes als Open-Source-Projekt veröffentlicht. Heute stellt es eines der größten und bekanntesten Open-Source-Projekte dar [51, S. 1].

²<https://hub.docker.com/>

³<https://docs.docker.com/engine/swarm/>

⁴<https://mesos.apache.org/>

⁵<https://kubernetes.io>

Kubernetes architecture

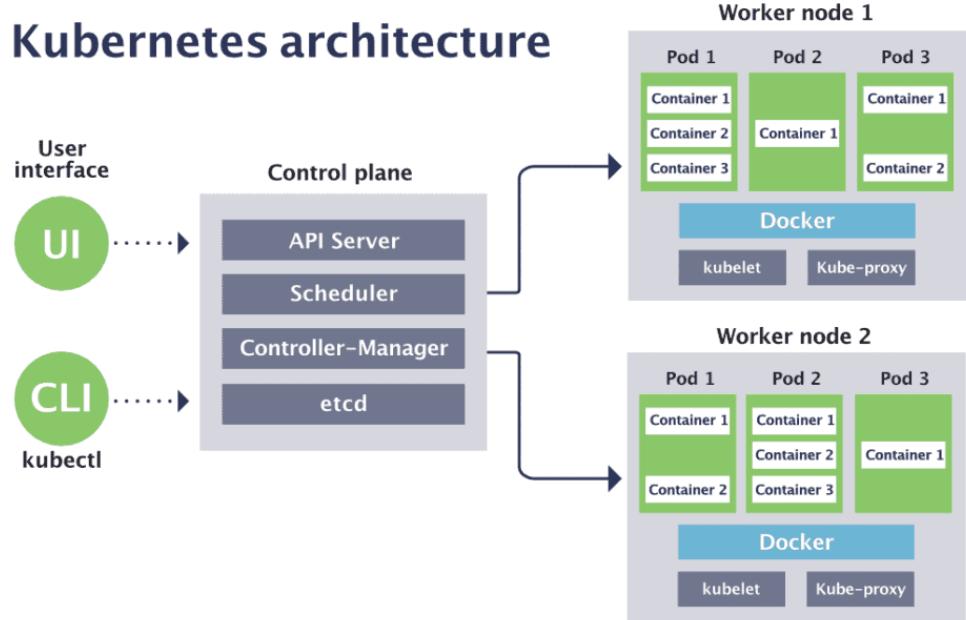


Abbildung 2.4: Kubernetes-Architektur [52]

Die atomare Einheit der Kubernetes-Plattform sind sogenannte Pods. Jeder Pod enthält einen oder mehrere Container. Die Container eines Pods teilen sich dabei die zur Verfügung stehenden Ressourcen, darunter den Speicher und das Netzwerk. Mehrere Pods werden auf einem Knoten (engl. *node*) ausgeführt. Dabei handelt es sich um eine Arbeitsmaschine, die entweder virtuell oder physisch vorhanden ist. Kubernetes verwaltet mehrere Worker-Nodes mit einem Master-Node. Neben der automatischen Bereitstellung von Pods auf verschiedenen Nodes bietet Kubernetes darüber hinaus einige nützliche Funktionen [53], darunter:

- **Service-Discovery und Load-Balancing:** Container werden über einen DNS-Namen oder eine IP-Adresse veröffentlicht und die Last wird automatisch verteilt.
- **Self-Healing:** Nicht funktionsfähige Container werden automatisch neu gestartet.
- **Secret- und Konfigurationsmanagement:** Private Daten wie Passwörtern und Tokens sowie Konfigurationsdaten lassen sich ohne Neuerstellung der Images verwalten.

3 Stand der Forschung

Viele Unternehmen haben ihre monolithischen Anwendungen migriert oder befinden sich in einer laufenden Migration. Die Erkenntnisse und Herausforderungen werden vielfach - etwa in Form von Fallstudien - veröffentlicht [24], [28], [54], [55]. Darüber hinaus existieren einige Arbeiten, die verschiedene Herangehensweisen und Herausforderungen gebündelt zusammenfassen [1], [40], [56], [57], [58], [59].

Fritzs et al. [40] etwa analysieren zehn Studien und klassifizieren die unterschiedlichen Herangehensweisen der Dekomposition in vier unterschiedliche Kategorien. Die identifizierten Techniken zur Dekomposition des Monolithen sind dabei die statische Quellcode-Analyse, die Analyse mithilfe von Metadaten, die Workload-Analyse und die dynamische Komposition. Das Ergebnis ihrer Studie ist ein Leitfaden, der etwa Unternehmen dabei helfen soll, anhand bestimmter Kriterien die richtige Technologie bei der Migration eines Monolithen auszuwählen. Fritzs et al. stellen dabei auch fest, dass es einen Mangel an praktikablen Vorgehensweisen gibt, die eine Tool-Unterstützung bieten.

In ähnlicher Weise untersuchen auch Bajaj et al. [60] die bestehenden Techniken, um eine Microservice-Migration vorzunehmen. Das Ergebnis ihrer Arbeit ist ein Modell, welches die Auswahl einer geeigneten Technologie erleichtern soll.

Sowohl Fritzs et al. [40] als auch Bajaj et al. [60] beschränken sich in ihren Modellen aber lediglich auf die Anwendbarkeit der identifizierten Ansätze. Das reine Vorhandensein der benötigten Eingabedaten ist aber nicht ausreichend, um eine geeignete Dekompositionsstrategie auszuwählen und kann daher nur ein Aspekt sein.

Auch Ponce et al. [1] geben in ihrer Arbeit einen Überblick über bestehende Migrationstechniken und kategorisieren diese. Sie kommen dabei zu einer ähnlichen Kategorisierung wie bereits Fritzs et al. [40]. Die Arbeit gibt einen Überblick über bestehende Technologien, die Auswahl einer geeigneten Vorgehensweise wird aber nicht unterstützt.

Darüber hinaus existiert eine weitere Studie von Fritzs et al. [57], welche die Intentionen, Strategien und Herausforderungen von Microservice-Migrationen in der Industrie untersucht. Dabei identifizieren die Autoren mehrere Prozessstrategien, die in der Industrie angewendet werden. Während viele Unternehmen auf eine komplette Neuentwicklung der Anwendung mit zeitgemäßen Technologien setzen, werden auch weitere Ansätze wie das Strangler-Muster

[61] genutzt, bei dem schrittweise Funktionalitäten aus dem Monolithen in Microservices ausgelagert werden. Es fehlen aber klare Richtlinien, die die Auswahl einer geeigneten Prozessstrategie ermöglichen.

Mit Hinblick auf die anstehende Migration des EJP-CC ist die angedachte Unterstützung zusätzlicher Spielarten ein weiterer Aspekt der Arbeit. In diesem Zusammenhang konnten ebenfalls mehrere Arbeiten identifiziert werden, welche sich mit der Verwaltung von Gemeinsamkeiten und Unterschieden in einer Microservice-Architektur auseinandersetzen.

Während für klassische Softwareanwendungen immer noch das *Don't repeat yourself* (DRY)-Prinzip [62] weit verbreitet ist, argumentieren einige Autoren, dass Code-Duplikationen in Bezug auf Microservice-Architekturen ein valides Mittel sind, um die Unabhängigkeit einzelner Services zu gewährleisten [63], [64], [65].

Andere Ansätze beschreiben das Erstellen mehrerer Services, wobei gemeinsame Funktionalitäten in einen eigenen Service [66, S. 155] oder eine gemeinsam genutzte Bibliothek [66, S. 112], [67] ausgelagert werden. Insbesondere letzterer Ansatz ist allerdings Kritik ausgesetzt [68, S. 13 f.], [69], [70], [71], da eine Kopplung eingeführt wird, welche die Unabhängigkeit der Microservices beeinträchtigt.

De Toledo et al. [70] untersuchen in ihrer Arbeit etwa explizit, zu welchen Problemen die Einführung von gemeinsam genutzten Bibliotheken in der Praxis führt.

Auch Carvalho et al. [23] und Wang et al. [22] untersuchen, wie die Industrie mit der Verwaltung von Gemeinsamkeiten und Unterschieden umgeht [23]. Sie identifizieren unterschiedliche Ansätze, darunter Code-Duplikationen, Feature Flags und gemeinsam genutzte Services.

Einen alternativen Ansatz schlagen Naily et al. [72] mit ihrem Framework vor, welches Methoden der Produktlinienentwicklung verwendet, um Variabilität zu gewährleisten. Das vorgestellte Framework basiert auf der *Abstract Behavioral Specification* (ABS) [73] und beschreibt ein Produkt als eine Zusammensetzung von mehreren Features. Auf diese Weise wird versucht, eine hohe Flexibilität und mehrere Produktvarianten zu unterstützen.

In ähnlicher Weise verfolgen auch Tizzei et al. [74] den Ansatz, Methoden der Produktlinienentwicklung zu nutzen, um die Wiederverwendbarkeit im Kontext von mandantenfähigen SaaS-Systemen zu gewährleisten.

Während somit einige Arbeiten identifiziert werden konnten, die sich mit Variabilitäten innerhalb einer Microservice-Architektur beschäftigen, so fehlen auch hier Richtlinien für die Auswahl einer geeigneten Strategie. Keine der identifizierten Arbeiten liefert ein Modell, welches die Auswahl einer Variabilitätsstrategie für verschiedene Anwendungsfälle ermöglicht. Diese Lücke soll durch die vorliegende Arbeit geschlossen werden.

4 Migration

Eine Softwaremigration ist grundsätzlich durch eine Modernisierung des Systems motiviert [58]. Kazman et al. [75] veröffentlichten 1998 das Hufeisen-Modell (engl. *horseshoe model*), welches den Prozess einer Architekturtransformation abbildet. Daran angelehnt beschreiben Di Francesco et al. [58] die Transformation im Kontext von Microservices (vgl. Abbildung 4.1).

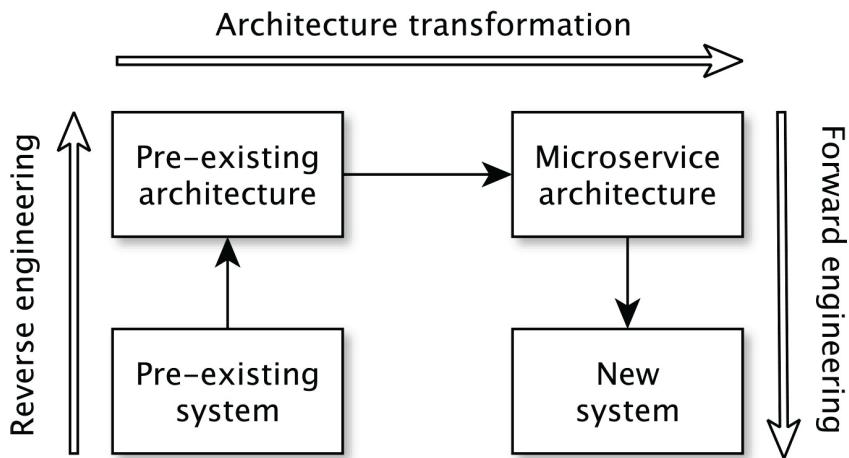


Abbildung 4.1: Abgewandeltes Hufeisenmodell [58]

Zunächst wird in diesem Modell die bestehende Anwendung analysiert (*Reverse engineering*), um Wissen und Erkenntnisse über das System zu gewinnen. Auf dieser Basis werden Microservice-Kandidaten ermittelt und die neue Architektur entworfen (*Architecture transformation*), bevor das neue System implementiert wird (*Forward engineering*).

Im Folgenden werden existierende Vorgehensweisen betrachtet, mit welchen sich die modellierte Migration vollziehen lässt. Dabei wird eine Unterscheidung zwischen Dekompositionstrategien und Prozessstrategien vorgenommen.

Erstere beschäftigen sich mit der Analyse des bestehenden Monolithen und der Identifizierung sowie Extraktion von geeigneten Microservices. Dies entspricht den Schritten *Reverse engineering* und *Architecture transformation* des abgewandelten Hufeisenmodells. Nachdem geeignete Microservice-Kandidaten ermittelt wurden, lassen sich verschiedene Prozessstrategien nutzen, um die Migration zu vollziehen. Die Prozessstrategien werden im *Forward engineering* Schritt des Modells angewendet.

Insbesondere die Dekomposition und das Finden von geeigneten Microservices wird in der Praxis als große Herausforderung gesehen [1], [14], [15]. Dabei wird die Dekomposition oft unstrukturiert vorgenommen [14], [20]. Dies ist besonders deshalb kritisch zu sehen, da eine schlechte Aufteilung der Microservices erhebliche Nachteile mit sich bringt und somit die Qualität des Systems beeinflusst. Je nach Feinheit des Systems werden etwa Performance und Testbarkeit beeinflusst [76].

In der Praxis werden Services oft falsch eingeteilt [71], was etwa dazu führt, dass Änderungen an der Software mehrere Services betreffen und Overhead in Form von erhöhter Inter-Service Kommunikation entsteht [57]. Dies wird durch die Tatsache begründet, dass Vorgehensweisen und Tools, welche die Dekomposition unterstützen, in der Industrie nicht bekannt sind oder davon ausgegangen wird, dass Tools eine komplexe Dekomposition nicht unterstützen können [57].

Die bestehenden Möglichkeiten und Technologien wurden im Rahmen einer Literaturrecherche ermittelt und werden nachfolgend dargestellt.

4.1 Dekompositionsstrategien

In ihrer Studie klassifizieren Fritzsch et al. [40] zehn Refactoring-Ansätze in vier verschiedene Kategorien. Die statische Code-Analyse, die durch Metadaten unterstützte Analyse, die Workload-Analyse und die dynamische Komposition. Zu einer ähnlichen Klassifikation kommen auch Ponce et al. [1] in ihrer Arbeit. Sie identifizieren die drei Kategorien der statischen Analyse, der dynamischen Analyse (entspricht der Workload-Analyse) und der modellgetriebenen Analyse (entspricht der durch Metadaten unterstützten Analyse).

Während in der Literaturrecherche auch einige neue Ansätze identifiziert werden konnten, die nicht in den Arbeiten von Fritzsch et al. und Ponce et al. enthalten waren, so lassen sich dennoch alle Ansätze in die bestehenden Kategorien einordnen.

4.1.1 Statische Analyse

Die statische Analyse versucht Microservice-Kandidaten auf Basis des Quelltextes zu identifizieren. Dazu können etwa Abhängigkeiten in Form von Methodenaufrufen ermittelt werden. Ziel ist es, Gruppen von atomaren Einheiten zu finden, die möglichst unabhängig voneinander sind und somit einen Microservice-Kandidaten darstellen können. Je nach Ansatz unterscheidet sich die Größe und Art der atomaren Einheiten. Einige Ansätze nehmen eine Gruppierung auf Basis von Klassen vor, während andere Ansätze feingranularer vorgehen und eine Gruppierung auf Basis von Funktionen vornehmen [40].

Ein konkreter Ansatz der statischen Analyse wird durch Pigazzini et al. [16] vorgestellt. Diese kommen zu der Erkenntnis, dass die Dekomposition ein Prozess ist, der größtenteils manuell vorgenommen wird und versuchen den manuellen Aufwand zu verringern, indem Sie einen neuen Ansatz in Form eines Tools bereitstellen.

Ihr Ansatz besteht aus drei Schritten:

1. Erkennung von schlechtem Architekturdesign (engl. *architectural smell detection*)
2. Analyse des Abhängigkeitsgraphen (engl. *dependency graph analysis*)
3. Themenerkennung (engl. *topic detection*)

Dafür erweitern Sie das Tool *Arcan*¹. Bei dieser Software handelt es sich um ein Analyseprogramm für Java-Anwendungen, welches schlechtes Architekturdesign automatisch erkennen soll. Diese Funktionalität wird auch im ersten Verarbeitungsschritt genutzt, um etwa zyklische Abhängigkeiten zu identifizieren. Die Erkennung und anschließende Behebung dieser Abhängigkeiten soll die Codequalität erhöhen und somit bessere Microservice-Kandidaten liefern.

Im zweiten Schritt werden die Abhängigen des Quelltextes analysiert. Dazu wird ein Graph erstellt, der die Zusammenhänge verschiedener Klassen repräsentiert. Die Knoten des Graphen stellen die Klassen dar, die Kanten stellen die Abhängigkeiten zwischen diesen dar.

Auf Basis des Graphen lassen sich drei verschiedene Auswertungen durchführen. Die erste Auswertung (*Connected Components Detection*) basiert auf einer Tiefensuche und ermittelt zusammenhängende Komponenten innerhalb des erstellten Graphen, die als potenziell eigenständiger Microservice ausgelagert werden können.

Eine weitere Ansicht erlaubt es, funktional zusammenhängende Klassen zu identifizieren. Die Annahme der Autoren ist, dass der Einstiegspunkt einer Funktionalität des Monolithen etwa ein REST-Controller ist, der eine Operation öffentlich zur Verfügung stellt. Durch Angabe dieser Klassen als Einstiegspunkte einer Tiefensuche werden vertikale Funktionalitäten ausgewertet. Pigazzini et al. bezeichnen diese Ansicht auch als *Vertical Functionality View*.

Die letzte Ansicht wird *Logical Layer View* genannt und versucht die vorhandenen Klassen in die drei Schichten Präsentation, Geschäftslogik und Persistenz einzuteilen. Die Zuordnung einer Klasse zu einer der drei Schichten basiert auf der Auswertung der Abhängigkeiten der JEE-Spezifikation. Dies soll Architekten, die bislang keinen Überblick über die Anwendung haben, dabei helfen, die verschiedenen Klassen einzuordnen und somit einen grobgranularen Überblick über die Anwendung geben.

¹<https://essere.disco.unimib.it/wiki/arcان/>

Die Analyse des Graphen kann lediglich Abhängigkeiten zwischen den Klassen ermitteln. Microservices hingegen werden typischerweise anhand von Domänen und Geschäftsfähigkeiten modelliert. Ob ein identifizierter Cluster des Graphen auch fachlich in die gleiche Domäne einzuordnen ist, kann das Verfahren nicht feststellen.

Um die Gruppierung auch anhand einer fachlichen Domäne vorzunehmen, ergänzen die Autoren das bestehende Vorgehen daher um einen weiteren Verarbeitungsschritt: die Themenerkennung. Dabei werden die vorhandenen Klassen analysiert und versucht Themengebiete zu erkennen. Ein Themengebiet kann eine fachliche Domäne repräsentieren und somit durch einen Microservice abgebildet werden. Im Rahmen der Themenerkennung wird für jede Klasse die semantische Ähnlichkeit mit den extrahierten Themen berechnet, und anschließend eine Gruppierung vorgenommen. Für die Themenerkennung nutzen Pigazzini et al. den Latent Dirichlet Allocation Algorithmus [77] der Python-Bibliothek *guidedLDA*².

Neben Pigazzini et al. propagieren einige weitere Arbeiten die statische Analyse, um Microservices zu identifizieren [27], [78], [79], [80], [81].

Brito et al. [80] verbinden ebenfalls strukturelle und lexikografische Informationen, um eine Themenerkennung mit einer anschließenden Clusteranalyse vorzunehmen.

Levcovitz et al. [79] beschreiben einen strukturierten manuellen Prozess, der Microservice-Kandidaten ermittelt, indem Abhängigkeiten zwischen Fassaden, Geschäftsfähigkeiten und Datenbanktabellen analysiert werden.

Saidani et al. [78] beschreiben die Extraktion von Microservices als ein Optimierungsproblem zweier Eigenschaften: die Minimierung der Kopplung und die Maximierung der Kohäsion. Ihr Ansatz analysiert die Abhängigkeiten zwischen verschiedenen Klassen und nutzt einen evolutionären Algorithmus [82] zur Optimierung der Fitnessfunktion.

4.1.2 Dynamische Analyse

Während die statische Analyse eines der meist genutzten Verfahren ist, hat sie dennoch einige Nachteile. Insbesondere die Nutzung von Techniken wie Reflection, das dynamische Laden von Klassen und Dependency Injection haben zur Folge, dass Abhängigkeiten nicht klar extrahiert werden können [83]. Auch das dynamische Verhalten der Anwendung unter Berücksichtigung von Nutzereingaben und Konfigurationsdateien kann für Unsicherheiten sorgen [31].

Um diesen Limitationen zu begegnen, nutzen einige Arbeiten dynamische Laufzeitinformationen, um Microservice-Kandidaten zu ermitteln. Einen dieser Ansätze präsentieren Taibi und Systä [84]. Dieser basiert auf der Auswertung von Trace-Logs. Zentrale Voraussetzung ist damit das Vorhandensein ausführlicher Log-Dateien. Diese müssen für alle Benutzer- und

²<https://guidedlda.readthedocs.io/en/latest/>

API-Operationen die involvierten Klassen und Methoden enthalten. Neben der Möglichkeit Code-Anpassungen für das Logging vorzunehmen, schlagen die Autoren das Tool *Elastic APM*³ vor, um die benötigten Informationen zu sammeln.

Sind die Grundvoraussetzungen erfüllt, besteht die Ermittlung der Microservice-Kandidaten aus mehreren manuellen Schritten. Zunächst werden die Ausführungspfade mithilfe des Process-Mining-Tools *DISCO*⁴ ermittelt. Dieses erlaubt es, die Geschäftsprozesse aus den Log-Dateien zu extrahieren und grafisch darzustellen. Im nächsten Schritt wird eine Frequenzanalyse durchgeführt, mit der sich sowohl häufig als auch selten genutzte Aufrufpfade identifizieren lassen.

Analog zum statischen Ansatz von Pigazzini et al. [16] analysiert auch der dynamische Ansatz die Aufrufpfade auf mögliche zirkuläre Abhängigkeiten, die im Microservice-Umfeld ein Anti-Pattern darstellen [85] und aufgelöst werden sollen.

Auf Basis der bereinigten Pfade werden Ablaufgraphen generiert und manuell ausgewertet. Grundsätzlich stellen Taibi und Systä fest, dass es keine objektiv beste Aufteilung gibt und Softwarearchitekten mehrere Dekompositionsoptionen miteinander vergleichen und abwegen müssen.

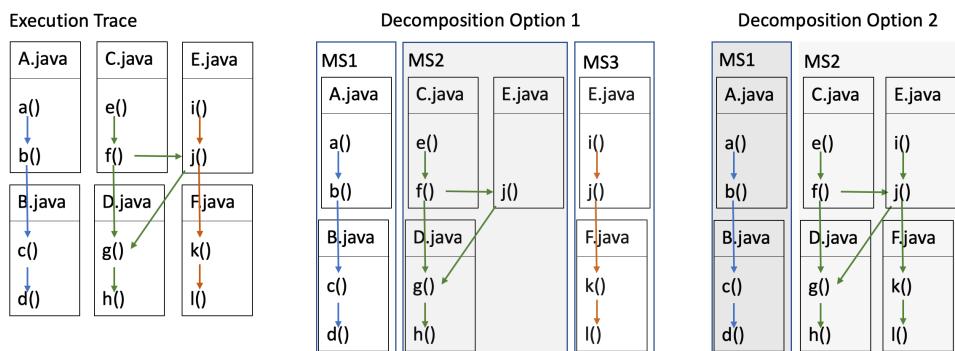


Abbildung 4.2: Auf Basis von Trace-Logs ermittelte Dekompositionsoptionen [84]

Abbildung 4.2 zeigt ein vereinfachtes Modell von drei identifizierten Aufrufpfaden zwischen verschiedenen Klassen. Die erste Option ist eine Aufteilung in drei Microservices, wobei eine von mehreren Pfaden genutzte Komponente in diesem Fall dupliziert werden muss. Soll eine Code-Duplikation vermieden werden, so ist die zweite Option eine gröbere Aufteilung in lediglich zwei Microservices.

Um die Entscheidung zu unterstützen, schlagen Taibi und Systä drei Metriken vor, auf deren Basis sich die Qualität der Aufteilung bewerten lässt: die Kopplung, die Anzahl der Klassen pro Microservice und die Anzahl der zu duplizierenden Klassen.

³<https://www.elastic.co/de/apm/>

⁴<https://fluxicon.com/disco/>

Während es sich bei dem Ansatz um ein größtenteils manuelles Vorgehen handelt, existieren ähnliche Ansätze, die Microservice-Kandidaten halbautomatisch ermitteln. Jin et al. [86] sammeln in ihrer Arbeit repräsentative Trace-Logs durch das Ausführen von funktionalen Tests. Sie ermitteln auf Basis der Logs zunächst funktionale, atomare Einheiten und nutzen anschließend einen evolutionären Algorithmus [82], um diese zu gruppieren und Microservice-Kandidaten zu generieren.

Ein Tool für die Dekomposition von Monolithen wird mit *Mono2Micro*⁵ angeboten [83], [87]. Dieses wird von IBM als Produkt vertrieben und ermittelt Microservice-Kandidaten ebenfalls auf Basis von Trace-Logs. Darüber hinaus bietet es die Möglichkeit, prototypische Implementierungen der identifizierten Microservices auf Basis des WebSphere-Liberty- oder des Open-Liberty-Applikationsservers zu generieren.

4.1.3 Kombination aus statischer und dynamischer Analyse

Während die dynamischen Ansätze versuchen einige Nachteile der statischen Analyse zu lösen, so hat auch die dynamische Analyse selbst eine Limitation. Die Qualität der ermittelten Microservice-Kandidaten ist abhängig von den gesammelten Laufzeitdaten [88]. Es werden entsprechend nur Abhängigkeiten zwischen Klassen erkannt, wenn diese während des Nutzungszeitraumes, in dem die Laufzeitdaten gesammelt wurden, aktiv waren [88]. Nicht oder wenig genutzte Funktionen sind andernfalls im Modell nicht vorhanden oder unterrepräsentiert [31]. Entsprechend wichtig ist es, die Anwendung im Überwachungszeitraum möglichst repräsentativ zu nutzen.

Um die Vorteile der statischen und dynamischen Ansätze zu vereinen, nutzen einige Arbeiten eine Kombination der beiden Techniken. Ren et al. [31] nutzen die statische Analyse, um einen abstrakten Syntaxbaum zu erzeugen. Die dynamischen Laufzeitinformationen werden ergänzt, um eine Gewichtung der Kanten vorzunehmen und so den Grad der Kopplung verschiedener Einheiten darzustellen. Für die Ermittlung von Microservice-Kandidaten wird der k-Means-Algorithmus [89] angewendet. Dabei handelt es sich um einen weit verbreiteten Cluster-Algorithmus, welcher Daten in genau k Gruppen unterteilt. Jede Gruppe stellt in diesem Kontext einen Microservice-Kandidaten dar.

Abbildung 4.3 zeigt einen erzeugten Graphen, der durch Variation des Parameters k in zwei (b), drei (c) sowie vier (d) Partitionen aufgeteilt wurde. Eine höhere Anzahl an Clustern entspricht somit einer feingranulareren Aufteilung der Microservice-Kandidaten. Durch Variation des Parameters k können mit geringem Aufwand verschiedene Dekompositionsmöglichkeiten evaluiert werden.

⁵<https://www.ibm.com/cloud/mono2micro>

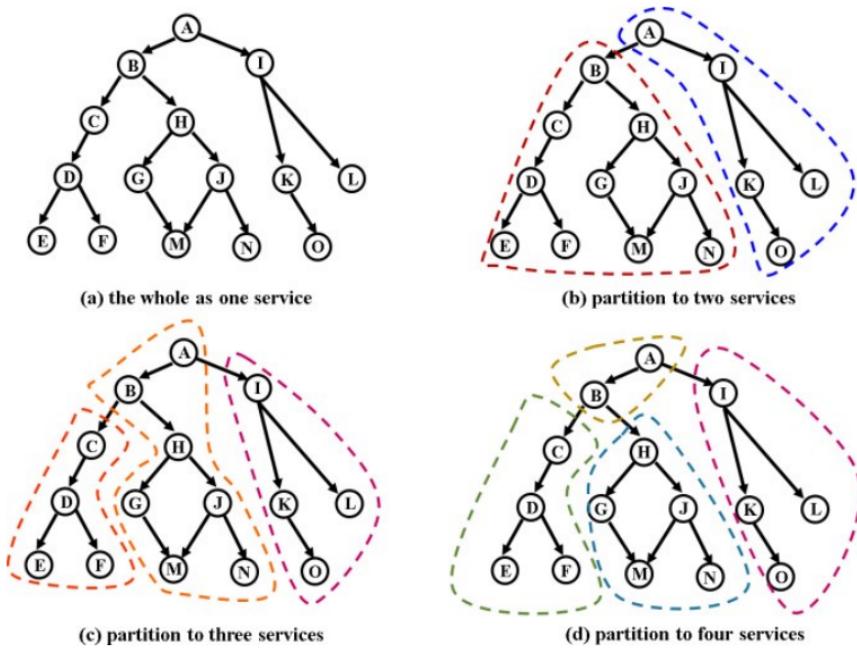


Abbildung 4.3: Partitionierung auf Basis des k-Means-Algorithmus [31]

Matias et al. [88] kombinieren die Informationen aus statischer und dynamischer Analyse in ähnlicher Weise. Sie stellen ihren Ansatz *Monobreaker*⁶ als Prototyp bereit. Dieser beschränkt sich auf Python-Anwendungen, welche das *Django-Framework*⁷ nutzen.

Ein manueller Prozess wird von Krause et al. [54] vorgestellt. Neben *Structure101*⁸ für die statische Analyse nutzen diese das Tool *ExplorViz*⁹, um dynamische Informationen grafisch darzustellen und so das Wissen über die monolithische Anwendung zu maximieren.

4.1.4 Modellgetriebene Analyse

Modellgetriebene Ansätze arbeiten mit abstrakteren Eingabedaten wie UML-Diagrammen, Use-Cases, Schnittstellendokumentationen und historischen Daten aus Versionskontrollsystmen [1], [40].

Ein bekannter Ansatz, welcher auf Basis von Modellen und 16 verschiedenen Kopplungskriterien arbeitet, ist *Service Cutter*¹⁰. Gysel et al. [90] veröffentlichten das Tool im Rahmen ihrer Arbeit bereits im Jahr 2016. Es ist damit eines der ersten Tools, welches die Komposition von Microservices unterstützt und wird allgemein als State-of-the-Art beschrieben [59], [91]. Da

⁶<https://github.com/tiagoCMatias/monoBreaker>

⁷<https://www.djangoproject.com/>

⁸<https://structure101.com/>

⁹<https://www.explorviz.net/>

¹⁰<https://servicecutter.github.io/>

es mit abstrakten Eingabedaten arbeitet und nicht etwa auf spezifische Programmiersprachen oder Frameworks beschränkt ist, lässt es sich immer anwenden [40].

Service Cutter arbeitet auf Basis einer Systemspezifikation, welche als *Entity-Relationship-Modell* (ERM) bereitgestellt wird. Die Spezifikation kann anschließend um weitere Informationen wie Use-Cases und Aggregates [92] ergänzt werden. Aus der Systemspezifikation extrahiert Service Cutter sogenannte Nanoentities. Diese umfassen Daten, Operationen und Artefakte. Der Benutzer hat anschließend die Möglichkeit, verschiedene Kopplungskriterien zu bewerten und zu priorisieren (vgl. Abbildung A.1). Auf Basis dieser Bewertung erstellt Service Cutter einen gewichteten Graphen. Dieser lässt sich wahlweise mit vier verschiedenen Cluster-Algorithmen [93], [94], [95], [96] in mehrere Partitionen unterteilen, welche sich visuell darstellen lassen.

Ein weiterer Ansatz basiert auf der Analyse der Anwendungsschnittstelle. Baresi et al. [91] analysieren in ihrer Arbeit die OpenAPI-Spezifikation einer Anwendung, um Microservice-Kandidaten zu ermitteln. Sie nutzen das DISCO-Modell [97], um die semantische Ähnlichkeit mehrerer Operationen zu berechnen und diese zu gruppieren. Bei dem Modell handelt es sich um eine vorberechnete Datenbank, welche die Ähnlichkeit verschiedener Wörter enthält. Diese wurde durch Analyse der Kookkurrenz in großen Textsammlungen wie Wikipedia erstellt [91].

Ebenfalls existieren Ansätze, die Information eines Versionsverwaltungssystems nutzen, um Microservice-Kandidaten zu identifizieren. Mazlami et al. [18] extrahieren aus dem Versionskontrollsystem Klassen, Änderungshistorie und die an dem Code beteiligten Entwickler. Die Autoren beschreiben drei verschiedene Extraktionsstrategien, anhand derer ein gerichteter Graph erstellt und anschließend partitioniert wird. Die logische Kopplung untersucht, welche Klassen sich gemeinsam geändert haben. Dabei wird davon ausgegangen, dass gemeinsam geänderte Klassen zur gleichen Funktionalität gehören. Die semantische Kopplung untersucht die semantische Ähnlichkeit mehrerer Klassen mithilfe des Tf-idf-Maßes [98]. Die letzte Strategie versucht Team-Strukturen offenzulegen, indem untersucht wird, ob mehrere Gruppen von Personen jeweils unterschiedliche Klassen bearbeiten.

4.1.5 Domain-driven Design

Microservices sollen anhand von Geschäftsfähigkeiten modelliert werden [3], [33, S. 2]. Änderungen an einer Fachlichkeit sollen nur einen Microservice betreffen und somit leicht umsetzbar sein [33, S. 2]. Auch lassen sich mehrere Features in verschiedenen Services parallel umsetzen, da unterschiedliche Fachlichkeiten betroffen sind [66, S. 65].

Ein Konzept zur Einteilung einer fachlichen Domäne bietet Evans mit seiner Arbeit *Domain-Driven Design: Tackling Complexity in the Heart of Software* [39]. Evans zufolge ist ein gutes Verständnis der Fachlichkeit essenziell, um die Komplexität von Software beherrschbar zu machen.

Dazu beschreibt er in seiner Arbeit einige Bausteine, mit denen sich eine Domäne modellieren lässt. Einige dieser Bausteine sind unter Entwicklern weit verbreitet und werden teilweise auch von Frameworks wie *Spring*¹¹ unterstützt [99, S. 152].

Zu den zentralen Bestandteilen des Domänenmodells gehören:

- Entitäten (*Entities*) - Objekte, die eine eigene Identität besitzen. Verschiedene Entitäten können die gleichen Attribute aufweisen [99, S. 151].
- Wertobjekte (*Value Objects*) - Ein Objekt, dass eine Sammlung von Attributen darstellt. Es besitzt keine eigene Identität [100].
- Serviceobjekte (*Services*) - Objekte, die Operationen implementieren, welche weder Entitäten noch Wertobjekten zugeordnet werden können [39, S. 104].
- Aggregate (*Aggregates*) - Domänenobjekte, welche mehrere Entitäten und Wertobjekte gruppieren. Ein Aggregat wird als Einheit verwendet und bildet die Transaktionsgrenze des Modells [39, S. 125–129].

Das *Domain-driven Design* (DDD) unterteilt eine komplexe Domäne in mehrere Bounded Contexts, welche fachlich zusammengehörige Einheiten darstellen. Jeder Bounded Context kann dann als ein eigenständiger Microservice implementiert werden.

In der Praxis ist das DDD ein weit verbreitetes Konzept zur Modellierung von Microservices [6], [14], [101].

Einige Arbeiten bauen auf dem DDD auf und versuchen die Identifikation von Bounded Contexts zu unterstützen. Tyszberowicz et al. [101] beschreiben einen systematischen Prozess, der dabei helfen soll, ein System in mehrere voneinander getrennte Subsysteme zu unterteilen und somit geeignete Microservice-Kandidaten zu identifizieren. Dafür analysieren sie vorliegende Use-Case-Spezifikationen und extrahieren Operationen sowie Zustandsvariablen des Systems. Darauf basierend erstellen sie eine Zuordnungstabelle, die festhält, welche Zustandsvariablen von welchen Operationen gelesen und geschrieben werden. Die Tabelle stellen sie mithilfe des Programms *NEATO*¹² [102] grafisch als gewichteten Graphen dar, um funktional zusammengehörige Cluster zu identifizieren. Mithilfe der zuvor erstellen Tabelle, lassen sich außerdem die benötigten Daten der einzelnen Microservices ablesen. Tyszberowicz et al. beschreiben, dass ihr funktionaler Dekompositionsansatz zu Bounded Contexts führt, ähnlich wie es das DDD

¹¹<https://spring.io/>

¹²<http://www.graphviz.org/docs/layouts/neato/>

empfiehlt. Das systematische Vorgehen beschleunigt laut ihnen allerdings die Identifikation der Microservice-Kandidaten.

4.2 Prozessstrategien

Nachdem geeignete Microservice-Kandidaten ermittelt wurden, muss die neue Systemlandschaft implementiert werden. Dies entspricht dem *Forward engineering* Schritt des Hufeisenmodells (vgl. Abbildung 4.1). Im Folgenden werden die verschiedenen Möglichkeiten betrachtet, diesen Prozess zu vollziehen.

4.2.1 Neuentwicklung

Eine intuitive und oft anwendbare Möglichkeit ist die vollständige Neuentwicklung der Anwendung. Eine solche Vorgehensweise wird auch als Greenfield-Entwicklung bezeichnet. Die Entwicklung startet auf einer „grünen Wiese“, ohne dass Abhängigkeiten zu dem potenziell vorhandenen Altsystem bestehen. Während der Entwicklung treten somit keine unerwünschten Seiteneffekte im bestehenden System auf.

Entwickler sehen Neuentwicklungen oft als Chance, die potenziell schlechte existierende Struktur des Monolithen zu vergessen und das neue System mit modernen Technologien und einer verbesserten Struktur zu entwickeln.

Fritzsch et al. [57] identifizieren in ihrer Studie verschiedene Vorgehensweise und Motivationen für eine Migration hin zu Microservices. Dafür führen Sie mehrere Interviews mit IT-Fachleuten aus der Praxis. Einige Teilnehmer berichten, ihre Microservice-Architektur im Rahmen einer Neuentwicklung etabliert zu haben. Als Motivation dafür gibt ein Teilnehmer etwa an: „The best consultant on earth can't grasp what they have built over 10 years.“ Die Komplexität bestehender Anwendungen wird als zu hoch erachtet, was Erweiterungen oder Änderungen am bestehenden Monolithen verhindert.

Ebenso gibt ein Teilnehmer in der Studie von Di Francesco et al. [58] an: „Of the old system we consider it as so bad that we do not look at the source code“. Die Qualität des bestehenden Quelltextes wird in diesem Fall als Last betrachtet, die bewusst nicht berücksichtigt wird. Eine Neuentwicklung kann in solchen Fällen eine gute Möglichkeit sein, die Microservice-Architektur von Grund auf neu zu entwickeln und die Softwarequalität zu verbessern.

Abbildung 4.4 zeigt die grundsätzliche Vorgehensweise einer Neuentwicklung. Die neue Microservice-Architektur wird parallel zu dem bestehenden Monolithen entwickelt. Sobald die Microservice-Architektur vollständig implementiert und getestet wurde, wird der produktive Betrieb von dem alten monolithischen System auf die neue Architektur verlagert.

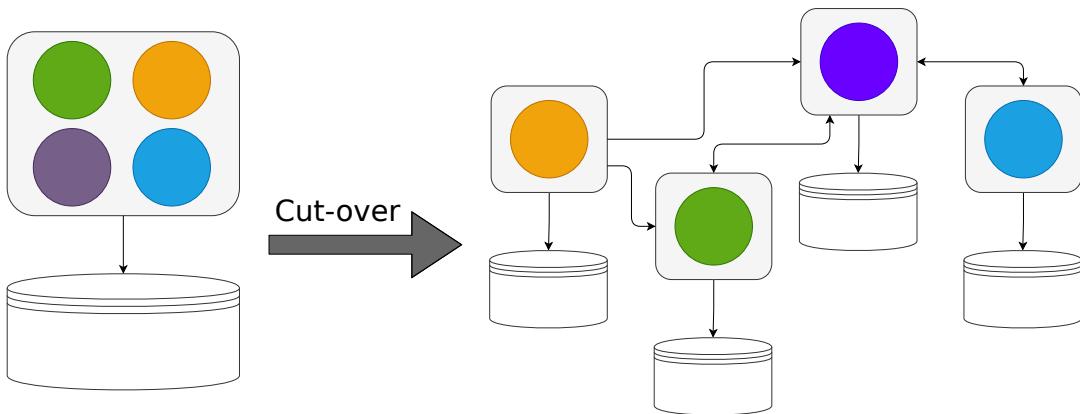


Abbildung 4.4: Migrationsprozess im Rahmen einer Neuentwicklung

Dieser Vorgang wird auch als Cut-over bezeichnet. Nachteil dieses Vorgehens ist, dass dies einen kritischen Vorgang darstellt, der mit einigen Risiken behaftet ist [103], [104, S. 17]. Nach der Umstellung ist es möglich, dass vermehrt Fehler auftreten, da die gesamte neue Systemlandschaft in einem einzigen Schritt und nicht iterativ eingeführt wird. Potenziell neu auftretende Fehler lassen sich somit nicht auf einen Teilbereich der Software eingrenzen.

4.2.2 Erweiterungen

Die Erweiterungsstrategie konzentriert sich auf neue Features. Ziel ist es, lediglich neue Funktionalitäten als Microservices zu implementieren. Der existierende Monolith bleibt dabei bestehen und wird um einige Microservices ergänzt (vgl. Abbildung 4.5), welche sowohl mit dem Legacy-System als auch untereinander kommunizieren können [99, S. 434].

Insbesondere für sehr alte und komplexe monolithische Systeme kann dies ein möglicher Anwendungsfall sein. Im Bankenumfeld etwa existieren oft noch große Mainframe-Programme [105], die in Programmiersprachen wie Cobol entwickelt wurden [26]. Die Komplexität dieser Programme wird oft als so hoch beschrieben, dass ein Überblick über die Gesamtanwendung nicht mehr möglich ist [26]. Änderungen und Erweiterungen am monolithischen Code bergen somit nicht abschätzbare Risiken.

Gouigoux und Tamzalit [28] beschreiben in ihrer Arbeit die Migrationen einer monolithischen Anwendung hin zu einer weborientierten Microservice-Architektur. Einige Kunden sind allerdings weiterhin auf die monolithische Legacy-Anwendung angewiesen. Um auch diese Kunden mit neuen Features zu versorgen, ohne den Monolithen weiterzuentwickeln, wurden mehrere Microservices implementiert, die das Altsystem um neue Funktionalitäten erweitern.

Insgesamt ist die Erweiterungsstrategie ein populärer Ansatz. In der Studie von Di Francesco et al. [58] geben 10 von 18 Teilnehmern an, zunächst lediglich neue Funktionalitäten als

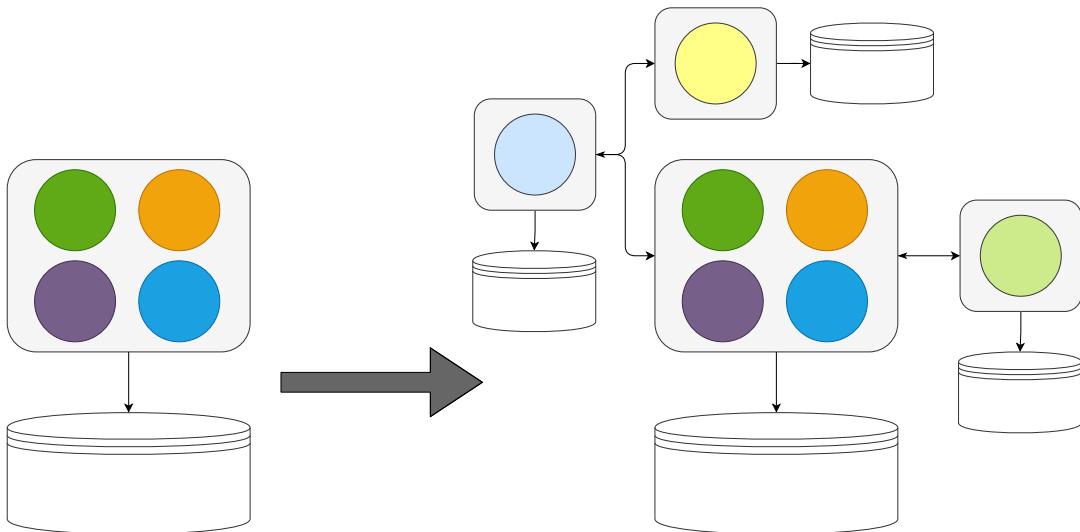


Abbildung 4.5: Monolith mit ergänzenden Microservices

Microservices zu implementieren.

4.2.3 Strangler-Muster

Eine Migration kann ein komplexer Vorgang sein, der mitunter viel Zeit in Anspruch nimmt [106]. Daher kann es sinnvoll sein, die Migration in mehreren einzelnen Schritten zu vollziehen. Ein iteratives Vorgehen ist in der Industrie weit verbreitet. In der Studie von Di Francesco et al. [58] nutzten 14 von 18 Teilnehmern diese Möglichkeit.

Einen Ansatz für eine schrittweise Migration liefert Martin Fowler mit dem Strangler-Muster (auch Strangler-Fig-Muster) [61].

Der Name dieser Vorgehensweise basiert auf Würgefeigen (engl. *strangler figs*), welche Fowler im Regenwald beobachtet hat. Diese beginnen ihren Lebenszyklus in der Krone eines anderen Baumes und wachsen Richtung Erde, um dort zu wurzeln. Dabei würgen sie den Wurzelbaum und lassen diesen tot zurück. In Anlehnung an diese Metapher beschreibt Fowler, wie ein monolithisches System langsam in eine Microservice-Architektur überführt werden kann.

Dazu wird jeweils ein Microservice-Kandidat ermittelt und aus dem Monolithen extrahiert. Eine Strangler-Fassade leitet eingehende Anfragen anschließend entweder an das Legacy-System oder an den neuen Microservice weiter. Erweist sich der extrahierte Microservice als stabil und funktionsfähig, kann die entsprechende Funktionalität aus dem Monolithen entfernt werden. Dieses Vorgehen wird anschließend für weitere Microservices wiederholt. Abbildung 4.6 zeigt die Entwicklung der Architektur im zeitlichen Verlauf einer Migration.

Während der Großteil der Funktionalität zu Beginn der Migration im Legacy-System liegt und nur ein kleiner Teil in Microservices implementiert ist, schrumpft das bestehende System

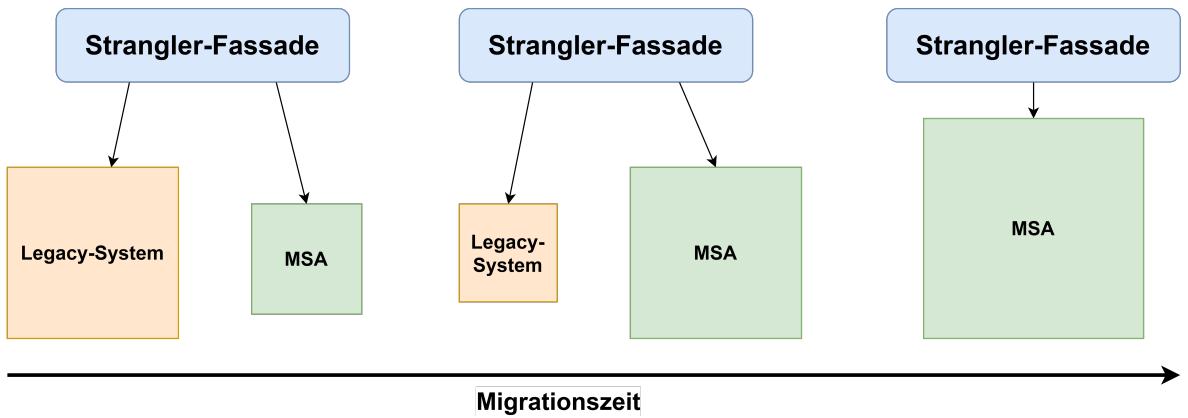


Abbildung 4.6: Anwendung des Strangler-Musters zur Überführung eines Legacy-Systems in eine Microservice-Architektur (MSA)

im zeitlichen Verlauf immer weiter, bis es vollständig durch eine Microservice-Architektur abgelöst wurde. Im Gegensatz zu einer parallelen Neuentwicklung wird auf diese Weise ein risikoreicher Cut-over-Prozess vermieden. Treten neue Fehler auf, lassen sich diese auf ein Teilsystem eingrenzen und können schrittweise bearbeitet werden [107].

Eine konkrete Umsetzung des Strangler-Musters wird von Freire et al. [108] vorgeschlagen. Diese nutzen die aspektorientierte Programmierung [109], um die Codeausführung einer Klasse dynamisch zu beeinflussen. Diese hat das Ziel wiederkehrende Belange wie Protokollierung oder Transaktionsmanagement - sogenannte Aspekte - von der Geschäftslogik der Anwendung zu trennen. An sogenannten Join Points - etwa bei Methodeneintritt oder Methodenaustritt - kann variabel weiterer Code eingeführt werden. Dies erweist sich vor allem für Querschnittsthemen (auch *Cross-Cutting Concerns*) wie der Protokollierung als hilfreich. Der an einem Join Point eingefügte Code wird Advice genannt.

Freire et al. [108] nutzen Advices, um Funktionsaufrufe des Monolithen abzufangen und diese mit Serviceaufrufen an den neuen Microservice zu ersetzen. Diese Vorgehensweise erfordert lediglich minimale Anpassungen am Quellcode des Legacy-Systems [108]. Einen weiteren Vorteil der Vorgehensweise sehen die Autoren in der Rückwärtsmigration. Sollte sich der extrahierte Microservice als fehlerhaft erweisen, kann durch Anpassen einer Konfigurationsdatei der Aspekt so geändert werden, dass wieder die ursprüngliche Implementation des Monolithen verwendet wird [108].

5 Variabilität

Dieses Kapitel beschäftigt sich mit verschiedenen Möglichkeiten, Variabilität in einer Microservice-Architektur zu erreichen. Grundsätzlich ist das Ziel, eine Software in mehreren Kontexten nutzbar zu machen. Als Beispiel dafür diene die anstehende Migration des EJP-CC, welches für die Verarbeitung des Spiels Eurojackpot verwendet wird. Die grundsätzliche Vorgehensweise der Verarbeitung lässt sich auch auf andere Spiele wie Lotto 6aus49 übertragen, wobei lediglich im Detail Unterschiede bestehen. Gemeinsamkeiten und Unterschiede sollen so implementiert werden, dass sich die Software nach Möglichkeit in mehreren Spielekontexten nutzen lässt.

In der Studie von Wang et al. [22] beschreiben 31 der 37 Teilnehmer, dass ihr Produkt mehrere Varianten unterstützen muss. Auch die Hälfte aller Teilnehmer in der Studie von Carvalho et al. [23] gibt an, Variabilität bei der Implementierung einer Microservice-Architektur berücksichtigt zu haben.

Insbesondere im Kontext von SaaS-Produkten, welche von verschiedenen Kunden genutzt werden, sind Variabilität und Anpassungsfähigkeit wichtige Themen. Alle Kunden greifen auf die gleiche Software zu, jeder Kunde hat aber gegebenenfalls andere Anforderungen an Benutzeroberfläche, Konfiguration und Workflows.

Im Folgenden werden verschiedene Vorgehensweisen betrachtet, wie Gemeinsamkeiten und Variabilitäten des Quellcodes in einer Microservice-Architektur verwaltet werden können.

5.1 Feature Flags

Eine Möglichkeit zur Einführung von Variabilität ist die Nutzung von sogenannten Feature Flags. Diese steuern den Programmfluss und führen bestimmte Codeabschnitte nur aus, wenn eine entsprechende Funktionalität freigeschaltet ist. Somit lassen sich etwa für verschiedene Kunden unterschiedliche Programmlogiken ausführen. Listing 5.1 zeigt exemplarisch eine Prozedur, die Variabilität mithilfe von Feature Flags realisiert.

```

procedure PROCESSREQUEST(request)
    p  $\leftarrow$  commonLogic()
    if request.featureA then
        result  $\leftarrow$  calcA(p)
    else
        if request.featureB then
            result  $\leftarrow$  calcB(p)
        end if
    end if
    return result
end procedure

```

Listing 5.1: Anfrageverarbeitung auf Basis von Feature Flags

Features können etwa in der Netzwerkanfrage übergeben werden und Produkte identifizieren (vgl. Abbildung 5.1). Je nach Auswahl können somit produktspezifische Pfade im Programmcode aufgerufen werden.

In der Arbeit von Wang et al. [22] nutzen 32 % der Befragten diese Möglichkeit, um mandantenspezifische Features auszuführen oder zwischen verschiedenen Varianten wie einer kostenfreien und einer kostenpflichtigen Version zu unterscheiden.

Vorteil dieser Architektur ist es, dass Features zur Laufzeit aktiviert und deaktiviert werden können. Auch können etwa einzelnen Mandanten zusätzliche Funktionalitäten zur Verfügung gestellt werden [22].

Durch die gemeinsame Verwaltung aller Varianten in einer Codebasis erhöht sich allerdings die Komplexität des Systems. Es besteht das Risiko, dass ein Microservice mit Unterstützung vieler Varianten in der gleichen Codebasis zu einem monolithischen System wächst, welches sich wiederum schwer warten lässt. Auch das Testen verschiedenster Kombinationen von Features birgt eine hohe Komplexität [22].

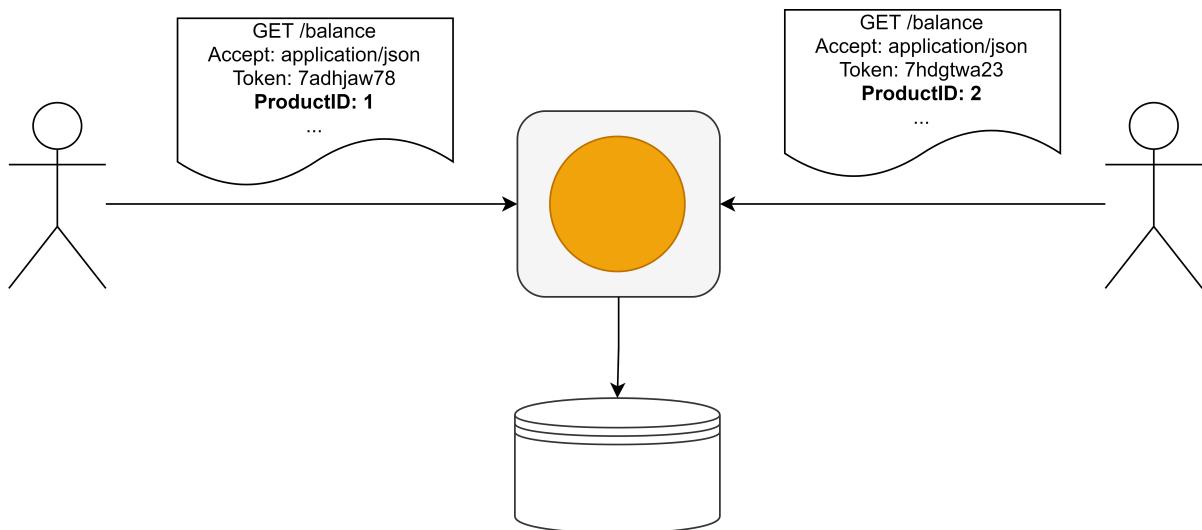


Abbildung 5.1: Verarbeitung mehrerer Varianten durch einen Microservice

5.2 Code-Duplikationen

Eine weitere Möglichkeit mehrere Varianten eines Produktes bereitzustellen, ist die Duplikation des bestehenden Quelltextes. Für jede neue Produktvariante wird der bestehende Code übernommen und an die spezifischen Anforderungen der neuen Variante angepasst. Verschiedene Varianten eines Produktes können sich somit unabhängig voneinander entwickeln. Die Weiterentwicklung oder Änderung eines Produktes hat somit keinen Einfluss auf die Codebasis einer anderen Produktvariante. Abbildung 5.2 zeigt die entsprechende Architektur im Kontext eines Microservices.

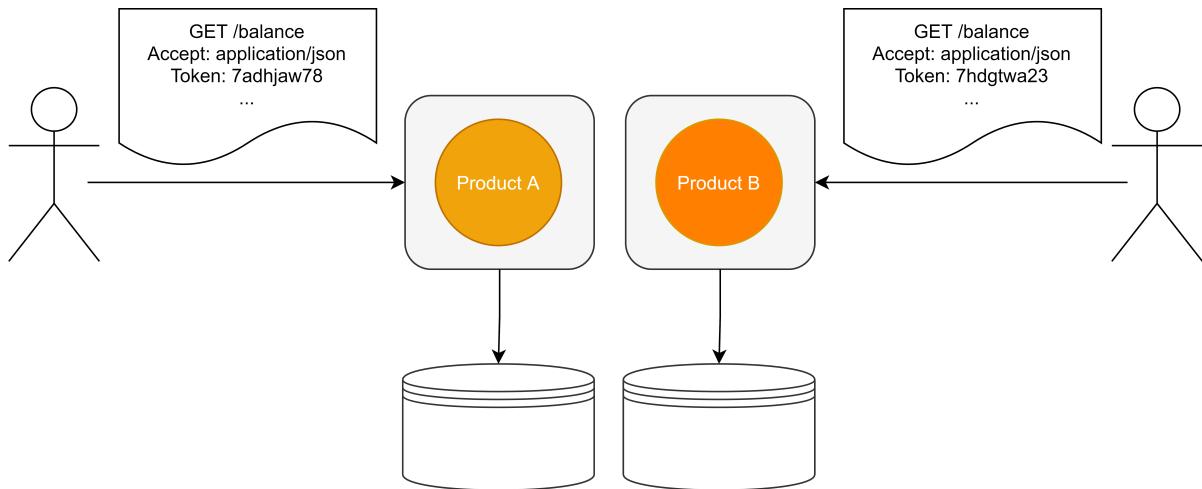


Abbildung 5.2: Verarbeitung mehrerer Varianten durch separate Microservices

Die Duplikation von Code zur Entwicklung mehrerer Produktvarianten ist in der Praxis weit verbreitet [22], [110]. Während das Klonen einer Software dem DRY-Prinzip [62] widerspricht, so wird auf diese Weise dennoch eine vollständige Unabhängigkeit der Produktvarianten erreicht. Insbesondere im Rahmen von Microservices, welche eine lose Kopplung propagieren, können Code-Duplikationen daher ein valides Mittel sein [63], [64], [65].

Darüber hinaus beschreiben die Teilnehmer der Studie von Dubinsky et al. [110], dass das Duplizieren von Code eine schnelle und günstige Möglichkeit ist, eine neue Produktvariante zu entwickeln. Die neue Variante basiert somit auf einer etablierten und getesteten Basis. Außerdem können Änderungen an der neuen Software gemacht werden, ohne dass andere Kopien der Anwendung davon beeinträchtigt werden.

Den initial geringen Entwicklungskosten stehen die steigenden Wartungskosten gegenüber. Durch die vollständige Trennung der Produktvarianten werden neben den Differenzen auch alle Gemeinsamkeiten der Produktvarianten unabhängig voneinander verwaltet. Änderungen oder Fehlerbehebungen an gemeinsamen Eigenschaften müssen somit in allen Quellcode-Kopien eingepflegt werden [110].

5.3 Konfigurierbare Instanzen

Mehrere Softwareumgebungen sind in der Industrie gängige Praxis. So ist üblich, dass etwa Entwicklungs-, Test- und Produktionsumgebungen existieren. Während in der Entwicklungs-umgebung oft die aktuellste Version einer zu entwickelnden Software bereitgestellt wird, so werden Testumgebungen genutzt, um die Qualitätsanforderungen an die Software vor einem Produktionseinsatz zu verifizieren. Je nach Umgebung unterscheiden sich einige Infrastrukturelemente. Beispielsweise existieren separate Datenbanken in Entwicklung und Produktion. Um nicht für jede Umgebung die Software anpassen zu müssen, ist es üblich, Konfiguration und Quellcode voneinander zu trennen. Dieses Muster wird auch externe Konfiguration (engl. *externalized configuration*) genannt. Viele Frameworks, darunter auch Spring, bieten Unterstützung für die Externalisierung der Anwendungskonfiguration.

```
import org.springframework.stereotype.*;
import org.springframework.beans.factory.annotation.*;

@Component
public class MyBean {
    @Value("${name}")
    private String name;
    // ...
}
```

application.properties

```
1 | server.port=8000
2 | name=ProduktA
```

Listing 5.2: Java Bean mit externer Konfiguration [111]

Listing 5.2 zeigt exemplarisch eine Java Klasse, welche ein extern konfiguriertes Attribut `name` enthält. Der Wert dieser Variablen wird nicht statisch im Code festgehalten. Stattdessen wird die Belegung der Variable in einer separaten Konfigurationsdatei festgelegt.

In diesem Fall handelt es sich dabei um die `application.properties` Datei, welche vom Spring Framework geladen wird. Die im Quelltext markierten Attribute werden automatisch mit den in der Konfigurationsdatei definierten Werten belegt.

Auf diese Weise lassen sich mehrere Services mit unterschiedlichen Konfigurationen verteilen. Abbildung 5.3 zeigt zwei Microservice-Instanzen, welche mit unterschiedlichen Konfigurationen bereitgestellt werden. Beide Instanzen basieren auf derselben Quellcodebasis und unterscheiden sich lediglich in ihrer Konfiguration, welche extern bereitgestellt wird.

Bildet man produktsspezifische Eigenschaften variabel im Quellcode ab, lassen sich somit mehrere Produktvarianten über unterschiedliche Konfigurationen unterstützen. Durch die gemeinsame Codebasis müssen Erweiterungen und Bugfixes an den Gemeinsamkeiten des Systems außerdem nur einmalig angewendet werden, was den Wartungsaufwand verringert.

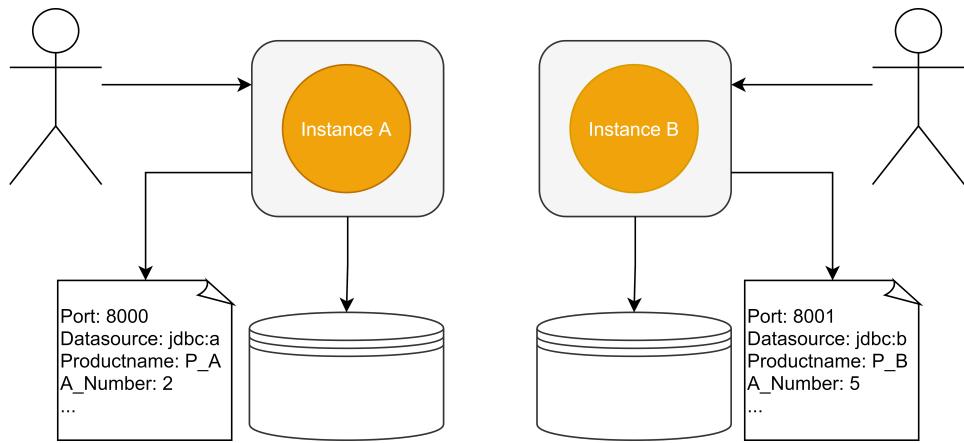


Abbildung 5.3: Microservice-Instanzen mit verschiedenen Konfigurationen

5.4 Shared Libraries

Eine Alternative zu Code-Duplikationen (vgl. Abschnitt 5.2), die das DRY-Prinzip verfolgt, ist die Nutzung von gemeinsam genutzten Bibliotheken (engl. *Shared Libraries*) [66, S. 112], [67].

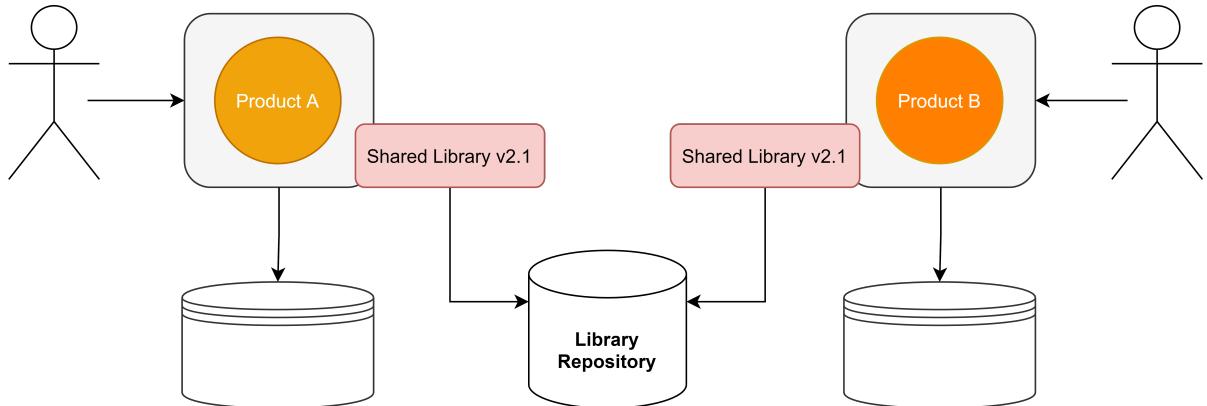


Abbildung 5.4: Microservices mit gemeinsam genutzter Bibliothek

Abbildung 5.4 zeigt exemplarisch zwei Microservices, welche jeweils eine geteilte Bibliothek verwenden. Funktionalitäten, die von mehreren Services benötigt werden, werden in die Bibliothek ausgelagert. Jeder Microservice kann bei Bedarf diese Bibliothek nutzen. Gleiche Funktionalitäten müssen somit nicht mehrfach in jedem Microservice umgesetzt werden. Im Rahmen der Entwicklung von mehreren Varianten eines Produktes stellt dies eine Möglichkeit dar, Gemeinsamkeiten und Unterschiede zu verwalten. Während Gemeinsamkeiten aller Produktvarianten in einer geteilten Bibliothek gekapselt werden, implementiert jeder Microservice die Differenzen, die die spezifische Produktvariante ausmachen.

Dabei ist allerdings zu beachten, dass die Nutzung geteilter Bibliotheken eine Kopplung zwischen Microservices einführt, was auch dazu führt, dass dieser Ansatz Kritik ausgesetzt ist

[68, S. 13 f.], [69], [70], [71]. Im Zusammenhang mit geteilten Bibliotheken beschreibt Krull [112] zwei wesentliche Kritikpunkte:

1. Geteilte Bibliotheken können komplexe Logik enthalten, die von allen Services benötigt wird. Je nach Änderung kann es dazu kommen, dass alle Services zur gleichen Zeit die Bibliothek aktualisieren müssen, um inkompatible Versionen zu vermeiden.
2. Projekte können in den Status der Abhängigkeitshölle (ugs. *dependency hell*) geraten. Dies geschieht, wenn die Bibliothek eigene Abhängigkeiten enthält, die mit den Abhängigkeiten des Projektes nicht kompatibel sind.

Darüber hinaus ist eine Voraussetzung für die Nutzung geteilter Bibliotheken, dass alle Microservices in der gleichen Programmiersprache entwickelt sind bzw. die gleiche Plattform nutzen [66, S. 112].

De Toledo et al. [70] untersuchen in ihrer Arbeit anhand von vier Unternehmen, wie geteilte Bibliotheken in der Praxis verwendet werden, um die Agilität zu verbessern. Alle Teilnehmer der Studie berichten, dass die Nutzung von geteilten Bibliotheken für eine hohe Kopplung sowohl zwischen Microservices als auch zwischen den einzelnen Teams führt. Nicht abwärtskompatible Änderungen an der Bibliothek führen etwa dazu, dass alle betroffenen Microservices koordiniert auf die neuste Version der Bibliothek aktualisiert werden müssen. Die Teilnehmer der Studie plädieren deshalb dafür, dass geteilte Bibliotheken nur eingesetzt werden, wenn alle Alternativen ausgeschlossen wurden.

5.5 Shared Service

Als mögliche Alternative zu geteilten Bibliotheken schlagen de Toledo et al. [70] einen gemeinsam genutzten Service (engl. *Shared Service*) vor. Taibi und Lenarduzzi [71] beschreiben geteilte Bibliotheken auch als „Bad Smell“ [113, S. 71] und beschreiben die Einführung eines geteilten Services ebenfalls als eine bessere Vorgehensweise.

Abbildung 5.5 zeigt exemplarisch eine Microservice-Architektur unter Verwendung eines gemeinsamen Services. Anstelle einer gemeinsam genutzten Bibliothek, greifen beide Microservices in diesem Fall auf einen dritten Service zu. Dieser Service kann im Rahmen der Produktentwicklung etwa gemeinsam genutzte Funktionalitäten implementieren, welche von allen Produktvarianten benötigt werden.

Im Gegensatz zu den gemeinsamen Bibliotheken hat diese Vorgehensweise den Vorteil, dass sich etwa Fehlerbehebungen auf den gemeinsamen Service beschränken. Alle Clients, die den geteilten Service nutzen, profitieren von der Fehlerbehebung, ohne dass diese neu bereitgestellt werden müssen [22].

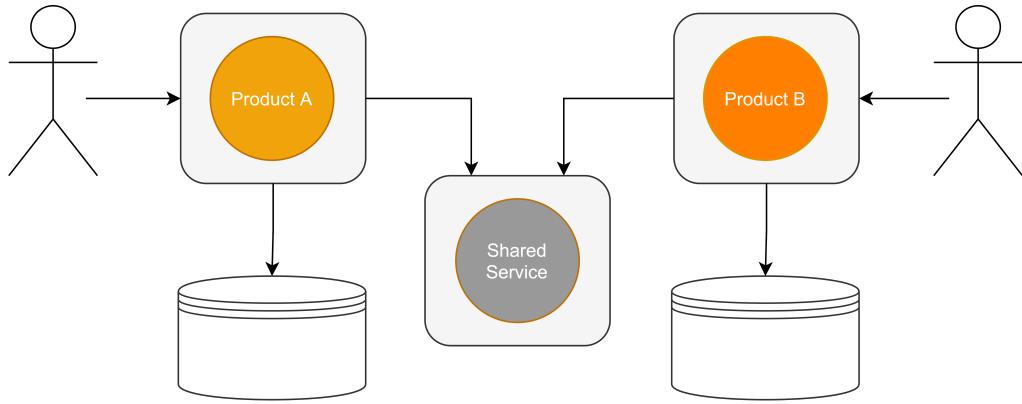


Abbildung 5.5: Verwendung eines gemeinsamen Services durch zwei Microservices

Darüber hinaus kann der geteilte Service auch ohne Kenntnis der Clients weiterentwickelt werden, sofern sich die Schnittstelle des Services nicht ändert. Um diesen Vorteil auszunutzen, ist es entsprechend wichtig, dass Änderungen an der Schnittstelle des geteilten Services nur in Ausnahmefällen vorgenommen werden [70].

Bei Einführung eines weiteren Services muss beachtet werden, dass dies zu erhöhter Netzwerkkommunikation führt und die Performance des Systems beeinflussen kann [22]. Wird ein geteilter Service etwa von vielen Client-Microservices genutzt, stellt dieser einen potenziellen Flaschenhals dar, welcher die Antwortzeiten von allen abhängigen Microservices beeinflusst. Auch der Ausfall des geteilten Services führt dazu, dass entsprechend alle abhängigen Clients beeinträchtigt werden.

5.6 Sidecar

Ein Ansatz, den Nachteilen des Shared Services zu begegnen, ist das Sidecar-Muster. Der Name des Musters ist an den Beiwagen eines Motorrads (engl. *sidecar*) angelehnt. In ähnlicher Weise wird auch ein Sidecar-Service parallel zur Hauptanwendung betrieben, um diese zu unterstützen.

Insbesondere im Kubernetes Umfeld ist dies ein verbreitetes Muster, um Querschnittsthemen wie Logging und Authentifizierung aus der Hauptanwendung zu extrahieren [114].

Abbildung 5.6 zeigt die Nutzung von Sidecars durch zwei Microservices. Jedem Microservice wird dabei ein separates Sidecar zugeordnet. Der Hauptservice und das Sidecar bilden eine Einheit und werden auf der gleichen Maschine betrieben. Im Kubernetes Umfeld wird dies etwa durch einen Pod (vgl. Unterabschnitt 2.3.2) abgebildet [115, S. 11]. Der Container der Hauptanwendung und der Sidecar-Container teilen sich in diesem Fall unter anderem Festplattenspeicher, Netzwerkumgebung und Dateisystem [115, S. 11].

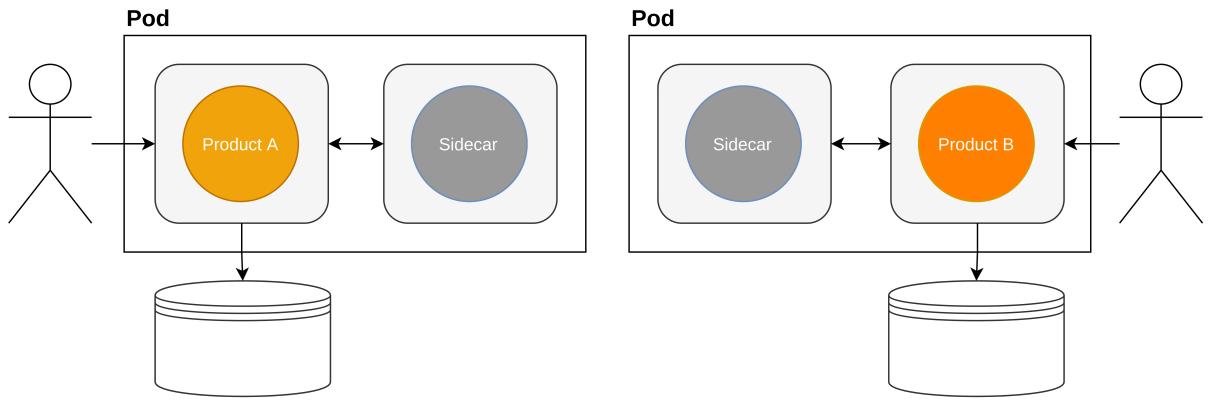


Abbildung 5.6: Verwendung des Sidecar-Musters durch zwei Microservices

Die Verwendung dieses Musters stellt in zwei Aspekten eine Verbesserung gegenüber dem Shared Service (vgl. Abschnitt 5.5) dar. Durch das Betreiben der beiden Container auf einem Hostsystem werden Netzwerklatenzen minimiert, was die Antwortzeiten verringert. Darüber hinaus besitzt jeder Microservice seine eigene Instanz des geteilten Services. Bei hoher Auslastung oder Ausfall eines Sidecars ist somit nur der zugeordnete Microservice betroffen. Alle weiteren Microservices greifen weiterhin auf ihre eigene Instanz des geteilten Services zu.

Im Gegensatz zu einem geteilten Service müssen allerdings mehrere Bereitstellungen verwaltet werden, was gegebenenfalls zu erhöhtem Wartungsaufwand und Ressourcenverbrauch führt.

Die Nutzung von Sidecars als Alternative zu geteilten Services ist in der Praxis noch relativ unbekannt. In der Studie von Wang et al. [22] haben lediglich 9 % der Befragten begonnen, dieses Muster in ihren Anwendungen einzusetzen. Die Autoren beschreiben Sidecars als „vielversprechende und elegante“ Lösung, um Gemeinsamkeiten zu verwalten [22].

5.7 Software-Produktlinien

Clements und Northrop [116] bezeichnen Software-Produktlinien als eine Reihe von Produkten, die auf Basis von gemeinsamen Ressourcen entwickelt werden [116, S. 5]:

“A software product line is a set of software-intensive systems that share a common, managed set of features satisfying the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way.”

Clements & Northrop, 2002

Dabei beschreibt die Produktlinienentwicklung einen Entwicklungsansatz, der sich unter anderem auf die Wiederverwendbarkeit von Komponenten in einer Produktfamilie konzentriert. Die Produkte einer Produktlinie weisen typischerweise große Gemeinsamkeiten auf und lassen sich somit aus den gleichen wiederverwendbaren Bausteinen erstellen.

Die Motivation für die Erstellung einer Produktlinie ist primär wirtschaftlich begründet. Eine hohe Wiederverwendbarkeit sorgt dafür, dass neue Varianten eines Produktes schneller erstellt werden können und weniger Entwicklungsaufwand notwendig ist [117, S. 3 f.], [118, S. 9].

Die Produktlinienentwicklung ist kein Thema, das der Softwareentwicklung entstammt. Bereits lange zuvor existierten Produktlinien etwa in der Fertigung von Autos. Diese lassen sich nach den Wünschen des Kunden konfigurieren. So können Farbe, Leistung und Ausstattung frei gewählt und kombiniert werden. Auf diese Weise sind mehrere tausend unterschiedliche Kombinationen denkbar [118, S. 5].

In der Softwareentwicklung gewann das Konzept von Produktlinien erstmals in den frühen 1990er Jahren an Bedeutung [118, S. 5 f.], [119]. Eines der bekanntesten Beispiele für eine Software-Produktlinie ist der Linux Kernel [120]. Dieser bietet mehr als 11.000 Features, welche sich konfigurieren lassen.

Ein Feature beschreibt eine Charakteristik oder ein für den Endnutzer sichtbares Verhalten eines Softwaresystems [118, S. 18]. Das Produkt einer Produktlinie ist die Kombination mehrerer Features. Darüber hinaus können Features Vorbedingungen unterliegen. Beispielsweise ist es möglich, dass nur eines von zwei bestimmten Features verwendet werden kann. Die Modellierung von Features einer Produktlinie wird in Feature-Modellen vorgenommen. Diese beschreiben die vorhandenen Features und deren Zusammenhang untereinander.

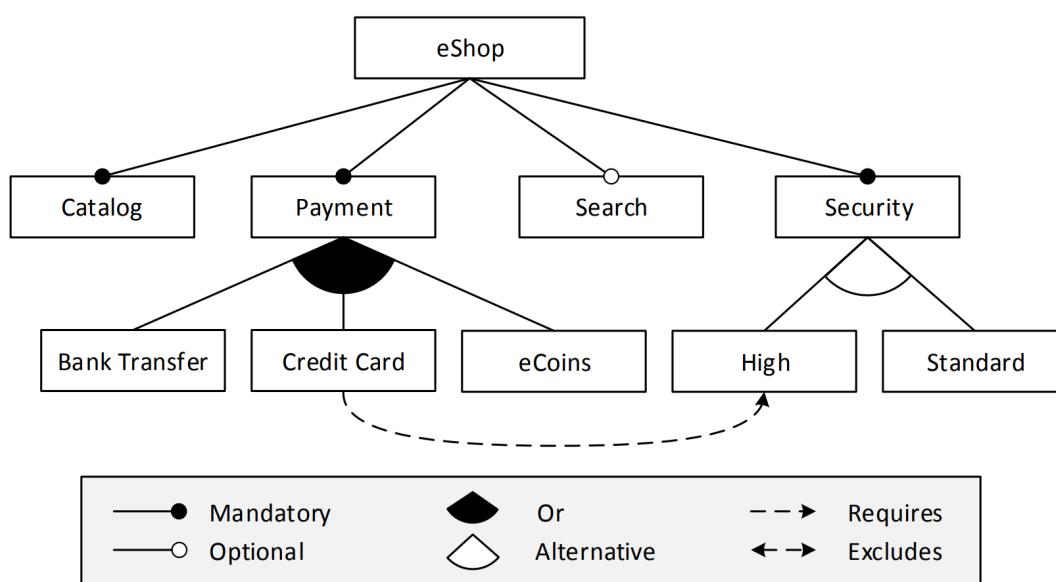


Abbildung 5.7: Feature-Modell eines Webshops [121]

Abbildung 5.7 zeigt exemplarisch das Feature-Modell eines Webshops. Das Modell beschreibt die Features des Shops und darüber hinaus die Variabilität der Produktfamilie. Dafür enthält das Modell Variationspunkte. Ein Beispiel dafür ist die Bezahlung. Ein Shop kann Bezahlungen etwa über Überweisungen, Kreditkarten oder virtuelle Währungen annehmen. Potenzielle Betreiber können somit frei entscheiden, welche der Zahlungsmittel sie in ihrer Produktvariante unterstützen möchten. Einige Features stehen dabei miteinander in Beziehung. Die Zahlung mit Kreditkarte ist etwa nur möglich, wenn die Sicherheit der Verbindung als hoch eingestuft wird (etwa eine TLS-Verbindung).

Die Entwicklung von Software-Produktlinien teilt sich in zwei Bereiche auf: die Domänenentwicklung (engl. *Domain engineering*) und die Applikationsentwicklung (engl. *Application engineering*).

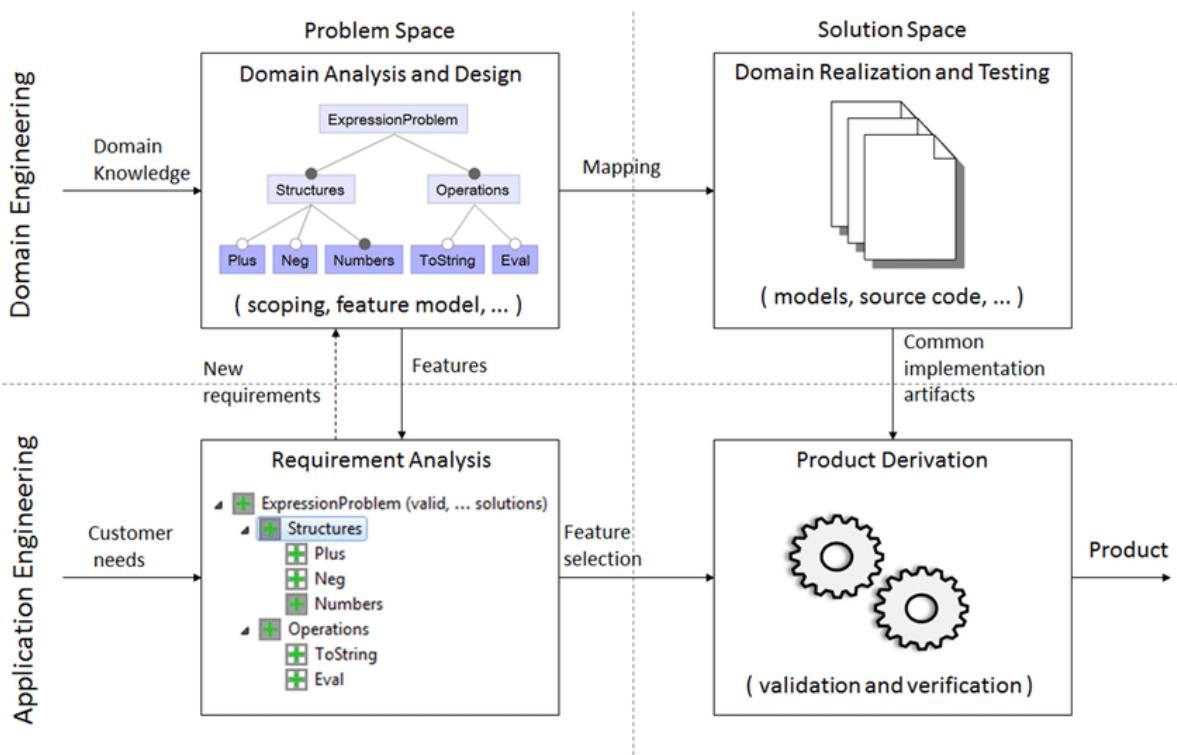


Abbildung 5.8: Überblick über die Entwicklung von Software-Produktlinien [118, S. 20]

Die Entwicklung der Domäne umfasst die Analyse der Produktfamilie. Dazu werden Gemeinsamkeiten und Variabilitäten ermittelt und als wiederverwendbare Einheiten implementiert. Das Ergebnis der Domänenentwicklung ist kein fertiges Produkt, sondern eine Plattform auf deren Basis mehrere Produkte erstellt werden können [118, S. 21].

Mit der Erstellung eines spezifischen Produktes beschäftigt sich die Applikationsentwicklung. Für einen Kunden werden die benötigten Features ermittelt und ausgewählt. Noch nicht vorhandene Features müssen dabei gegebenenfalls im Rahmen der Domänenentwicklung ergänzt werden. Aus der Auswahl der Features wird dann ein spezifisches Produkt erstellt. Dieser

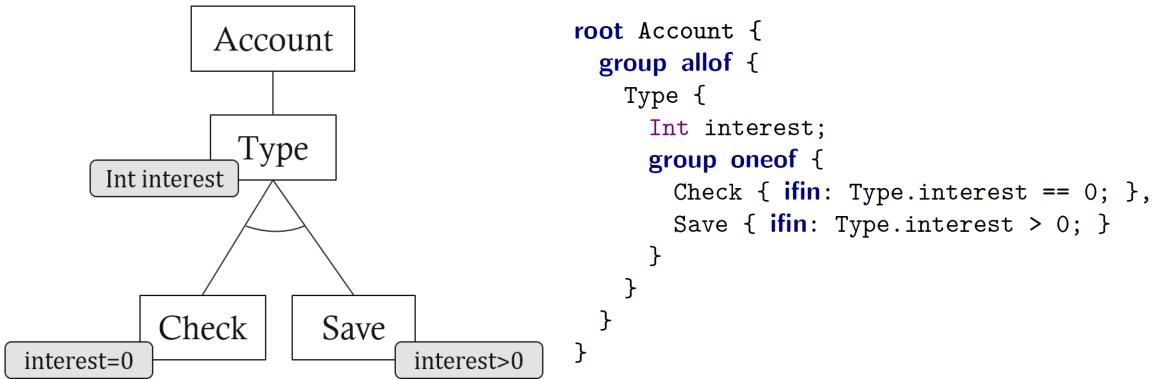


Abbildung 5.9: Feature-Modell mit zugehöriger ABS-Modellierung [72]

Vorgang wird im besten Fall durch Automatisierung unterstützt, um manuelle Aufwände zu vermeiden und die Produktentwicklungszeit somit zu minimieren.

Variabilität innerhalb der Produktfamilie lässt sich dabei auf unterschiedliche Weise realisieren [117], [118], [122]. Apel et al. [118] etwa beschreiben mehrere Muster, um Variabilität in der Implementierung zu erreichen. Neben klassischen Mechanismen wie Feature Flags und Entwurfsmustern [118, S. 99 ff.] gehören dazu auch erweiterte Methoden wie feature-orientierte Programmierung, aspektorientierte Programmierung und delta-orientierte Programmierung [118, S. 129 ff.].

Neben den sprachbasierten Möglichkeiten existieren toolbasierte Vorgehensweisen, um Variabilität zu erreichen. Dazu gehören etwa Versionsverwaltungen, Build-Skripte und Präprozessoren [118, S. 99 ff.].

Einen konkreten Ansatz, Microservices und Produktlinien zu verbinden, liefern Naily et al. [72]. Sie entwickeln das *ABS-Microservice-Framework*¹, welches auf der *Abstract Behavioral Specification* (ABS) [73] basiert. Die Spezifikation wurde explizit für die Modellierung von Variabilitäten entwickelt und eignet sich somit insbesondere für die Entwicklung von Software-Produktlinien [123].

Um Gemeinsamkeiten und Variabilitäten abzubilden nutzt die ABS Core-Module und Delta-Module. Core-Module enthalten die Gemeinsamkeiten der Produktlinie, Delta-Module implementieren die Variabilitäten einzelner Produktvarianten.

Abbildung 5.9 zeigt das Feature-Modell eines Bankkontos mit der zugehörigen ABS-Beschreibung. Je nach Typ des Bankkontos ist die Zinsrate entweder 0 (Girokonto) oder größer 0 (Sparkonto). Diese Variabilität wird in Delta-Modulen abgebildet. Das Delta-Modul des Sparkontos etwa überschreibt das Attribut `interest` mit einem positiven Wert.

¹<https://github.com/afifun51/abs-microservices-framework>

Auf Basis der Modellierung eines Produktes, welche Features und zugehörige Deltas enthält, werden die entsprechenden Microservices mithilfe eines Code-Generators erstellt.

Die Autoren stellen fest, dass sich Änderungen an mehreren Produktvarianten mit dem ABS-Microservice-Framework effizienter umsetzen lassen, da lediglich die Feature- und Delta-Module einmalig erweitert werden müssen. Die Generierung kann dann für alle Produktvarianten vorgenommen werden. Manuelle Änderungen an mehreren Produktvarianten sind somit nicht notwendig.

6 Konzept

Dieses Kapitel beschäftigt sich mit Kriterien, welche den Auswahlprozess einer geeigneten Modernisierungsstrategie unterstützen. Sowohl für die Migration (vgl. Kapitel 4) als auch für die Variabilität (vgl. Kapitel 5) konnten mehrere mögliche Ansätze identifiziert werden. Keiner dieser Ansätze und Vorgehensweisen kann pauschal als richtig oder falsch bezeichnet werden. Vielmehr hängt es von der individuellen Situation eines jeden Projektes ab, welche Strategie sich als sinnvoll erweist.

Um den Auswahlprozess zu erleichtern, werden die verschiedenen Ansätze im Folgenden anhand mehrerer Bewertungskriterien ausgewertet und verglichen. Dabei werden die Dekompositions-, Prozess- und Variabilitätsstrategien getrennt betrachtet, da sich diese größtenteils unabhängig voneinander kombinieren lassen.

6.1 Bewertung der Dekompositionsstrategien

Für die Bewertung der Dekompositionsstrategien wurde untersucht, in welchen Aspekten die verschiedenen Vorgehensweisen relevante Unterschiede aufweisen. Dabei wurden vier Kriterien identifiziert: die Anwendbarkeit, die vorhandene Tool-Unterstützung, die Komplexität des bestehenden Systems und die Qualität der Codebasis. Die Auswirkungen und Unterschiede der ermittelten Kriterien auf die jeweiligen Dekompositionsstrategien werden im Folgenden einzeln ausgewertet.

6.1.1 Anwendbarkeit

Ein essenzielles Kriterium für die Auswahl einer Dekompositionsstrategie ist die Anwendbarkeit. Nicht jede Vorgehensweise lässt sich universell anwenden. Sind bestimmte Rahmenbedingungen nicht erfüllt, können einige der Ansätze bereits im Vorfeld verworfen werden, da diese nicht auf das eigene Projekt anwendbar sind.

Die Anwendbarkeit in den identifizierten Ansätzen wird vor allem durch die Verfügbarkeit der benötigten Eingabedaten bestimmt. Ansätze der statischen Analyse sind etwa auf den Quelltext eines vorhandenen Systems angewiesen. Dynamische Analyseansätze nutzen in der

Regel Logs, welche das dynamische Verhalten der Anwendung protokollieren. Im Falle der vollständigen Neuentwicklung einer Anwendung ohne bestehenden Monolithen lassen sich diese Strategien nicht nutzen, da kein System vorhanden ist, welches analysiert werden kann. Auch im Rahmen von Brownfield-Entwicklungen lassen sich statische und dynamische Analysemöglichkeiten nicht immer anwenden. Trotz eines bestehenden Systems ist etwa die Verfügbarkeit von detaillierten Logs oder der Zugang zum Quelltext nicht immer gewährleistet. Darüber hinaus existieren je nach Ansatz Einschränkungen in Bezug auf die genutzten Technologien. Viele Arbeiten erwarten etwa die Nutzung bestimmter Programmiersprachen oder Frameworks, um Microservice-Kandidaten zu extrahieren. Eine universelle Einsetzbarkeit ist somit nicht gegeben.

Tabelle 6.1 zeigt die Liste der identifizierten Dekompositionsansätze und deren benötigte Eingabedaten. Je nach Verfügbarkeit der Daten kann somit bereits eine Vorauswahl der zur Verfügung stehenden Ansätze getroffen werden. Sofern möglich kann etwa eine Kombination aus statischer und dynamischer Analyse gewählt werden, um möglichst viele Informationen über das bestehende Legacy-System zu sammeln.

Ist kein bestehender Monolith vorhanden, sind dennoch einige Ansätze anwendbar. Insbesondere die Anwendung Service Cutter [90] wird als allgemein anwendbar beschrieben [40]. Die Autoren selbst beschreiben die Greenfield-Entwicklung als ein geeignetes Szenario für die Nutzung von Service Cutter [126]. Das Tool arbeitet mit einem Entity-Relationship-Modell sowie optionalen Use-Case-Beschreibungen des neuen Systems und generiert auf dieser Basis Microservice-Kandidaten, welche sich anschließend implementieren lassen.

Darüber hinaus lassen sich auch Methoden des DDD unabhängig von einem bestehenden System anwenden. Da sich das DDD auf die Modellierung der fachlichen Domäne konzentriert, sind Eingabedaten wie bestehender Quelltext nicht erforderlich.

6.1.2 Tool-Unterstützung

Ein weiteres Kriterium für die Auswahl einer Dekompositionsstrategie ist das Vorhandensein einer Tool-Unterstützung.

Grundsätzlich beschreiben die meisten Arbeiten einen Prozess, welcher durch Algorithmen unterstützt wird. Die automatische Verarbeitung von statischen und dynamischen Informationen ist allerdings kein trivialer Prozess, welcher ohne entsprechende Fachkenntnisse implementiert werden kann. Gleichermaßen gilt ebenfalls für die modellgetriebenen Strategien.

Fast alle Arbeiten erstellen auf Basis der vorhandenen Informationen einen Graphen, welcher anschließend in mehrere Partitionen unterteilt wird. Sowohl für die Erstellung des Graphen als auch für die anschließende Partitionierung werden Algorithmen eingesetzt, welche oft nur

Tabelle 6.1: Dekompositionsstrategien und benötigte Eingabedaten

Nr.	Titel	Benötigte Daten
1	Service Cutter: A Systematic Approach to Service Decomposition [90]	Entity-Relationship-Modell
2	Microservices Identification Through Interface Analysis [91]	OpenAPI-Spezifikation
3	A Dataflow-Driven Approach to Identifying Microservices from Monolithic Applications [29]	Datenfluss-Diagramm
4	A Feature Table Approach to Decomposing Monolithic Applications into Microservices [27]	Quelltext
5	A Decomposition and Metric-Based Evaluation Framework for Microservices [17]	Trace-Log
6	From a Monolith to a Microservices Architecture: An Approach Based on Transactional Contexts [124]	Quelltext
7	A Microservice Decomposition Method Through Using Distributed Representation of Source Code [81]	Quelltext
8	Microservice Decomposition via Static and Dynamic Analysis of the Monolith [54]	Quelltext, Zugriff auf Anwendung
9	Migrating Web Applications from Monolithic Structure to Microservices Architecture [31]	Quelltext, Trace-Log
10	Tool Support for the Migration to Microservice Architecture: An Industrial Case Study [16]	Quelltext
11	Determining Microservice Boundaries: A Case Study Using Static and Dynamic Software Analysis [88]	Quelltext, Trace-Log
12	Mono2Micro: A Practical and Effective Tool for Decomposing Monolithic Java Applications to Microservices [83]	Trace-Log
13	Identification of Microservices from Monolithic Applications through Topic Modelling [80]	Quelltext
14	An Automatic Extraction Approach: Transition to Microservices Architecture from Monolithic Application [15]	Quelltext, Versionsverwaltung-Historie
15	A New Decomposition Method for Designing Microservices [125]	OpenAPI-Spezifikation
16	Towards a Technique for Extracting Microservices from Monolithic Enterprise Systems [79]	Quelltext
17	Extraction of Microservices from Monolithic Software Architectures [18]	Versionsverwaltung-Historie
18	Service Candidate Identification from Monolithic Systems Based on Execution Traces [86]	Trace-Log
19	Towards Automated Microservices Extraction Using Multi-objective Evolutionary Search [78]	Quelltext

schematisch als Pseudocode angegeben sind. Für die Anwendung eines solchen Ansatzes muss zunächst eine Implementierung der beschriebenen Vorgehensweise vorgenommen werden. Dies erfordert neben Zeit und Geld auch Sachkenntnis der Entwickler.

Al-Debagy und Martinek [81] nutzen etwa Methoden des maschinellen Lernens, um vorhandenen Quelltext zu analysieren und zusammenhängende Cluster zu finden. Die Anwendung dieser Methoden ist mangels Sachkenntnis und fehlender Zeit in der Praxis oft nicht realistisch.

In Fällen, in denen ein Ansatz aufgrund fehlender Fachkenntnisse - etwa im Bereich des maschinellen Lernens - nicht verfolgt werden kann, ist das Vorhandensein von unterstützenden Tools essenziell. Diese ermöglichen die Anwendung ohne Kenntnis über Details der zugrundeliegenden Implementierung.

Tabelle A.1 zeigt, welche der identifizierten Arbeiten ein Tool bereitstellen. Dabei ist zu beachten, dass lediglich Tools berücksichtigt werden, die Microservice-Kandidaten ermitteln. Im Gegensatz dazu existieren einige unterstützende Tools, die etwa benötigte Eingabedaten sammeln können. Krause et al. [54] nutzen die Anwendungen Structure101 und ExplorViz, um die statische Struktur des Legacy-Systems zu analysieren und die dynamischen Aufrufpfade zu visualisieren. Diese Tools können die Analyse des Legacy-Systems unterstützen, die Identifikation von geeigneten Microservice-Kandidaten bleibt in diesen Fällen allerdings ein manueller Vorgang. Die in Tabelle A.1 referenzierten Tools beziehen sich explizit auf die Dekomposition eines Monolithen in Microservices und unterstützen diesen Prozess, indem etwa eine Liste von potenziellen Microservice-Kandidaten ermittelt wird. Manuelle Aufwände sollen so minimiert werden.

Die Tool-Unterstützung der identifizierten Ansätze lässt sich in drei Kategorien einordnen:

Prototypischer Tool-Support

Etwa ein Viertel aller Arbeiten hat einen Prototyp entwickelt, um die vorgestellte Vorgehensweise zu demonstrieren und zu evaluieren. Dabei handelt es sich in der Regel um Projekte, die nicht aktiv entwickelt oder gewartet werden. Darüber hinaus sind die Prototypen in der Regel für einige spezifische Anwendungsfälle konzipiert und lassen sich nicht breit anwenden.

Matias et al. [88] etwa stellen das Tool MonoBreaker vor, welches auf Basis von statischen und dynamischen Informationen Microservice-Kandidaten ermittelt. Das Tool lässt sich allerdings nur für die Analyse von Python Anwendungen nutzen, welche das Django-Framework verwenden. Die Anwendungsmöglichkeiten sind entsprechend stark eingeschränkt.

Tool-Support

Neben prototypischen Implementierungen, existiert ein weiteres Viertel der Arbeiten, welches weiter ausgereiferte Produkte beschreibt. Service Cutter [90] gilt in der Literatur etwa als das

reifste Tool in Bezug auf die Dekomposition und wird auch als Stand der Technik beschrieben [59], [91].

Auch bei kommerziellen Produkten wie IBM Mono2Micro¹ [83] ist davon auszugehen, dass die Anwendung aktiv gewartet und entwickelt wird.

Weiter ausgereifte und verbreitete Tools unterliegen allerdings - analog zu den Prototypen - typischerweise ebenfalls Einschränkungen. Insbesondere wenn auf Basis eines bestehenden Monolithen gearbeitet wird, lassen sich in der Regel keine beliebigen Legacy-Systeme verwenden. Die in dieser Arbeit identifizierten Tools erwarten mehrheitlich das Vorhandensein eines Java-Monolithen. Darüber hinaus werden teilweise Anforderungen an genutzte Technologien und Frameworks gesetzt (vgl. Tabelle A.2). Ausnahme bilden Tools wie Service Cutter, welche mit abstrakteren Daten wie ER-Modellen arbeiten und somit unabhängig von einer spezifischen Programmiersprache sind.

Kein Tool-Support

Etwa die Hälfte aller identifizierten Arbeiten liefert keine direkte Unterstützung durch Tools oder beschreibt einen manuellen Prozess, der keine Tool-Unterstützung vorsieht. Für die Nutzung der vorgestellten Vorgehensweisen und Algorithmen muss in diesen Fällen gegebenenfalls eine eigene Implementierung geschaffen werden. Mangels Ressourcen und Wissen ist dies in vielen Praxisszenarien keine realistische Option.

6.1.3 Komplexität des Legacy-Systems

Ein weiteres Kriterium für die Auswahl einer Dekompositionsstrategie ist die Komplexität des bestehenden Systems. Grundsätzlich sollen Microservices anhand von Geschäftsfähigkeiten modelliert werden. Dies verspricht eine lose Kopplung zwischen mehreren Services. Aus diesem Grund ist etwa das DDD eine weit verbreitete Technik, um eine Anwendungsdomäne zu modellieren und mehrere Bounded Contexts (vgl. Abschnitt 2.2) zu ermitteln, welche jeweils durch einen Microservice abgebildet werden. Die Anwendung von Techniken des DDD wird oft als die empfohlene Vorgehensweise für eine gute Modellierung von Microservices genannt [21, S. 61 f.], [33, S. 56 ff.], [46, S. 79 f.], [99, S. 54 f.], [127].

Die Identifizierung von Subdomänen innerhalb einer Domäne kann aber insbesondere bei komplexen Legacy-Systemen eine Herausforderung darstellen. In diesen Fällen kann die Anwendung einer Dekompositionstechnik (vgl. Tabelle 6.1) einen Mehrwert bieten, indem versucht wird, zusammenhängende Komponenten innerhalb des Monolithen zu identifizieren. Insgesamt sind die vorgeschlagenen Vorgehensweisen und Tools allerdings nur eine unterstützende Hilfe [80], [124], [125]. Die ermittelten Microservice-Kandidaten erfordern eine

¹<https://www.ibm.com/de-de/cloud/mono2micro>

manuelle Bewertung und Diskussion. Gegebenenfalls muss der generierte Schnitt in diesem Zuge angepasst werden. Dies liegt daran, dass automatische und halbautomatische Prozesse die Fachlichkeit einer Anwendung nicht bewerten können. Ob ein identifizierter Cluster mehrerer Komponenten wirklich eine eigenständige Subdomäne der Anwendung darstellt, muss letztlich durch einen Menschen beurteilt werden.

Aus diesem Grund kann festgehalten werden, dass die Nutzung von Dekompositionstools vor allem dann empfehlenswert ist, wenn die zu migrierende Anwendung ein komplexes System mit einer potenziellen unklaren Domäne ist. Insbesondere bei alten Anwendungen, welche über viele Jahre entwickelt wurden, ist ein Überblick über das System oft nicht mehr möglich [26]. In diesen Fällen können Dekompositionstools eine gute Möglichkeit sein, die Struktur des Systems zu analysieren und potenziell unbekannte Abhängigkeiten zwischen verschiedenen Modulen offenzulegen.

Fritzsch et al. [57] führen in ihrer Studie Interviews mit 16 Teilnehmern aus zehn Unternehmen durch. Sie analysieren, welche Vorgehensweisen und Intentionen für die Migration hin zu einer Microservice-Architektur gegeben waren. Dabei zeigt sich auch, dass sich die Anzahl der Microservices je nach Projekt stark unterscheiden kann. Ein Produkt wurde in lediglich sechs Microservices unterteilt, während im größten Projekt etwa 250 Microservices implementiert wurden.

Während sich überschaubare Anwendungen mit wenigen Microservices gut manuell mit Methoden des DDD modellieren lassen, sind statische und dynamische Analysen eines bestehenden Legacy-Systems, welches in hunderte Microservices unterteilt wird, zu empfehlen. Eine manuelle Analyse, die die bestehende Anwendungsdomäne vollständig umfasst, ist in Anbetracht der Größe des Systems schwer möglich.

Ein weiteres Beispiel dafür ist der Dekompositionsansatz von Al-Debagy und Martinek [125]. Diese versuchen Microservice-Kandidaten auf Basis der semantischen Ähnlichkeit der Schnittstellenoperationen zu ermitteln. Ihre Vorgehensweise evaluieren Sie unter anderem an der Schnittstelle von Amazon Web Services (AWS). Die Schnittstelle besteht aus 313 Operationen, welche in 47 Microservices unterteilt wurden. Für Daten dieser Größenordnung kann ein unterstützender Algorithmus einen Mehrwert bieten. Für kleine Schnittstellen, welche nur wenige Operationen anbieten, stehen Aufwand und Nutzen nicht in Verhältnis zueinander. Entsprechend kann eine manuelle Dekomposition in diesen Fällen zielführender sein.

6.1.4 Qualität und Wert der Codebasis

Neben der Komplexität des Legacy-Systems ist auch dessen Qualität ein notwendiges Bewertungskriterium für die Auswahl einer Dekompositionsstrategie. Insbesondere die halb- und vollautomatische Generierung von Microservice-Kandidaten auf Basis von statischen und

dynamischen Informationen des Legacy-Systems ist auf eine gute Qualität der bestehenden Codebasis angewiesen. Weist der bestehende Quelltext keine gute Struktur auf, wird das Ergebnis der Dekomposition entsprechend negativ beeinträchtigt.

Ein Softwaresystem ohne erkennbare Architektur wird auch „Big ball of mud“ [128] genannt. In solchen Systemen ist eine Erkennung von zusammenhängenden Modulen schwer möglich, da entweder keine dedizierten Module vorhanden sind, oder Modulgrenzen unkoordiniert übertreten werden.

“ A Big Ball of Mud is a haphazardly structured, sprawling, sloppy, duct-tape-and-baling-wire, spaghetti-code jungle. These systems show unmistakable signs of unregulated growth, and repeated, expedient repair. Information is shared promiscuously among distant elements of the system, often to the point where nearly all the important information becomes global or duplicated. The overall structure of the system may never have been well defined. If it was, it may have eroded beyond recognition. Programmers with a shred of architectural sensibility shun these quagmires. Only those who are unconcerned about architecture, and, perhaps, are comfortable with the inertia of the day-to-day chore of patching the holes in these failing dikes, are content to work on such systems. ”

Brian Foote & Jospeh Yoder, 1997

Während diese Systeme besonders von einem Refactoring - etwa in Form einer Microservice-Migration - profitieren würden, ist es gleichzeitig besonders schwer, diesen Vorgang durch vorhandene Tools zu unterstützen. Auch in der Praxis ist dieses Problem bekannt. Bei schlechter Qualität des vorhandenen Quelltextes wird das bestehende System oft nicht berücksichtigt [58]. Stattdessen bietet in diesen Fällen eine Neuentwicklung die Chance, schlechte Designentscheidungen der Vergangenheit unbeachtet zu lassen und eine von Grund auf neue Architektur zu schaffen. In diesen Fällen ist Nutzung von modellgetriebenen Ansätzen ratsam, welche eine Dekomposition auf Basis von Software-Designartifakten der neuen Architektur vornehmen [60].

Darüber hinaus spielt auch der Wert des Quellcodes eine Rolle. Strategien, welche die Dekomposition eines bestehenden Systems vornehmen, bauen auf der bestehenden Codebasis auf. Dazu extrahieren und gruppieren die Algorithmen mehrere atomare Einheiten, etwa Methoden oder Klassen. Jede Gruppe dieser atomaren Einheiten stellt einen potenziellen Microservice-Kandidaten dar. Es handelt sich also um eine Extraktion auf Basis von bestehendem Quelltext.

Dies kann den Vorteil haben, dass für die Neustrukturierung der Anwendung weniger Implementierungsaufwand notwendig ist, da Quelltext des bestehenden Systems übernommen werden kann. Voraussetzung dafür ist es aber, dass der bestehende Quellcode einen hohen intellektuellen Wert hat [60], [129].

Zusammenfassend lässt sich somit festhalten, dass Struktur und Wert des bestehenden Monolithen maßgeblich für die Qualität der Microservice-Dekomposition mittels dynamischer und statischer Analyse verantwortlich sind. Weist das bestehende System eine schlechte Architektur auf, sind automatische Dekompositionsmethoden auf Basis des monolithischen Systems nicht zielführend. In diesen Fällen sollten stattdessen Ansätze genutzt werden, welche etwa auf Basis von Modellen der neuen Systemarchitektur Microservice-Kandidaten ermitteln.

Die gesammelten Bewertungskriterien sind zusammengefasst in Abbildung 6.1 dargestellt und geben einen Überblick über die Aspekte, welche bei der Auswahl einer Dekompositionssstrategie zu berücksichtigen sind. Dabei ist zu beachten, dass die Bewertung verschiedener

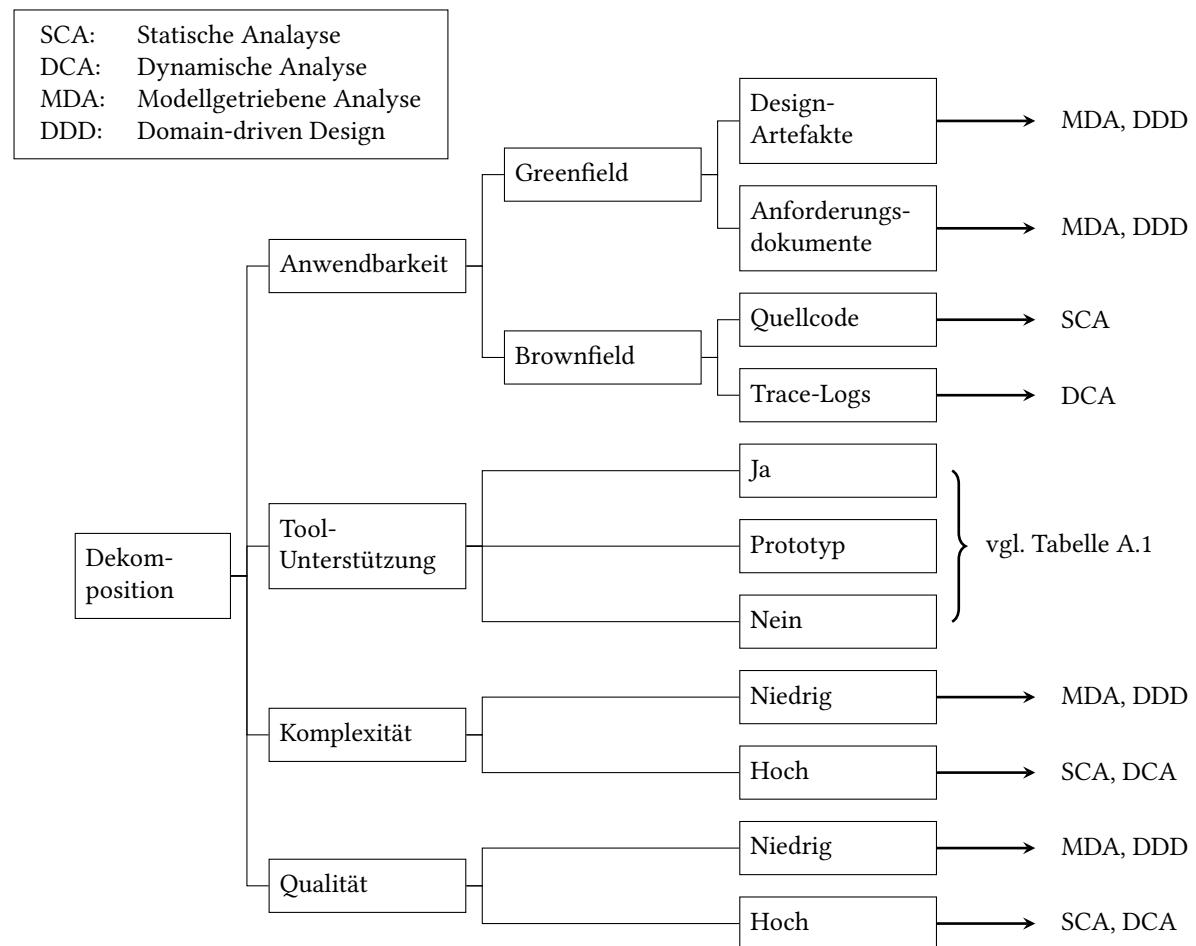


Abbildung 6.1: Bewertungskriterien für die Auswahl einer Dekompositionsstrategie

Kriterien, zu unterschiedlichen empfohlenen Dekompositionsstrategien führen kann. Eine hohe Komplexität der Anwendung legt unterstützende Dekompositionsmethoden auf Basis des bestehenden Quelltextes und Logdaten nahe. Ist die Qualität der bestehenden Codebasis

gleichzeitig von geringem Wert, kann dies für eine Neuentwicklung der Software unter Anwendung von Methoden des DDD sprechen. In Fällen, in denen je nach Bewertungskriterium eine andere Methodik der Dekomposition geeignet erscheint, muss eine Abwägung getroffen werden, welches Kriterium am stärksten wiegt. Dies lässt sich vorab nicht bestimmen und muss im Einzelfall anhand der bestehenden Rahmenbedingungen entschieden werden.

6.2 Bewertung der Prozessstrategien

Für die Bewertung der Prozessstrategien wurde ebenfalls ermittelt, inwiefern Unterschiede zwischen den identifizierten Vorgehensweisen bestehen. Eine Neuentwicklung mit Cut-over ist etwa mit einem höheren Risiko für neu auftretende Fehler verbunden als eine iterative Umstellung der Anwendung. Die Kritikalität der Anwendung ist somit ein wichtiger Aspekt bei der Auswahl einer Prozessstrategie.

Durch die Analyse relevanter Unterschiede der ermittelten Vorgehensweisen konnten vier Bewertungskriterien ermittelt werden: die Stabilität der Schnittstelle, die Komplexität sowie die Kritikalität des bestehenden Systems und die entstehenden Kosten.

Die Auswirkungen der unterschiedlichen Prozessstrategien auf die Bewertungskriterien werden im Folgenden betrachtet.

6.2.1 Schnittstellenstabilität

Ein Bewertungskriterium für die Auswahl einer geeigneten Prozessstrategie ist die Stabilität der externen Schnittstelle. Typischerweise stellt eine Anwendung eine Schnittstelle bereit, die von Kunden genutzt wird. Je nach Grad der Modernisierung können Änderungen an der externen Schnittstelle vorgenommen werden.

Die gewählte Prozessstrategie kann Einfluss darauf haben, wie viel zusätzlicher Aufwand für die Integration der neuen Microservices notwendig ist. Der Integrationsaufwand ist insbesondere bei Änderungen der Schnittstelle, welche das Protokoll betreffen, erhöht.

Als Beispiel diene ein Legacy-System, welches eine SOAP-Schnittstelle zur Verfügung stellt. Im Rahmen einer anstehenden Modernisierung soll eine Microservice-Architektur mit GraphQL-Schnittstelle etabliert werden. Wählt man in diesem Fall einen iterativen Vorgang, ergeben sich während der Migrationszeit einige Herausforderungen. Abbildung 6.2 zeigt die Situation im Rahmen der Anwendung des Strangler-Musters (vgl. Unterabschnitt 4.2.3). Eingehende Anfragen werden von der Strangler-Fassade entweder an den bestehenden Monolithen, oder an einen extrahierten Microservice weitergeleitet. Je nach Umsetzung der Fassade kann

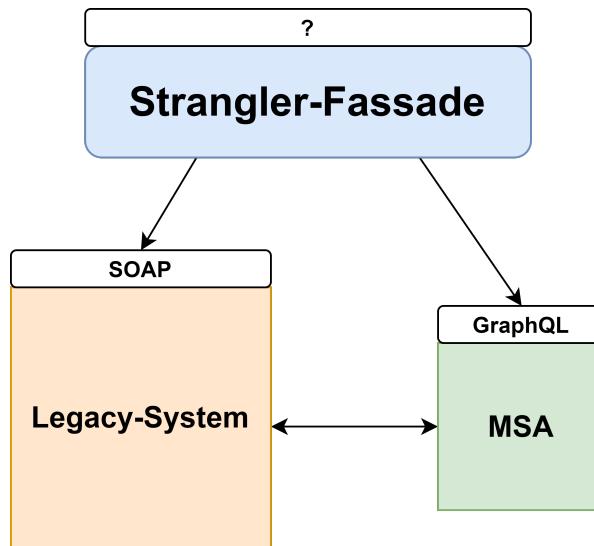


Abbildung 6.2: Strangler-Muster im Rahmen einer Modernisierung, welche die externe Schnittstelle betrifft

eine Weiterleitung an die entsprechenden Microservices auch durch den Monolithen selbst erfolgen.

Durch die unterschiedlichen Schnittstellen der alten und neuen Architektur ergibt sich ein zusätzlicher Aufwand in der Verwaltung und Implementierung der Strangler-Fassade. Grundsätzlich ist es möglich, dass die Fassade sowohl SOAP-Anfragen als auch GraphQL-Anfragen entgegennimmt. Erstere werden durch das Legacy-System verarbeitet, letztere durch die neue Microservice-Architektur. Da es sich bei dem Strangler-Muster um einen iterativen Prozess handelt, werden fortlaufend Services extrahiert. Während die Größe der alten SOAP-Schnittstelle kontinuierlich abnimmt, wächst die GraphQL-Schnittstelle des modernisierten Systems. Die externe Schnittstelle ist also nicht stabil und unterliegt frequenten Änderungen. Fortlaufende Änderungen der externen Schnittstelle erfordern somit eine Koordination und stetige Änderungen an den Clients, welche die externe Schnittstelle nutzen und sind somit nicht empfehlenswert.

Als weitere Option kann die externe Schnittstelle einheitlich auf Basis eines Protokolls gehalten werden. In diesem Fall empfängt die Strangler-Fassade etwa nur SOAP-Anfragen. Betrifft die Anfrage eine Funktionalität, die bereits in einen Microservice ausgelagert wurde, so muss eine Protokollumwandlung vorgenommen werden. Dies erfordert zusätzliche Implementierungen und ist mit einem erhöhten Aufwand verbunden.

Dieser Sachverhalt trifft auch auf die Prozessstrategie der Erweiterungen (vgl. Unterabschnitt 4.2.2) zu. Basieren die Schnittstellen des bestehenden Monolithen und der neuen Microservices auf unterschiedlichen Technologien und Protokollen, wird ein erhöhter Aufwand für die Integration notwendig.

In diesen Fällen kann die parallele Neuentwicklung (vgl. Unterabschnitt 4.2.1) von Vorteil sein. Die neue Microservice-Architektur wird parallel entwickelt und hat keinerlei Berührungs-punkte mit dem bestehenden Legacy-System. In einem Cut-over-Prozess wird der Betrieb anschließend von der alten auf die neue Anwendung migriert. Zusätzlicher Aufwand in Form von Integrationen, welche etwa Protokolltransformationen zwischen Monolith und Micro-services vornehmen, ist in diesem Fall nicht notwendig. Dennoch ist auch dieser Vorgang mit Aufwand für die Clients verbunden. Nach Etablierung der neuen Schnittstelle müssen die Aufrufe auf Basis des neuen Protokolls vorgenommen werden. Die Umstellung erfolgt aber lediglich einmalig für die gesamte externe Schnittstelle der Anwendung und nicht auf Basis einzelner Funktionalitäten.

6.2.2 Komplexität des Legacy-Systems

Je komplexer ein System ist, desto mehr Risiken sind bei einer Umstellung vorhanden. Mit der Komplexität steigt auch die Wahrscheinlichkeit, dass Randfälle nicht bedacht wurden und die neuen Implementierungen Fehler aufweisen. Die gesamte Funktionalität in einem Cut-over-Prozess umzustellen, ist deshalb ein kritischer Prozess, der mit vielen Risiken behaftet ist [103].

Die mit angrenzender Sicherheit erhöhte Fehlerquote nach einer Cut-over-Modernisierung beschreibt auch Martin Fowler:

“ If you do a big-bang rewrite, the only thing you’re guaranteed of is a big bang. ”

Martin Fowler

Besonders für Systeme, welche eine hohe Komplexität aufweisen, empfiehlt sich somit ein iteratives Vorgehen, um die Komplexität schrittweise in ein neues System zu verlagern. Gegebenenfalls auftretende Probleme können somit auf einen Teilbereich eingeschränkt werden und schrittweise behandelt werden.

In solchen Fällen bietet sich eine Migration etwa auf Basis des Strangler-Musters an. Ein Praxisbeispiel dafür ist Amazon, welche ihr monolithisches System über mehrere Jahre hin zu Microservices migriert haben [99, S. 432]. Auch die Microservice-Migrationen vieler Teilnehmer in der Studie von Fritzsch et al. [57] sind schrittweise über mehrere Jahre geplant.

Während eine iterative Vorgehensweise generell empfohlen wird, sind allerdings Fälle zu beachten, in denen dies als nicht realistisch beschrieben wird. Das hängt insbesondere mit der Qualität der bestehenden Codebasis zusammen (vgl. Unterabschnitt 6.1.4). Ein Teilnehmer der

Studie von Fritzsch et al. [57] berichtet etwa, dass die bestehende Codebasis zu komplex war, um Services zu extrahieren: „The best consultant on earth can't grasp what they have built over 10 years“.

Für Systeme, die sich aufgrund von Komplexität oder schlechtem Design der bestehenden Codebasis nicht aufteilen lassen, bieten sich die verbleibenden Prozessstrategien an. Handelt es sich bei dem Monolithen um ein stabiles System, welches keine akuten Probleme wie mangelnde Skalierbarkeit aufweist, ist die Erweiterungsstrategie eine gute Möglichkeit, lediglich neue Funktionalitäten als Microservice zu implementieren. Insbesondere wenn Änderungen und Wartbarkeit am Legacy-System aufgrund der wachsenden Codebasis erschwert werden, kann dies eine vielversprechende Möglichkeit sein. Voraussetzung dafür ist, dass neue Funktionalitäten, welche als Microservice implementiert werden sollen, eine geeignete Subdomäne darstellen, welche sich sinnvoll als eigenständiger Microservice implementieren lässt.

Betreffen Änderungsanforderungen die bestehenden Subdomänen, die bereits im Monolithen implementiert sind, und ist die Wartbarkeit des bestehenden Systems aufgrund der schlechten Codebasis bereits stark eingeschränkt, kann und muss gegebenenfalls eine komplette Neuentwicklung in Betracht gezogen werden.

6.2.3 Kritikalität des Legacy-Systems

Neben der Komplexität ist auch die Kritikalität des Legacy-Systems entscheidend, um eine geeignete Prozessstrategie auswählen zu können. Die komplette Neuentwicklung einer hochkomplexen Anwendung, welche über viele Jahre gewachsen ist, ist potenziell fehleranfälliger, als eine schrittweise Migration (vgl. Unterabschnitt 6.2.2).

Ob eine Migration in bestimmten Fällen dennoch als Neuentwicklung umgesetzt werden soll, hängt auch von den Auswirkungen möglicher Fehler im Rahmen der Softwaremodernisierung ab.

Handelt es sich um ein kritisches System, welches eine hohe Verfügbarkeit gewährleisten muss, haben Softwarefehler besonders große Auswirkungen. Dies zeigt auch der 2017 von Tricentis veröffentlichte *Software Fail Watch* [130]. Die Autoren identifizierten 616 Softwarefehler in 314 Unternehmen, die Verluste von mehr als 1.7 Billionen US-Dollar verursacht haben.

Die hohen Kosten potenzieller Softwarefehler haben somit auch Auswirkungen auf die Auswahl einer Prozessstrategie. Haben Ausfälle schwere Auswirkungen zur Folge und ist die Kritikalität damit sehr hoch, sollte eine komplette Neuentwicklung mit einem anschließenden Cut-over-Prozess nach Möglichkeit vermieden werden.

Handelt es sich um ein stabiles aber komplexes System, aus welchem sich Microservices nur schwierig extrahieren lassen, ist die Erweiterungsstrategie die empfohlene Vorgehensweise, um neue Funktionalitäten hinzuzufügen, ohne die Komplexität weiter zu erhöhen und die

Wartbarkeit zu verschlechtern. Insbesondere im Bankenumfeld, in welchem oft noch alte Legacy-Anwendungen verbreitet sind [26], [105], kann sich dies anbieten. Dies bestätigt auch die Studie von Mazzara et al. [26]. Diese beschäftigten sich mit der Migration eines kritischen Softwaresystems der Danske Bank. Die bestehende Mainframe-Anwendung war weiterhin an die entwickelten Microservices angebunden, da eine vollständige Migration aufgrund der Komplexität des Systems nicht möglich war. Ziel der Autoren war es, schrittweise weitere Funktionalitäten aus dem Monolithen herauszulösen und so im weiteren Verlauf der Migration das Strangler-Muster zu verfolgen.

Werden kritische Systeme im Rahmen einer vollständigen Neuentwicklung mit anschließendem Cut-over-Prozess migriert, steigt hingegen der Testaufwand, da die gesamte Funktionalität der Anwendung zu einem Zeitpunkt umgestellt wird und anschließend möglichst fehlerfrei lauffähig sein muss.

In Bezug auf kritische Systeme sollte eine Neuentwicklung somit nur in Betracht gezogen werden, nachdem iterative Möglichkeiten ausgeschlossen wurden.

Handelt es sich hingegen um eine weniger kritische Anwendung, so sind entsprechend auch potenziell auftretende Fehler nach einer vollständigen Neuentwicklung leichter zu tolerieren. Dies kann etwa bei ausschließlich intern genutzten Anwendungen der Fall sein, welche keine direkten Auswirkungen auf Endkunden haben.

6.2.4 Kosten

Eine Motivation für eine Neuentwicklung oder das Refactoring einer Anwendung sind die technischen Schulden (engl. *technical debt*), welche im Laufe der Zeit entstehen. Die Metapher der technischen Schulden wurde durch Cunningham [131] geprägt, der erstmals die Komplexität einer Software mit Schulden im finanziellen Sinne in Verbindung brachte. Technische Schulden entstehen, wenn im Rahmen der Softwareentwicklung Entscheidungen getroffen werden, welche auf kurze Sicht die einfachste Lösung darstellen, insgesamt aber nicht die beste Lösung sind. Diese Entscheidungen führen im Laufe der Zeit zu erhöhten Aufwänden, was die Entwicklung neuer Funktionalitäten und die Wartung erschwert [132, S. 5].

Je nach Höhe der technischen Schulden einer Anwendung steigen somit die Kosten für die Wartung und das Hinzufügen neuer Funktionalitäten (vgl. Abbildung 6.3). Auch die Wahl der Prozessstrategie hat einen Einfluss auf die Architektur der modernisierten Anwendung und damit auch auf die entstehenden Kosten. Die verschiedenen Ansätze unterscheiden sich im Hinblick auf die zu erwartenden Entwicklungs- und Wartungskosten.

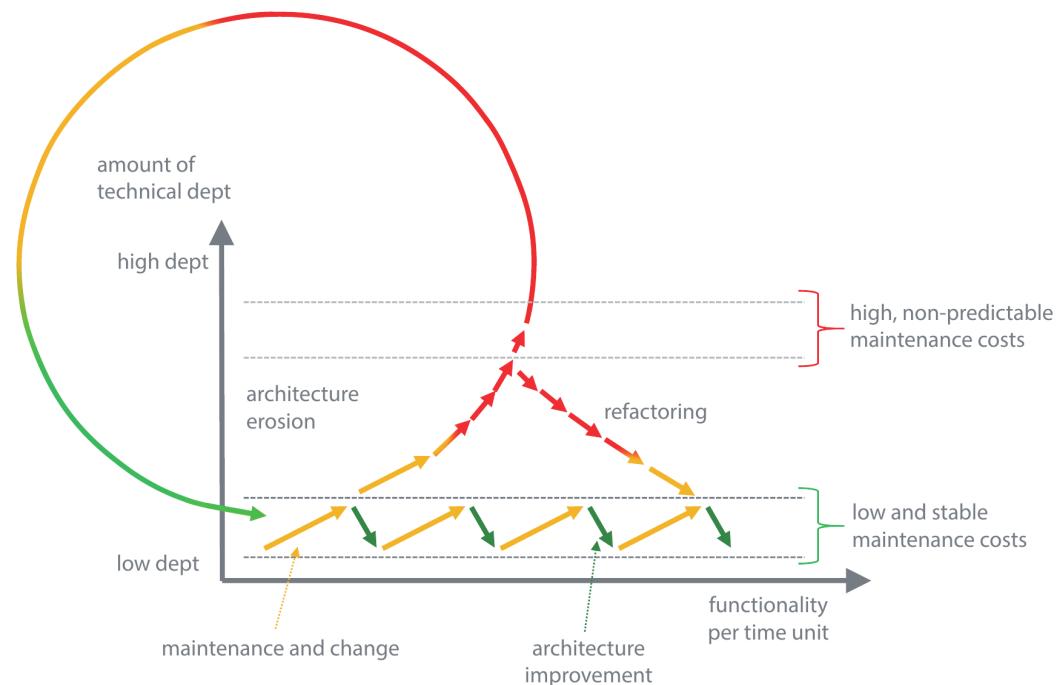


Abbildung 6.3: Technische Schulden und Architekturerosion [132, S. 5]

Entwicklungskosten

Die Entwicklungskosten beschreiben die Kosten, die für die reine Entwicklung der Microservice-Architektur anfallen. Da bei der Erweiterungsstrategie lediglich neue Funktionalitäten als Microservices implementiert werden, geht diese Vorgehensweise mit den geringsten Entwicklungskosten einher. Die bestehende Anwendung wird nicht angepasst. Somit werden auch keine bestehenden Funktionen extrahiert und innerhalb eines Microservices neu implementiert.

Im Rahmen einer iterativen Umsetzung, etwa dem Strangler-Muster, werden auch bestehende Funktionalitäten des Monolithen extrahiert und als Microservice neu implementiert. Zusätzliche Entwicklungsaufwände fallen außerdem für die Erstellung einer Strangler-Fassade an, welche für die Weiterleitung von Anfragen an das Legacy-System oder die Microservice-Architektur verantwortlich ist. Darüber hinaus existieren typischerweise Integrationen zwischen altem und neuem System, welche etwa für die Synchronisation der Daten zuständig sind. Vorteil ist allerdings, dass neue Funktionalitäten, welche eine eigenständige Subdomäne betreffen, ähnlich zu der Erweiterungsstrategie direkt als Microservice implementiert werden können.

Eine komplette Neuentwicklung der Architektur erfordert ebenfalls die Reimplementierung aller Funktionalitäten des Monolithen. Im Vergleich zur Erweiterungsstrategie erfordert eine Neuentwicklung somit ebenfalls erhöhte Entwicklungsaufwände.

Gegenüber dem Strangler-Muster hat die Neuentwicklung in Bezug auf die Entwicklungskosten den Vorteil, dass neue und alte Architektur vollständig voneinander getrennt sind. Es wird

entsprechend kein Code für die Integration von Legacy-System und Microservice-Architektur benötigt.

Nachteil einer Neuentwicklung ist, dass Funktionalitäten gegebenenfalls mehrfach entwickelt werden müssen. Eine Migration ist typischerweise ein Prozess, welcher mehrere Jahre andauert [57]. Werden während der Neuentwicklung der Anwendung neue Funktionalitäten seitens der Kunden gefordert, welche zeitnah zur Verfügung gestellt werden müssen, so muss die Implementierung sowohl im Legacy-System als auch in der neuen Microservice-Architektur erfolgen [99, S. 430]. Diese Aufwände sorgen entsprechend für erhöhte Entwicklungskosten.

Wartungskosten

Neben den Entwicklungskosten sind auch die Kosten für die dauerhafte Wartung der modernisierten Anwendung relevant. Diese verursachen in typischen Projekten einen bedeutenden Teil der Gesamtkosten. Abbildung 6.4 zeigt, welche Kosten in einem typischen Projekt für die

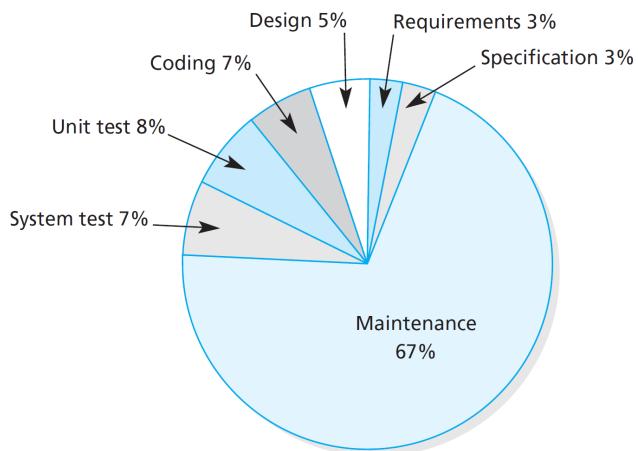


Abbildung 6.4: Relative Kosten verschiedener Softwareentwicklungsphasen [133, S. 12]

jeweiligen Phasen aufgewendet werden. Der überwiegende Teil der Kosten wird dabei für die Wartung der Software benötigt. Bell [133, S. 12] beschreibt etwa, dass es in der Praxis nicht unüblich ist, drei Viertel der Entwicklungszeit für die Wartung der Software zu investieren. Durch die potenziell großen Zeitaufwände hat die Wartbarkeit damit einen hohen Einfluss auf die Kosten eines Softwareprojektes.

In Bezug auf die Wartungskosten selbst sorgt das Hinzufügen und Anpassen von Funktionalitäten laut Sommerville [134] für die größten Aufwände (vgl. Abbildung 6.5). Die genauen Kosten und Aufwände unterscheiden sich je nach Projekt und Organisation. Die Daten von Bell und Sommerville geben allerdings einen groben Anhaltspunkt, mit welchen Kosten und Aufwänden innerhalb eines typischen Softwareprojektes gerechnet werden muss. Auf dieser Basis können auch Rückschlüsse auf die Kosten in Abhängigkeit der gewählten Prozessstrategie getroffen werden.

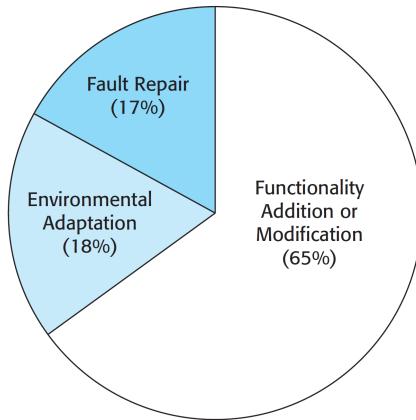


Abbildung 6.5: Kostenanteil einzelner Wartungsaktivitäten [134]

Die langfristigen Wartungsaufwände hängen insbesondere von der gewählten Zielarchitektur ab. Die Microservice-Architektur verspricht durch die lose Kopplung mehrerer Services eine geringe Komplexität innerhalb eines jeden Services. Darüber hinaus können parallel Änderungen durch verschiedene Teams an mehreren Services vorgenommen werden. Diese Aspekte erhöhen die Wartbarkeit und senken langfristig die Aufwände und Kosten. Die verbesserte Wartbarkeit ist in der Praxis eine Hauptmotivation für die Einführung von Microservice-Architekturen. Taibi et al. [135] befragen in ihrer Studie 21 Teilnehmer nach den Gründen für die Einführung einer Microservice-Architektur. Die überwiegende Mehrheit nennt dabei die verbesserte Wartbarkeit als Treiber der Migration. Diese Motivation wird auch durch die Studien von Fritzsch et al. [57] und Auer et al. [4] bestätigt.

Eine vollständige Microservice-Architektur, welche eine gute Wartbarkeit gewährleisten soll, wird durch die Anwendung des Strangler-Musters oder eine vollständige Neuentwicklung erreicht. In beiden Fällen ist das Ziel, die bestehende Anwendung vollständig abzulösen und durch modulare, lose gekoppelte Microservices zu ersetzen.

Im Falle der Anwendung der Erweiterungsstrategie bleibt der Monolith bestehen und wird lediglich um neue Microservices ergänzt.

Eine Übersicht der beschriebenen Bewertungskriterien ist in Abbildung 6.6 dargestellt. Ähnlich zu den Dekompositionsstrategien lässt sich auch in diesem Fall keine pauschal beste Vorgehensweise bestimmen. Je nach Anwendungsfall und Rahmenbedingungen eignen sich verschiedene Vorgehensweisen. Die Wahl der Prozessstrategie hat auch Einfluss auf die Dekompositionsstrategie. Bei einer schrittweisen Migration der bestehenden Anwendung liegt es nahe, eine Dekompositionsstrategie zu wählen, welche Microservices auf Grundlage der bestehenden Codebasis ermittelt. Für eine Neuentwicklung kann es je nach Qualität des bestehenden Systems hingegen sinnvoll sein, ausschließlich Methoden des DDD zu nutzen, um die neue Architektur zu modellieren und geeignete Microservices zu definieren.

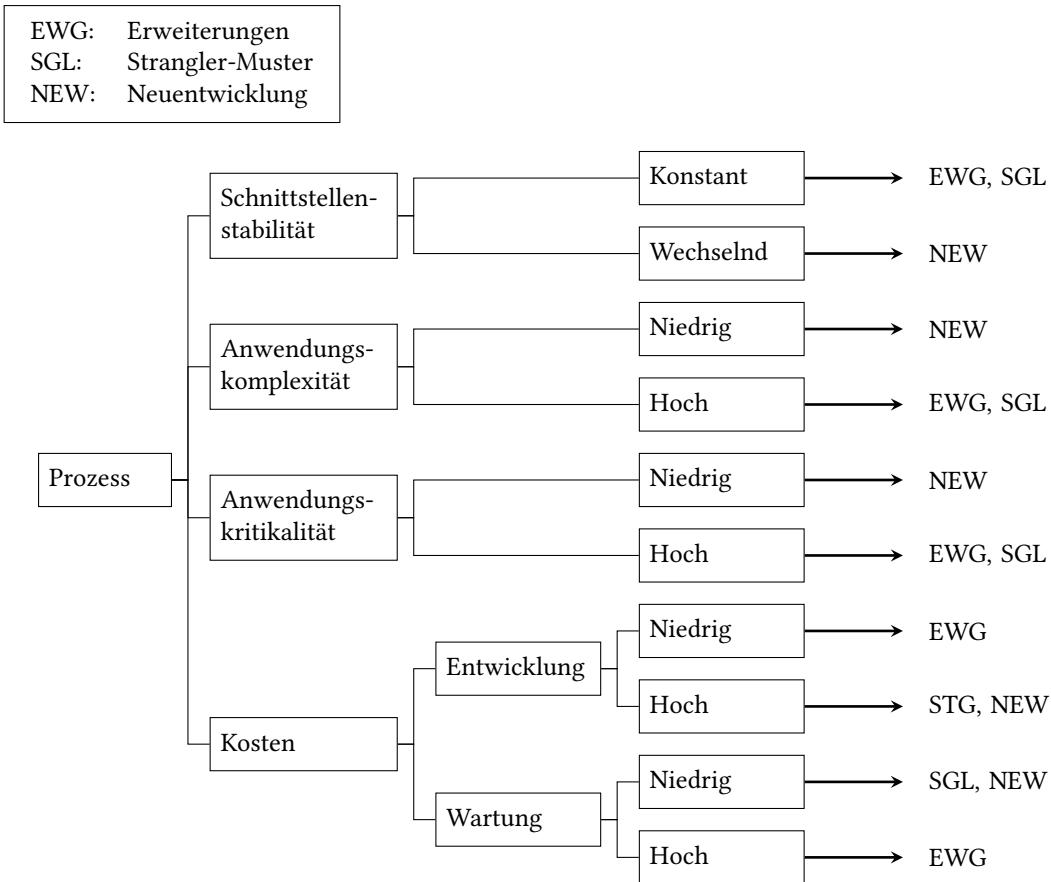


Abbildung 6.6: Bewertungskriterien für die Auswahl einer Prozessstrategie

6.3 Bewertung der Variabilitätsstrategien

6.3.1 Abgrenzung

Im Folgenden wird eine Bewertung der verschiedenen Variabilitätsstrategien aus Kapitel 5 vorgenommen. Für die Betrachtung verschiedener Bewertungskriterien wird zunächst eine Einschränkung vorgenommen: die Software-Produktlinien (vgl. Abschnitt 5.7) beschreiben einen Entwicklungsansatz, welcher die Wiederverwendbarkeit von Komponenten in einer Produktfamilie erleichtern soll. Mehrere Varianten eines Produktes sollen somit möglichst dynamisch ohne manuellen Aufwand erstellt werden können. Dies erlaubt es etwa, individuelle Softwarereprodukte für verschiedene Kunden ohne großen Mehraufwand zu entwickeln. Der Ansatz der Produktlinienentwicklung beschreibt allerdings ein abstraktes Konzept. Die Gemeinsamkeiten und Variabilitäten lassen sich auf verschiedene Weisen umsetzen. Konkrete Ansätze sind etwa Feature Flags (vgl. Abschnitt 5.1) oder konfigurierbare Komponenten (vgl. Abschnitt 5.3). Diese konkreten Möglichkeiten der Umsetzung werden bereits unabhängig von Software-Produktlinien betrachtet.

Darüber hinaus umfassen Software-Produktlinien weitere Aspekte, die für die Motivation

dieser Arbeit nicht relevant sind. Dazu gehören insbesondere die Anforderungsanalyse und die Ableitung einer Produktvariante für einen spezifischen Kunden. Auf diese Weise können etwa mithilfe von Build-Tools oder dem Präprozessor unterschiedliche Produktvarianten für verschiedene Kunden generiert werden. Im Rahmen dieser Arbeit wird allerdings eine geschlossene Umgebung vorausgesetzt. Die Anzahl der verschiedenen denkbaren Produktvarianten ist bereits bekannt und auf eine kleine Menge beschränkt. Außerdem sind die Anforderungen und benötigten Features für jede Variante bekannt. Eine dynamische Zusammensetzung verschiedener Features ist somit nicht von Relevanz.

In diesem Zusammenhang wird auch das von Naily et al. [72] vorgestellte ABS-Microservice-Framework, welches Microservices und Produktlinien verbindet, von der Betrachtung ausgenommen. Das Framework wird unter anderem auch in der Arbeit von Setyautami et al. [136] genutzt, um Produkte aus mehreren Microservices zu generieren. Die Generierung von Microservices anhand von Features berücksichtigt allerdings nicht Eigenschaften wie lose Kopplung und hohe Kohärenz, die essenziell für eine gute Aufteilung mehrerer Microservices sind. Zwei Features, die voneinander abhängen und oft miteinander interagieren, führen somit etwa zu zwei stark gekoppelten Microservices.

Darüber hinaus ist das ABS-Microservice-Framework lediglich als Prototyp veröffentlicht. Letztmals im Januar 2017 aktualisiert, findet keine aktive Weiterentwicklung statt. So existieren Einschränkungen in Bezug auf Sicherheitsfunktionalitäten, HTTP-Anfragemethoden und Datenformate, was eine produktive Nutzung nicht möglich macht.

Die verbleibenden Techniken, um Gemeinsamkeiten und Variabilitäten in einer Microservice-Architektur zu verwalten, werden im Folgenden anhand von verschiedenen Bewertungskriterien evaluiert und je nach Situation auf ihre Anwendbarkeit analysiert. Die Bewertungskriterien wurden ermittelt, indem untersucht wurde, in Bezug auf welche Eigenschaften die Variabilitätsstrategien Unterschiede aufweisen. Ein gemeinsam genutzter Service erfordert etwa die Kommunikation über ein Netzwerk, wodurch zusätzliche Latenzen entstehen. Bestehen hohe Anforderungen an die Performance, ist dies entsprechend ein Kriterium, was berücksichtigt werden muss. Insgesamt konnten fünf Aspekte identifiziert werden, in denen die Variabilitätsstrategien relevante Unterschiede zueinander aufweisen: der Grad der Variabilität, die Wartbarkeit, die Performance, die Verfügbarkeit und die Kosten.

6.3.2 Grad der Variabilität

Als erstes Bewertungskriterium kann der Grad der Variabilität herangezogen werden. Je nach Grad der Gemeinsamkeiten und Unterschiede wird bereits die Anwendbarkeit einiger Ansätze eingeschränkt.

Im Falle von Feature Flags (vgl. Abschnitt 5.1) werden alle Varianten eines Produktes in der gleichen Codebasis verwaltet. Ein Microservice verarbeitet also Anfragen für mehrere Produkte und führt je nach Auswahl unterschiedliche Geschäftsprozesse aus. Statt mit klassischen Feature Flags kann eine solche Umsetzung auch über Designmuster erfolgen. Die grundsätzliche Idee, unterschiedliche Varianten in einem Prozess bereitzustellen, bleibt allerdings die gleiche. Je nach Produktvariante führt der Prozess lediglich unterschiedliche Kontrollstrukturen aus.

Je stärker sich mehrere Produktvarianten voneinander unterscheiden, desto schwieriger wird typischerweise die Integration verschiedenster Features innerhalb einer Codebasis. Es besteht somit die Gefahr, dass die Komplexität steigt und der Microservice sich zu einer Art Monolith entwickelt, welcher nur schwer erweiterbar ist. Dies berichten auch Teilnehmer der Studie von Wang et al. [22]. Einige der Teilnehmer haben begonnen unterschiedliche Varianten mithilfe von Feature Flags zu implementieren. Im Laufe der Zeit wurde der Code oftmals zu komplex, was dazu führte, dass nach alternativen Lösungen gesucht wurde.

Enthalten die Produktvarianten einen geringen Grad der Variabilität und unterscheiden sich lediglich in einigen wenigen Details, können Feature Flags ein einfaches Mittel sein, um unterschiedliche Produkte in einem Prozess anzubieten.

Einen ähnlich hohen Grad an Gemeinsamkeiten erfordert die Anwendung der konfigurierbaren Instanzen (vgl. Abschnitt 5.3). In diesem Fall werden etwa Attribute innerhalb der Codebasis mit extern bereitgestellten Werten initialisiert. Verschiedene Instanzen einer Software können dann mit verschiedenen Konfigurationen bereitgestellt werden. Dieses Vorgehen bietet sich vor allem dann an, wenn sich mehrere Varianten eines Produktes lediglich anhand fester Attribute unterscheiden. Etwa können sich der Produktname oder Minimal- bzw. Maximalwerte verschiedener Plausibilitätsprüfungen je nach Kunde unterscheiden. Die Bereitstellung mehrerer Produkte erfordert in diesem Fall keinen zusätzlichen Programmieraufwand, sondern lediglich die mehrfache Bereitstellung eines Services.

Weisen mehrere Produktvarianten eine höhere Variabilität auf und unterscheiden sich in mehr als nur festen Attributen, so ist die Nutzung der extern konfigurierbaren Instanzen vergleichbar zu den zuvor genannten Feature Flags (vgl. Abbildung 6.7). In diesen Fällen muss der Kontrollfluss der Software in Abhängigkeit von externen Attributen gesteuert werden. Dies stellt eine Form der Feature Flags dar, was auf Dauer zu einer steigenden Komplexität der Codebasis führt.

Geteilte Bibliotheken (vgl. Abschnitt 5.4), geteilte Services (vgl. Abschnitt 5.5) und Sidecars (vgl. Abschnitt 5.6) weisen in der Verwaltung der Gemeinsamkeiten und Unterschiede einige ähnliche Eigenschaften auf. Insbesondere geteilte Services und das Sidecar-Muster unterscheiden sich lediglich in der Bereitstellung. In allen Fällen können Gemeinsamkeiten, welche von allen Produktvarianten geteilt werden, in einer Codebasis implementiert werden. Die Variabilitäten, die die Varianten unterscheiden, werden jeweils in einer eigenen Codebasis

implementiert. Während in diesen Fällen zwar mehrere Projekte existieren, werden Produktvarianten voneinander getrennt. Die Komplexität jeder einzelnen Codebasis kann somit reduziert werden, da innerhalb dieser nicht zwischen verschiedenen Varianten und Vorgehensweisen unterschieden werden muss.

Mit steigendem Grad der Variabilität kann es sich daher anbieten, jede Produktvariante separat zu entwickeln und vorhandene Gemeinsamkeiten in einer geteilten Bibliothek oder einem geteilten Service zu kapseln.

Eine weitere Vorgehensweise, welche besonders im Rahmen von Microservices oft als Alternative zu geteilten Bibliotheken propagiert wird, sind Code-Duplikationen (vgl. Abschnitt 5.2). Bei der Ableitung neuer Produktvarianten ist es in der Praxis weit verbreitet, eine bestehende Codebasis zu duplizieren und an die Gegebenheiten der neuen Produktvariante anzupassen (engl. *Clone-and-Own*) [137], [138], [139]. In diesen Fällen werden Gemeinsamkeiten nicht zentral verwaltet. Jede Variante eines Produktes wird separat in einer Codebasis entwickelt. Dies betrifft sowohl alle Variabilitäten als auch alle Gemeinsamkeiten. Weisen unterschiedliche Produktvarianten große Unterschiede auf und können sich diese unabhängig voneinander entwickeln, so können Code-Duplikationen ein valides Mittel sein, um mehrere Produktvarianten zu verwalten.

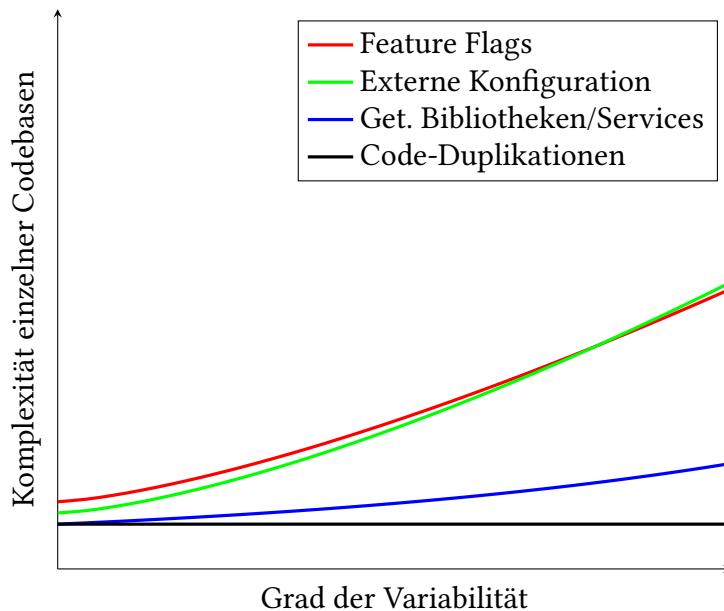


Abbildung 6.7: Zusammenhang zwischen dem Grad der Variabilität und der Komplexität einzelner Codebasen unter Anwendung verschiedener Variabilitätsstrategien

6.3.3 Wartbarkeit

Eine Hauptmotivation für die Einführung von Microservices in der Praxis ist die verbesserte Wartbarkeit der Software [4], [57], [135]. Wie gut eine Software wartbar ist, hängt auch von

der Komplexität der Codebasis ab. Der in Abschnitt 6.3 beschriebene Grad der Variabilität hat somit direkte Auswirkungen auf die Wartbarkeit und muss berücksichtigt werden. Ziel ist es, dass die Einführung von Variabilität die Wartbarkeit nicht maßgeblich negativ beeinträchtigt. Für mehrere Produktvarianten, die sich stark voneinander unterscheiden, bietet sich eine Umsetzung der Variabilität etwa mittels Features Flags daher nicht an.

Auch darüber hinaus unterscheiden sich die unterschiedlichen Umsetzungsmöglichkeiten in Bezug auf ihre Wartbarkeit. Im Falle von Feature Flags und der extern konfigurierten Instanzen, werden alle Produktvarianten innerhalb der gleichen Codebasis verwaltet. Ähneln sich die Produktvarianten stark und besteht ein hohes Vertrauen, dass sich neue Funktionen in der Regel für alle Varianten gleich verhalten, so kann dies ein Anwendungsfall für die Verwendung dieser Vorgehensmuster sein. Die Wartungsaufwände werden in diesem Fall gering gehalten, da Änderungen lediglich einmalig vorgenommen werden müssen. In Kontrast dazu steht der Ansatz der Code-Duplikationen, welcher die Anpassung mehrerer Codebasen erfordert.

Ein Ansatz, welcher oft für seine schlechte Wartbarkeit kritisiert wird, sind die geteilten Bibliotheken (vgl. Abschnitt 5.4). Insbesondere im Kontext von Microservices, welche unabhängig voneinander sein sollen, können geteilte Bibliotheken für eine starke Kopplung sorgen. Anpassungen an der Bibliothek sorgen somit dafür, dass alle Services angepasst werden müssen [21, S. 155], [99, S. 43]. Aufgrund der Kopplung von Microservices und der damit einhergehenden erschweren Wartbarkeit, sollen geteilte Bibliotheken deshalb mit Bedacht eingesetzt werden. Insbesondere für die Auslagerung von Geschäftslogik sind Bibliotheken im Kontext von Microservices nur selten ein geeignetes Mittel [112]. Bibliotheken, welche Gemeinsamkeiten kapseln, sollten nur dann in Betracht gezogen werden, wenn wahrscheinlich ist, dass sich die Funktionalität nicht ändern wird oder akzeptiert werden kann, dass unterschiedliche Services mit verschiedenen Versionen der Bibliothek bereitgestellt werden. Dies kann etwa für statische Konstanten der Fall sein [33, S. 183].

Als Alternative zu geteilten Bibliotheken werden häufig geteilte Services genannt [22], [33, S. 184 f.], [70], [99, S. 43], [112]. Diese kapseln gemeinsame Funktionalitäten und werden von mehreren anderen Services über eine öffentliche Schnittstelle angesprochen. Die Korrektur von Fehlern führt somit dazu, dass lediglich der geteilte Service betroffen ist. Es muss somit keine koordinierte Bereitstellung mehrerer Services vorgenommen werden. Stattdessen profitieren alle Microservices, die den geteilten Service nutzen, implizit von der Fehlerkorrektur.

Auch die Herausforderung von inkompatiblen transitiven Abhängigkeiten wird auf diese Weise umgangen, da der geteilte Service als eigenständige Instanz bereitgestellt wird und nicht mit den Abhängigkeiten anderer Services interferiert. Durch die verbesserte lose Kopplung im Vergleich zu den geteilten Bibliotheken, kann somit die Wartbarkeit der Gesamtarchitektur erhöht werden.

Die verbleibende Option, mehrere Varianten eines Produktes bereitzustellen, sind Code-Duplikationen. Dabei wird für jede Variante ein neuer Microservice erstellt. In der klassischen Softwareentwicklung gelten diese Duplikationen als schlechter Stil, da sie gegen das DRY-Prinzip [62] verstößen. In einer Microservice-Architektur gelten Duplikationen jedoch weithin als valides Mittel, um eine maximale Unabhängigkeit zu erreichen [63], [64], [65]. Die vollständige Trennung aller Gemeinsamkeiten und Variabilitäten führt dazu, dass Änderungen, welche alle Varianten betreffen, mehrfach vorgenommen werden müssen. Insbesondere bei einer hohen Anzahl an Produktvarianten ist somit ein erhöhter Wartungsaufwand notwendig, um Gemeinsamkeiten in allen Codebasen einzupflegen.

Bei Änderungen an Variabilitäten wirkt sich diese Form der Architektur positiv auf die Wartbarkeit aus. Jede Variante kann unabhängig angepasst werden, ohne dass potenzielle Auswirkungen auf andere Produktvarianten berücksichtigt werden müssen.

6.3.4 Performance

Die Performance hängt vor allem davon ab, wie viele Microservices benötigt werden, um eine Funktionalität einer Produktvariante zur Verfügung zu stellen.

Im Falle von Feature Flags, konfigurierbaren Instanzen, geteilten Bibliotheken und Code-Duplikationen wird jeweils ein Microservice benötigt, um die Geschäftsfähigkeit einer Produktvariante bereitzustellen.

Wird hingegen ein weiterer Microservice benötigt, wie es im Falle von geteilten Services und dem Sidecar-Muster der Fall ist, sind zwei Microservices für die Bereitstellung einer Geschäftsfähigkeit beteiligt.

In ersterem Fall findet eine Kommunikation verschiedener Komponenten lediglich innerhalb eines Prozesses statt. Aufrufe innerhalb eines Prozesses können darüber hinaus durch den Compiler - etwa durch Inline-Ersetzung - optimiert werden [21, S. 90]. Sind mehrere Microservices für die Bearbeitung einer Anfrage involviert, so wird eine Kommunikation über das Netzwerk erforderlich. Dies führt zu einer deutlich erhöhten Latenz [2], da über potenziell verschiedene Regionen kommuniziert wird und Daten serialisiert werden müssen [21, S. 90]. Je nach Anforderungen an die Verarbeitungsgeschwindigkeit ist eine Aufteilung in mehrere Microservices deshalb nicht immer sinnvoll [140].

Muss die Software kritische Anforderungen in Bezug auf Performance gewährleisten, spricht dies für eine Umsetzung, welche nur einen oder wenige Microservices für die Bereitstellung einer Produktfunktionalität benötigt. Bestehen keine Anforderungen in dieser Hinsicht, können Netzwerklatenzen für die Inter-Service Kommunikation toleriert werden.

Ein Kompromiss kann die Anwendung des Sidecar-Musters sein. Ein Sidecar wird neben dem Hauptprozess innerhalb des gleichen Pods betrieben. Geteilter Service und Hauptanwendung

teilen sich somit Ressourcen und werden auf dem gleichen Host ausgeführt. Während weiterhin eine Kommunikation über das Netzwerk notwendig ist, werden Antwortzeiten durch die Ausführung auf dem gleichen Host minimiert [22].

6.3.5 Verfügbarkeit

Die Wahl der Variabilitätsstrategie hat ebenfalls Einfluss auf die Verfügbarkeit der Microservice-Architektur. Als erstes Kriterium für die Bewertung der Verfügbarkeit dient analog zu der Performance (vgl. Unterabschnitt 6.3.4) die Anzahl der Microservices, die benötigt werden, um eine Geschäftsfähigkeit auszuführen.

Werden gemeinsame Funktionalitäten, die von mehreren Microservices benötigt werden, in einen geteilten Service ausgelagert, so stellt dies einen potenziellen Single Point of Failure dar. Eine hohe Last des geteilten Services, welche zu langen Antwortzeiten führt, hat somit Auswirkungen auf alle Microservices. Auch der Ausfall des geteilten Services führt dazu, dass alle abhängigen Microservices ihre Geschäftsfähigkeiten potenziell nicht mehr vollständig ausführen können. Besteht die Gefahr einer hohen Last auf einen einzelnen geteilten Service, sollten entsprechend mögliche Alternativen in Betracht gezogen werden. Die Skalierung des gemeinsamen Services oder die Bereitstellung mehrerer Sidecars kann potenziell auftretenden Performance-Engpässen vorbeugen. Von fachlichen Fehlern innerhalb eines geteilten Services, die zum Absturz desselben führen können, sind allerdings weiterhin alle abhängigen Microservices betroffen.

Handelt es sich bei einer Geschäftsfähigkeit um eine Funktionalität, die hochverfügbar sein muss, sollten Abhängigkeiten von weiteren Services nach Möglichkeit vermieden werden. Ist dies nicht praktikabel, gilt es sicherzustellen, dass die Stabilität der abhängigen Services durch geeignete Skalierungsmaßnahmen gesichert ist.

Für die Verfügbarkeit der gesamten Software ist ebenfalls relevant, wie viele Produktvarianten durch eine Microservice-Instanz zur Verfügung gestellt werden. Im Falle von Feature Flags werden etwa alle Varianten einer Produktfamilie innerhalb eines Microservices implementiert. Kommt es zu einem Fehler innerhalb der Verarbeitungskette einer spezifischen Variante, der zum Ausfall des Services führt, so sind alle Varianten des Produktes betroffen.

Im Falle von geteilten Bibliotheken und Services, sowie Code-Duplikationen werden die Variabilitäten einzelner Produktvarianten innerhalb eigener Codebasen entwickelt. Für jede Produktvariante existiert somit ein eigener Microservice. Fehler oder Ausfälle einer spezifischen Produktvariante führen somit nicht zwangsläufig dazu, dass auch alle anderen Produktvarianten betroffen sind.

6.3.6 Kosten

Ein weiteres Kriterium für die Umsetzung von Variabilitäten sind die Kosten. Ein relevanter Aspekt für die Kosten ist die Wartbarkeit der Architektur. Die Wartungsarbeiten sind ein wesentlicher Kostenfaktor innerhalb eines Softwareprojektes (vgl. Unterabschnitt 6.2.4). Die Auswahl einer Variabilitätsstrategie hat Auswirkungen auf die Wartbarkeit und somit auch auf die langfristigen Wartungskosten. Der Einfluss der Variabilitätsstrategien auf die Wartbarkeit wird gesondert in Unterabschnitt 6.3.3 behandelt und muss bei der Bewertung der Kosten berücksichtigt werden.

Darüber hinaus existieren die Bereitstellungskosten, die mit der Ressourcennutzung der Microservice-Architektur zusammenhängen. Mithilfe von Feature Flags etwa werden verschiedene Produktvarianten innerhalb eines Microservices angeboten. Eine Softwareinstanz kann somit für potenziell mehrere Kunden genutzt werden. Je nach Mandant (engl. *tenant*) und dessen freigeschalteten Features verhält sich die Microservice-Implementierung entsprechend unterschiedlich. Dadurch, dass mehrere Kunden mit einer Softwarebereitstellung interagieren (vgl. Abbildung 6.8), verringern sich die Bereitstellungskosten. Das Deployment wird vereinfacht, Ressourcen werden besser ausgelastet und Wartungsmaßnahmen müssen nur für eine Instanz vorgenommen werden [141]. Die geringeren Gesamtkosten können somit auch für eine bessere Preisbildung gegenüber Konkurrenten sorgen [141].

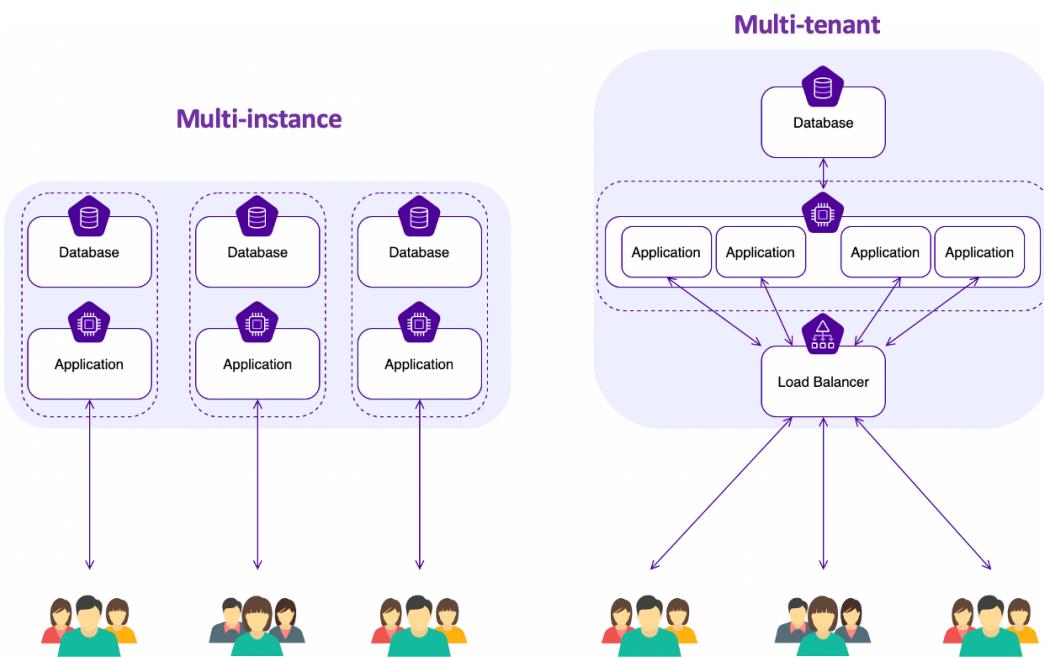


Abbildung 6.8: Single-Tenant und Multi-Tenant Applikationen [142]

Alle weiteren Umsetzungen arbeiten mit einer eigenen Microservice-Instanz für jede Produktvariante. Durch die erhöhte Anzahl von Microservice-Bereitstellungen können - je nach Anzahl der Varianten und Mandanten - die Bereitstellungskosten merklich ansteigen.

Die Verarbeitung von mehreren Mandanten innerhalb einer Instanz bringt jedoch auch einige Herausforderungen in Bezug auf Performance, Skalierbarkeit und Sicherheit mit sich [141]. Aus diesem Grund kann eine separate Bereitstellung insbesondere bei einer geringen Anzahl von Mandanten die bessere Option sein [143].

Eine Übersicht der beschriebenen Bewertungskriterien ist in Abbildung 6.9 dargestellt. Analog

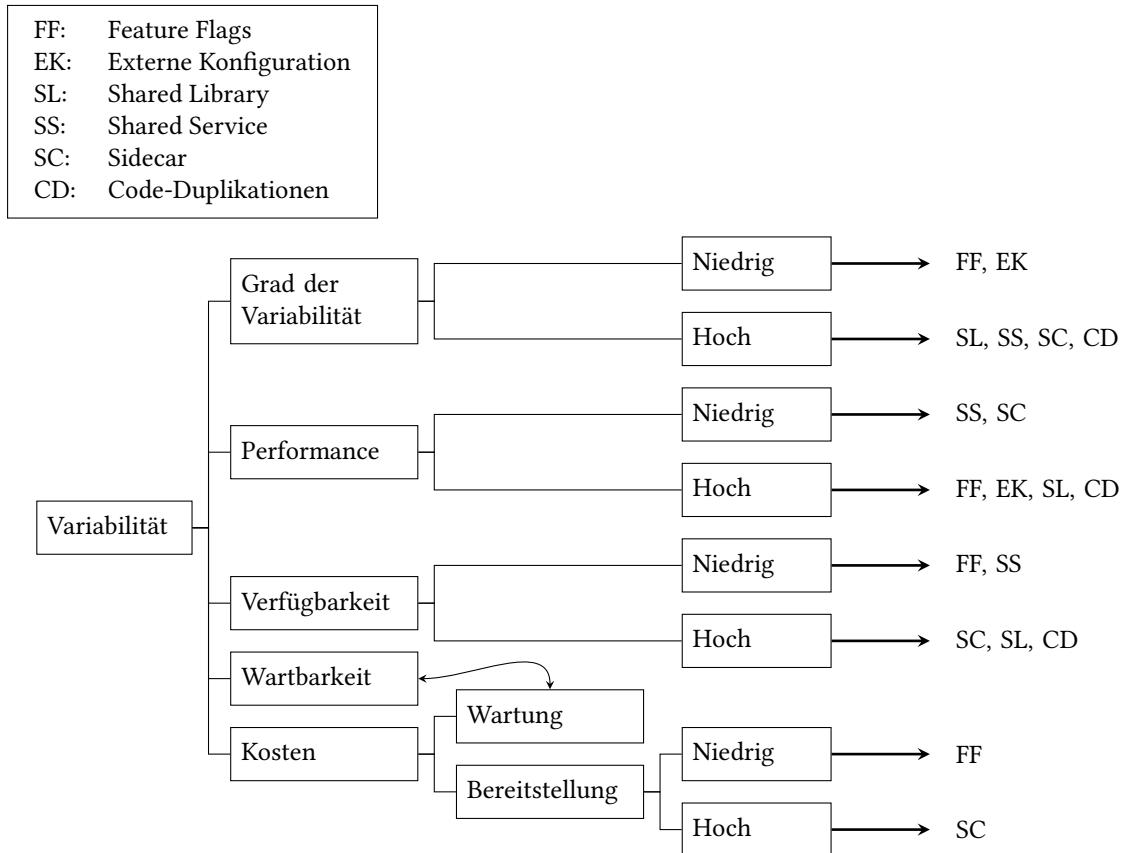


Abbildung 6.9: Bewertungskriterien für die Auswahl einer Variabilitätsstrategie

zu den Dekompositions- und Prozessstrategien lässt sich je nach Anwendungsfall eine andere Umsetzung der Variabilität implementieren. Die Bestimmung einer pauschal besten Strategie ist in Anbetracht der verschiedenen Anwendungsfälle nicht realistisch. Die Kriterien helfen allerdings bei der Bewertung des eigenen Anwendungsfalls und ermöglichen so eine geeignete Auswahl.

Insbesondere bei den Variabilitätsstrategien ist außerdem keine Beschränkung auf genau eine Vorgehensweise vorhanden. Während Code-Duplikationen für einen Microservice geeignet sind, können etwa extern konfigurierbare Instanzen für einen anderen Microservice innerhalb der gleichen Softwarearchitektur gewählt werden.

Darüber hinaus sind auch Kombinationen mehrerer Techniken innerhalb eines Microservices denkbar. Ein Sidecar etwa kann zusätzlich extern konfiguriert werden, sodass verschiedenen Microservices unterschiedlich konfigurierte Sidecars zur Verfügung gestellt werden.

6.4 Zusammenfassung

Das Konzept beschreibt für die Dekomposition, den Prozess und die Variabilität verschiedene Bewertungskriterien, anhand derer sich eine geeignete Auswahl treffen lässt. Dabei werden verschiedene Vor- und Nachteile der jeweiligen Vorgehensweisen beschrieben und Anwendungsfälle betrachtet, in welchen sich die unterschiedlichen Strategien anbieten. Dabei wird auch deutlich, dass nicht in jedem Anwendungsfall alle Kriterien die gleiche Strategie propagieren. In diesen Fällen muss abgewogen werden, welche Eigenschaften am höchsten priorisiert werden. So können etwa Risiken im Rahmen eines Cut-over-Prozesses gegenüber langfristigen Wartungskosten als weniger wichtig eingestuft werden. Eine Abwägung und Bewertung der unterschiedlichen Kriterien obliegt somit den Anwendern im konkreten Anwendungsfall.

Das Konzept erleichtert jedoch die Entscheidung, indem Kriterien ermittelt wurden, auf deren Basis sich eine systematische und begründbare Entscheidung treffen lässt.

Abbildung 6.10 zeigt die Übersicht aller im Konzept entwickelten Bewertungskriterien, welche eine Migration und die Umsetzung von Variabilität innerhalb der Microservice-Architektur unterstützen. Die Kriterien werden im Folgenden im Rahmen einer Fallstudie angewendet, um das entwickelte Konzept zu evaluieren und eine geeignete Migrationsstrategie für das EJP-CC zu entwickeln.

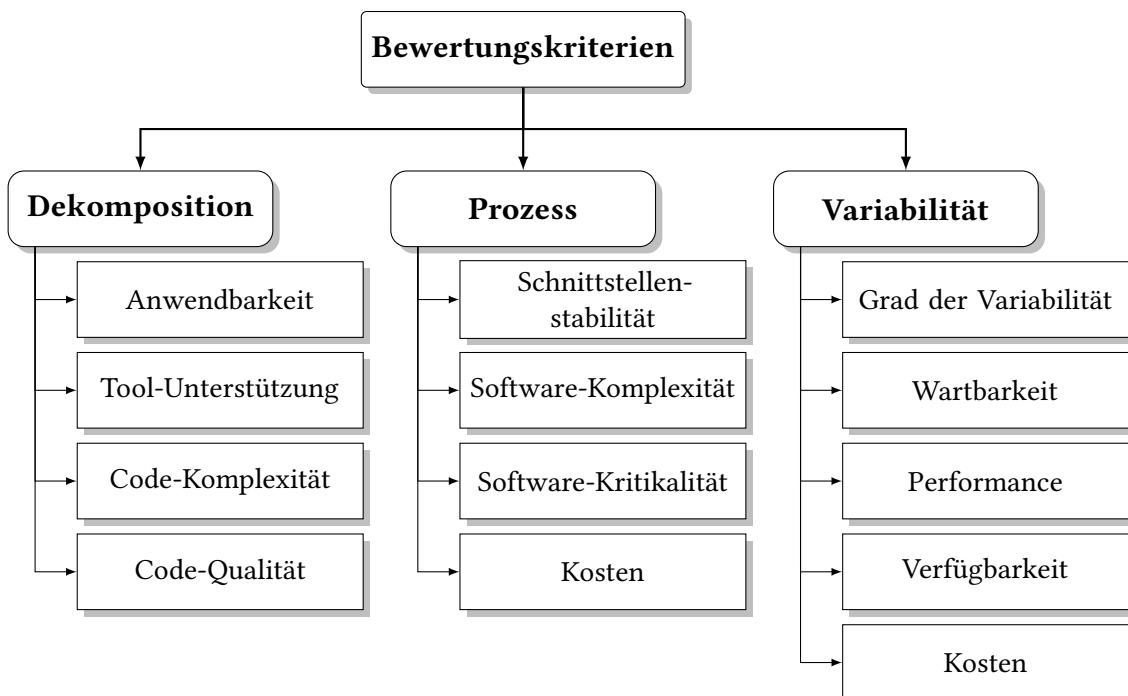


Abbildung 6.10: Bewertungskriterien für die Auswahl von Dekompositions-, Prozess- und Variabilitätsstrategien

7 Fallstudie EJP-CC

7.1 Bestehendes System

Bei dem Spiel Eurojackpot handelt es sich um eine Zahlenlotterie, an der 33 Lotteriegesellschaften aus 18 europäischen Ländern teilnehmen. Für die Teilnahme an der Lotterie können durch einen Kunden mehrere Tipps abgegeben werden. Ein Tipp besteht aus fünf Zahlen der Zahlenreihe 1 bis 50 und zwei Zahlen der Zahlenreihe 1 bis 10 (ab März 2022: 1 bis 12).

Zur Koordination der Ziehungen und Durchführung der zentralen Gewinnauswertung sowie der Abrechnung wurden zwei Kontrollzentren entwickelt. Diese haben die Aufgabe, die Tipps der verschiedenen Lotteriegesellschaften zu sammeln, auszuwerten und entsprechende Gewinne zu berechnen. Das durch Westlotto betriebene EJP-CC wurde initial im Jahr 2011 implementiert und 2012 in Produktion genommen. Abbildung 7.1 zeigt die Architektur des bestehenden Systems.

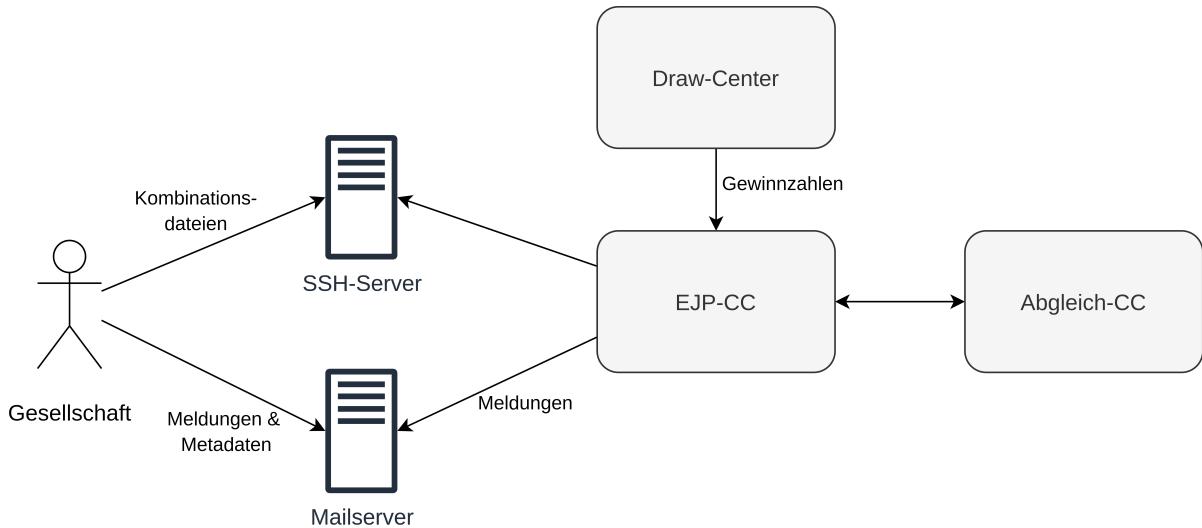


Abbildung 7.1: Architektur des bestehenden EJP-CC

Grundsätzlich existieren zwei Kommunikationswege, über welche die teilnehmenden Gesellschaften und das EJP-CC kommunizieren: einen SSH-Server und einen Mailserver. Die Verarbeitung einer Ziehung läuft dabei vereinfacht dargestellt wie folgt ab:

Zu Beginn der Ziehung bestätigen die Gesellschaften ihre Bereitschaft mit einer E-Mail. Die E-Mails werden automatisch abgerufen und verarbeitet. Eingehende E-Mails werden durch das EJP-CC mit entsprechenden Antwortmails bestätigt.

Anschließend werden sogenannte Kombinationsdateien (auch *Combifiles*) durch die Gesellschaften auf dem SSH-Server bereitgestellt. Eine Kombinationsdatei enthält Informationen darüber, wie viele Tipps für jede gültige Zahlenkombination abgegeben wurden. Neben den Kombinationsdateien werden auch Metadaten durch die Gesellschaften bereitgestellt, welche den Spieleinsatz und eine Prüfsumme enthalten. Das EJP-CC gleicht diese Daten mit den selbst errechneten Daten ab. Bestätigungen und Ablehnungen der bereitgestellten Daten erfolgen über entsprechende E-Mails.

Die Gewinnzahlen der Lotterie werden in Helsinki gezogen und durch das sogenannte Draw-Center an das EJP-CC übertragen. Mithilfe der zuvor empfangenen Kombinationsdateien und den Gewinnzahlen kann die Software für jede Lotteriegesellschaft die Anzahl der Gewinner in jeder Gewinnklasse ermitteln. Auf Basis der Gewinnermittlung wird durch das EJP-CC eine Abrechnung erstellt, die unter anderem beinhaltet, welche Geldbeträge zwischen den Gesellschaften transferiert werden müssen. Diese Ausgleichszahlungen sind notwendig, da eine Poolungsgemeinschaft besteht und die Gewinner somit von allen Gesellschaften bezahlt werden.

Bei der Abwicklung von Lotterien sind potenzielle Manipulationsmöglichkeiten unter allen Umständen zu verhindern. Aus diesem Grund existiert eine unabhängige zweite Implementation des Control-Centers, welche durch die dänische Lotteriegesellschaft entwickelt und betrieben wird. Sämtliche Daten der Gesellschaften werden durch beide Implementationen verarbeitet und anschließend abgeglichen. Für eine erfolgreiche Verarbeitung ist es notwendig, dass beide Control-Center zu identischen Ergebnissen kommen.

Das bestehende EJP-CC ist ein monolithisches Softwaresystem, welches in der Programmiersprache Java entwickelt wurde und als Fat-Client bereitgestellt wird. Im Jahr 2021 wurde durch die Eurojackpot-Kooperation eine Modernisierung des bestehenden Systems beschlossen. Als Rahmenbedingung wurde in diesem Zusammenhang bereits vorgegeben, dass die Kommunikation zwischen Gesellschaften und dem EJP-CC in Zukunft über eine REST-Schnittstelle erfolgen soll.

Die interne Umsetzung und Ausgestaltung dieser Anforderung obliegt den beiden Kontrollzentren. Architektur und verwendete Technologien können somit frei gewählt werden. Um Vorteile wie bessere Wartbarkeit und Skalierbarkeit zu gewährleisten, wurde bei WestLotto eine Migration des bestehenden Systems hin zu Microservices vereinbart.

Im Folgenden werden die entwickelten Bewertungskriterien herangezogen, um eine geeignete Migrationsstrategie zu entwickeln.

7.2 Bewertung der Dekomposition

7.2.1 Anwendbarkeit

Für die Auswahl geeigneter Dekompositionsstrategien wird zunächst die Anwendbarkeit ausgewertet. Die Anwendbarkeit ist eine Vorbedingung, welche zunächst evaluiert werden muss, um die zur Verfügung stehenden Optionen zu bestimmen.

Im vorliegenden Fall liegt eine bestehende Anwendung vor. Der Zugriff auf den Quellcode der Anwendung ist gegeben. Somit lassen sich Methoden, welche die statische Analyse nutzen, anwenden.

Auch die Ausführung der Anwendung ist in der lokalen Entwicklungsumgebung möglich. Somit lassen sich zusätzlich dynamische Laufzeitinformationen sammeln. Methoden der dynamischen Analyse sind somit ebenfalls anwendbar.

Darüber hinaus können modellbasierte Ansätze angewendet werden, die auf Basis der Historie einer Versionsverwaltung oder ER-Modellen Dekompositionsvorschläge generieren. Insbesondere in letzterem Fall ist allerdings gegebenenfalls manueller Aufwand notwendig, um die Beziehungen des bestehenden Systems zu extrahieren.

Das DDD ist eine Herangehensweise für die Modellierung von Software, die nicht zwingend auf eine bestimmte Art von Eingabedaten angewiesen ist, und somit universell im Entwurfsprozess eingesetzt werden kann.

Insgesamt lässt sich festhalten, dass in Bezug auf die Anwendbarkeit nur wenige Einschränkungen gegeben sind, und sich die überwiegende Mehrheit der analysierten Methodiken anwenden lässt, da die benötigten Eingabedaten bereits vorhanden sind, oder erzeugt werden können (vgl. Tabelle 6.1).

7.2.2 Tool-Unterstützung

Als weiteres Kriterium für die Auswahl einer geeigneten Dekompositionsstrategie wird die Tool-Unterstützung herangezogen. Während viele Ansätze keine unterstützende Software anbieten, ist bei anderen Arbeiten eine Tool-Unterstützung vorhanden, sodass keine tiefen Kenntnisse über die Implementationsdetails der Vorgehensweise vorhanden sein müssen.

Im Falle der anstehenden Dekomposition wird eine Unterstützung durch angemessene Tools als essenziell erachtet. Um eine Methodik anzuwenden, welche nicht in Form eines Tools unterstützt wird, muss eine eigene Implementation geschaffen werden. Dies erfordert typischerweise Kenntnisse in Bezug auf die Analyse von Quelltext oder Trace-Logs, den Aufbau

von geeigneten Graphen und die Clusteranalyse. Darüber hinaus erfordern einige Methodiken Kenntnisse der semantischen Themenerkennung oder des maschinellen Lernens. Eine manuelle Implementierung eines abstrakten Ansatzes ist somit aufwendig und sehr zeitintensiv. Eine Tool-Unterstützung wird deshalb explizit erwünscht. Der vielfach in der Literatur beschriebene Mangel an Tools [16], [18], [40] führt in der Praxis auch dazu, dass viele der Dekompositionsansätze in der Industrie nicht verbreitet sind.

Während etwa die Hälfte der identifizierten Arbeiten einen Tool-Support mindestens in prototypischer Form bietet, existieren allerdings Einschränkungen in Bezug auf die unterstützten monolithischen Systeme. Typischerweise wird die Verwendung von bestimmten Programmiersprachen und Frameworks vorausgesetzt, sodass die Liste der anwendbaren Ansätze entsprechend eingeschränkt wird (vgl. Tabelle A.2).

7.2.3 Komplexität

Die bestehende Anwendungsdomäne weist eine tendenziell geringe Komplexität auf. Darüber hinaus sind bei den Entwicklern und Fachverantwortlichen der Anwendung weitreichende Kenntnisse über die Domäne vorhanden und die Geschäftsfähigkeiten, welche durch die Anwendung erfüllt sein müssen, sind klar definiert. Das gute Verständnis über die Anwendungsdomäne resultiert auch daraus, dass der Ablauf einer Ziehung fest definiert ist und stets gleich abläuft.

Die überschaubare Komplexität des Systems spiegelt sich auch in der neu zu entwickelnden REST-Schnittstelle wider. In Vorbereitung auf die anstehende Migration wurde im Vorfeld bereits eine erste Beschreibung der neuen REST-Schnittstelle zwischen Control-Center und den teilnehmenden Gesellschaften als Diskussionsgrundlage vorgeschlagen. Der erste Entwurf sieht 14 Operationen vor, welche den Gesellschaften angeboten werden.

Die eher geringe Komplexität der Anwendungsdomäne spricht somit für eine manuelle Aufteilung der Microservices unter Berücksichtigung der Methodiken des DDD. Alternativ können auch modellbasierte Ansätze gewählt werden, welche auf Basis einer abstrakten Beschreibung des Systems arbeiten.

7.2.4 Qualität

Als letztes Bewertungskriterium für die Auswahl einer geeigneten Dekompositionsstrategie nennt das Konzept die Qualität der bestehenden Codebasis. Insbesondere Methoden der statischen und dynamischen Analyse haben das Ziel, die bestehende Codebasis in Microservices aufzuteilen, indem zusammenhängende Module gefunden werden.

Die Aufteilung des bestehenden Quellcodes ist insbesondere dann sinnvoll, wenn eine hohe Qualität vorliegt und weite Teile des Programms somit übernommen werden können. Weist die bestehende Codebasis bereits eine schlechte Qualität auf, so ist davon auszugehen, dass auch statische und dynamische Analyseansätze nur schlechte Microservice-Kandidaten ermitteln können.

Um die Qualität der bestehenden Codebasis zu beurteilen, wurden die Softwares *Structure101*¹ und *Sonargraph*² eingesetzt, welche jeweils Metriken für die Qualität einer Anwendung berechnen.

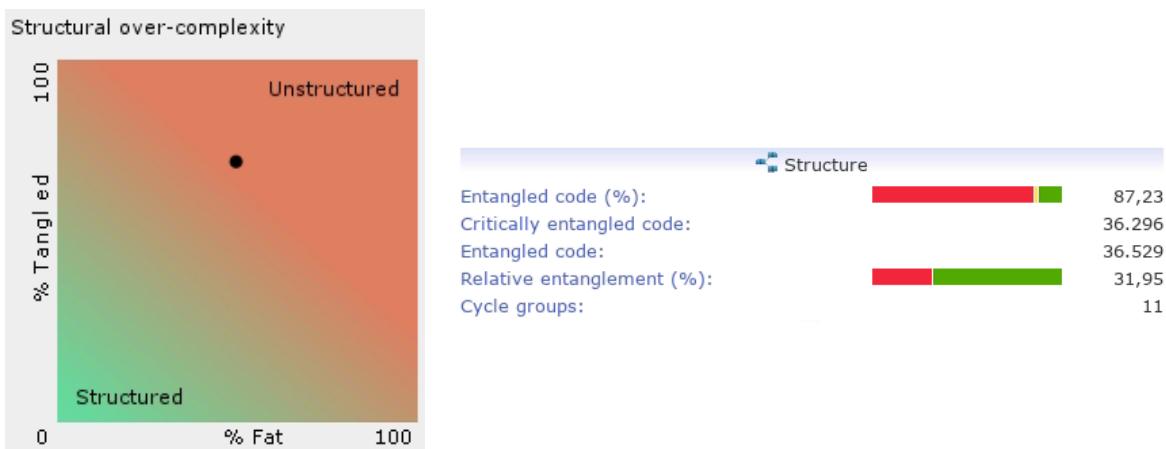


Abbildung 7.2: Bewertung der Codebasis durch Structure101 (links) und Sonargraph (rechts)

Die Komplexität der Codebasis wird durch beide Tools als gering eingestuft. Sonargraph berücksichtigt etwa die maximale Verschachtelungstiefe einzelner Methoden und darüber hinaus die zyklomatische Komplexität (auch *McCabe-Metrik*) [144], um die Komplexität der Codebasis zu bewerten.

Gleichzeitig identifizieren beide Tools die Codebasis als unstrukturiert und verwickelt (engl. *tangled*). Sonargraph ermittelt etwa 11 zyklische Gruppen mit bis zu 45 Komponenten (vgl. Abbildung A.2).

Sowohl Pigazzini et al. [16] als auch Taibi und Systä [17] bestätigen, dass das Auflösen zyklischer Abhängigkeiten notwendig ist, um lose gekoppelte Module zu erkennen und somit qualitativ hochwertige Microservice-Kandidaten identifizieren zu können. Das Auflösen einer derart hohen Anzahl an zyklischen Abhängigkeiten wird im Rahmen dieser Arbeit als zu zeitaufwändig eingeschätzt und stellt somit keine realistische Option dar.

Insgesamt kann die Qualität der bestehenden Systemarchitektur somit als gering eingestuft werden. Eine Identifikation von Microservices auf Grundlage des bestehenden Quellcodes ist auf Basis des Qualitätskriteriums somit keine sinnvolle Option.

¹<https://structure101.com/>

²<https://www.hello2morrow.com/products/sonargraph/architect9>

7.2.5 Zusammenfassung

Die ersten beiden Bewertungskriterien - Anwendbarkeit und Tool-Unterstützung - dienen in erster Linie dazu, den Kreis der verfügbaren Optionen einzuschränken und zu evaluieren. Im vorliegenden Anwendungsfall lassen sich sowohl statische und dynamischen Ansätze als auch modellbasierte Techniken sowie das DDD nutzen. Der geforderte Tool-Support schränkt die Auswahl der Techniken allerdings stark ein.

Die beiden letzten Bewertungskriterien - Komplexität und Qualität des Systems - dienen vor allem der Einschätzung, welche der zur Verfügung stehenden Dekompositionsansätze sinnvoll anwendbar sind. In diesem Fall spricht die Kombination aus geringer Komplexität zusammen mit geringem Wert der Codebasis für einen modellgetriebenen Ansatz gepaart mit Methodiken des DDD.

Ein weiterer Aspekt, der diese Einschätzung unterstützt und der im Konzept bislang nicht berücksichtigt wurde, ist der Grad der Modernisierung. Die Migration des EJP-CC ist in erster Linie durch die Modernisierung des Systems motiviert. Die E-Mail Kommunikation soll durch eine Webschnittstelle ersetzt werden, wodurch Prozesse erheblich vereinfacht werden können. Darüber hinaus soll eine Trennung zwischen Frontend und Backend eingeführt werden, welche ebenfalls über eine Schnittstelle kommunizieren. Der Grad der Modernisierung kann somit als hoch bezeichnet werden, was entsprechend viele Änderungen an der Codebasis nach sich zieht. Auch dieser Aspekt spricht dafür, keine direkte Aufteilung des bestehenden Quellcodes mit Techniken der dynamischen und statischen Analyse vorzunehmen.

7.3 Bewertung des Prozesses

Nach Bewertung der Dekompositionsmöglichkeiten gilt es, den Prozess zu bewerten, nach welchem die Migration vollzogen wird. Um diesbezüglich eine Entscheidung zu treffen, sieht das Konzept ebenfalls vier Bewertungskriterien vor, welche im Folgenden im Rahmen des konkreten Anwendungsfalls betrachtet werden.

7.3.1 Schnittstellenstabilität

Die bestehende Kommunikation zwischen Gesellschaften und dem EJP-CC basiert primär auf E-Mails. Mit Ausnahme des Dateiuploads werden E-Mails bidirektional genutzt, um Daten zu übertragen und zu bestätigen. Im Rahmen der Projektplanung wurde bereits im Vorfeld der Migration eine modernisierte REST-Schnittstelle für die Kommunikation vorgegeben. Diese kann entsprechend als feste Anforderung betrachtet werden.

Der Wechsel von asynchroner E-Mail Kommunikation auf eine weitestgehend synchrone REST-Schnittstelle, stellt eine große Umstellung dar. Die Schnittstellenstabilität kann somit als sehr gering betrachtet werden.

Aus Sicht der Schnittstellenstabilität bietet sich eine iterative Vorgehensweise wie das Strangler-Muster somit nicht an. Dieses benötigt eine Fassade, welche Anfragen an das alte bzw. neue System weiterleitet (vgl. Abbildung 6.2). Eine iterative Umstellung erforderte, dass die Gesellschaften einen Teil ihrer Anfragen über E-Mails abwickeln, während ein anderer Teil bereits über die REST-Schnittstelle abgewickelt wird. Diese uneinheitliche Schnittstelle auf Basis zweier sehr verschiedener Protokolle stellt keine Option dar, da alle Gesellschaften ihre Client-Programme im Rahmen der Migration fortlaufend und koordiniert anpassen müssten.

Auch eine einheitliche Schnittstelle nach außen mit entsprechender Protokollumwandlung innerhalb der Fassade ist aufgrund der verwendeten Protokolle als kritisch zu bewerten, da diese große Unterschiede aufweisen.

Aufgrund der sehr unterschiedlichen Schnittstellen von altem und neuem System ist aus Sicht dieses Bewertungskriteriums eine Neuentwicklung mit einmaliger Umstellung der Schnittstelle die empfohlene Option. Eine iterative Umstellung erweist sich aufgrund der erhöhten Aufwände für die Integration der verschiedenen Protokolle als schwierig (vgl. Unterabschnitt 6.2.1).

7.3.2 Komplexität

Bei der Bewertung der Komplexität kann auf die Ergebnisse aus Unterabschnitt 7.2.3 zurückgegriffen werden, welche bereits im Rahmen der Dekomposition erarbeitet wurden.

Insbesondere Systeme, welche eine hohe Komplexität aufweisen, sollten in einem iterativen Prozess migriert werden. Die Komplexität lässt sich somit schrittweise verlagern und auftretende Probleme können auf einen Teilbereich des neuen Systems eingegrenzt werden.

Für das EJP-CC wurde ermittelt, dass eine eher geringe Komplexität des Systems vorliegt. Während eine iterative Umstellung des Systems grundsätzlich anwendbar ist, ist in Bezug auf die Komplexität auch eine Neuentwicklung denkbar. In letzterem Fall werden keine zusätzlichen Aufwände für die Integration von altem und neuen System notwendig.

7.3.3 Kritikalität

Die Kritikalität einer Anwendung hat Auswirkungen auf potenziell auftretende Fehler während einer Migration. Je kritischer ein System ist, desto schwerwiegender ist typischerweise ein auftretender Fehler zu bewerten. Darüber hinaus wurde im Konzept erarbeitet, dass eine

komplette Neuentwicklung mit anschließendem Cut-over-Prozess ein höheres Risiko für neu auftretende Fehler mit sich bringt. Insbesondere für hochkritische Anwendungen bietet sich daher eine iterative Umstellung des Systems an, um das Risiko von Fehlern zu minimieren.

Im vorliegenden Anwendungsfall kann das EJP-CC als hochkritische Anwendung eingestuft werden. Dies drückt sich auch dadurch aus, dass Änderungen der Software bereits heute weitreichende Tests erforderlich machen. Dazu gehören neben internen Tests unter anderem auch Integrations- sowie Zertifizierungstests mit den Gesellschaften. Es unterliegt darüber hinaus strengen Auflagen und ist verantwortlich für die Verwaltung und Abrechnung von Beträgen im dreistelligen Millionenbereich. Potenzielle Fehler können somit weitreichende Folgen nach sich ziehen und sind in jedem Fall zu vermeiden.

Die hohe Kritikalität der Anwendung spricht somit für eine iterative Umstellung der Anwendung, etwa nach dem Strangler-Muster.

7.3.4 Kosten

Die Kosten dienen als letztes Bewertungskriterium für die Auswahl einer geeigneten Prozessstrategie. Diese lassen sich in einmalige Entwicklungskosten und langfristige Wartungskosten unterteilen.

Die Entwicklungskosten richten sich vor allem nach der langfristigen Zielarchitektur. Wird der bestehende Monolith beibehalten und lediglich neue Funktionalitäten als Microservices implementiert, werden entsprechend nur geringe Kosten für die Entwicklung erforderlich. Im Rahmen der EJP-CC Migration spielen die Entwicklungskosten eine untergeordnete Rolle. Dies begründet sich daraus, dass es sich um eine Eigenentwicklung handelt, welche lediglich intern genutzt wird. Eine Konkurrenzsituation mit anderen Unternehmen, die zur Minimierung von Verkaufspreisen und somit Kostendruck führt, ist nicht gegeben. Stattdessen wurde die Modernisierung des Systems bereits in Auftrag gegeben und die Kosten für die Entwicklung werden durch alle Gesellschaften getragen.

Von größerer Bedeutung sind im Falle des vorliegenden Projekts die Wartungskosten, welche nach Möglichkeit minimiert werden sollen. Hierfür bietet es sich an, die gesamte Anwendung auf eine neue Microservice-Architektur umzustellen, um die potenziell hohen Wartungsaufwände für monolithische Anwendungen zu vermeiden. Dies spricht gegen eine Erweiterungsstrategie, welche sich aufgrund der weitreichenden Änderungen ohnehin nur schwer umsetzen ließe. Unter Berücksichtigung der langfristigen Wartungskosten ist deshalb eine vollständige Umstellung des Systems unter Anwendung eines iterativen Vorgehens oder einer Neuentwicklung zielführend.

7.3.5 Zusammenfassung

In den vorangegangenen Unterkapiteln wurden die vier entwickelten Bewertungskriterien, welche den Migrationsprozess betreffen, im Rahmen der EJP-CC Migration ausgewertet. Dabei konnte festgestellt werden, dass die sich stark ändernde Schnittstelle eine Neuentwicklung nahelegt. Die geringe Komplexität und die langfristig niedrigen Wartungskosten sprechen ebenfalls für dieses Vorgehen.

Lediglich die hohe Kritikalität der Anwendung ist ein Indiz für eine schrittweise Migration der Software. In diesen Fällen müssen die einzelnen Bewertungskriterien miteinander abgewogen werden.

Die Abwägung zwischen Neuentwicklung und iterativer Migration wird im vorliegenden Anwendungsfall zugunsten der Neuentwicklung entschieden. Als ausschlaggebender Faktor kann hier die hohe Anzahl an externen Partnern genannt werden. Eine sich iterativ wandelnde Schnittstelle sorgt sowohl bei teilnehmenden Gesellschaften als auch bei der zweiten Kontrollinstanz des EJP-CC für anhaltende Änderungen, welche koordiniert werden müssen. Darüber hinaus erfordert jede Änderung erneute Integrations- und Zertifizierungstests mit allen Gesellschaften, was zu erheblichen Aufwänden führt. Aus diesem Grund wird eine Neuentwicklung mit Cut-over-Prozess bevorzugt, welcher lediglich einen einmaligen Umstellungsaufwand verursacht, allerdings mit einem erhöhten Risiko für Fehler einhergeht.

7.4 Analyse und Dekomposition

Im folgenden Unterkapitel wird die Analyse und Dekomposition des EJP-CC in eine Microservice-Architektur beschrieben. Die Auswertung der Bewertungskriterien hat gezeigt, dass eine Aufteilung der bestehenden Codebasis nicht zielführend ist. Die Anwendung des Konzeptes besagt, dass modellbasierte Ansätze, sowie das DDD im vorliegenden Anwendungsfall besser angewendet werden können. Da die Verfügbarkeit der anwendbaren Ansätze durch den geforderten Tool-Support stark eingeschränkt wird, wird das Tool Service Cutter eingesetzt, um potenzielle Microservice-Kandidaten zu ermitteln. Da das Tool mit abstrakten Eingabedaten arbeitet, lässt es sich immer anwenden [40]. Darüber hinaus gilt die Anwendung als State-of-the-Art [59], [91]. Die Ergebnisse des Service Cutters dienen neben den Ergebnissen des DDD als Diskussionsgrundlage für eine geeignete Dekomposition der Anwendung in Microservices.

In Bezug auf den Prozess wurde eine Neuentwicklung der Anwendung beschlossen. Diese Vorgehensweise wird durch zwei der Bewertungskriterien - Schnittstellenstabilität und Komplexität - gestützt. Lediglich die hohe Kritikalität der Anwendung spricht für eine iterative Umsetzung.

Der Prototyp wird daher ebenfalls als Neuentwicklung aufgebaut, ohne Bezug zu dem bestehenden System zu haben.

7.4.1 Domain-driven Design

Methoden des DDD können dabei helfen, ein gemeinsames Verständnis der Domäne zu erarbeiten und diese zu modellieren. Dies beinhaltet auch die Identifikation von einzelnen Subdomänen, welche potenzielle Microservice-Kandidaten darstellen. Ein guter Microservice erfüllt somit lediglich Aufgaben einer Subdomäne und kann diese eigenständig ausführen. Der Ausfall eines Services sorgt somit nicht dafür, dass das gesamte System betroffen ist.

Um im gesamten Team ein einheitliches Bild der vorliegenden Domäne zu erhalten, wurden die Methoden des Domain Storytellings [145] und des Event Stormings [146] eingesetzt. Diese dienen dazu, die Geschäftsprozesse und Interaktionen der Software zu identifizieren und entsprechende Subdomänen zu erkennen. Insbesondere das Event Storming hilft darüber hinaus dabei, auftretende Events der Domäne zu erkennen. Dies kann vor allem im Kontext von eventgetriebenen Microservices hilfreich sein.

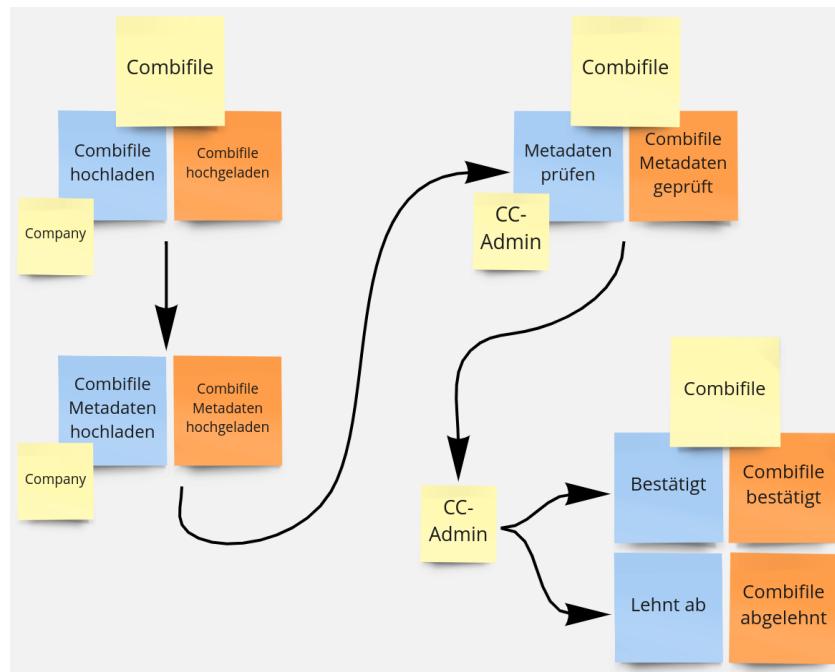


Abbildung 7.3: Ausschnitt der Ergebnisse des Event Stormings

Abbildung 7.3 zeigt einen Ausschnitt der Ergebnisse des Event Stormings. Ein Aktor führt dabei ein Kommando (blau) auf einem Aggregat (gelb) aus. Diese Aktion löst ein Domain Event (orange) aus.

Im vorliegenden Beispiel etwa wird eine Kombinationsdatei durch eine Gesellschaft (im EJP-CC Kontext auch *Company* genannt) hochgeladen. Der Upload löst ein entsprechendes Event

aus, auf welches gegebenenfalls reagiert werden kann. In ähnlicher Form lösen auch der Upload der zugehörigen Metadaten und das Bestätigen oder Ablehnen der Kombinationsdatei durch einen Administrator des EJP-CC entsprechende Events aus.

Auf Basis des Domain Storytellings und des Event Stormings konnte innerhalb des Teams ein einheitliches Bild der Domäne gewonnen werden. Darüber hinaus wurden erste Kontextgrenzen identifiziert, welche als potenzielle Microservice-Kandidaten fungieren (vgl. Abbildung 7.4). Die identifizierten Subdomänen stellen sich wie folgt dar:

Ziehungen:	Erstellung und Verwaltung der Ziehungen
Kombinationsdateien:	Entgegennahme und Prüfung der Kombinationsdateien
Gewinne:	Verwaltung der Gewinnzahlen und Gewinnermittlung
Abrechnungen:	Erstellung der Abrechnungen
Reporting:	Erstellung verschiedener Reports
Gewinninformationen:	Versenden von Informationsmails, etwa an die Presse

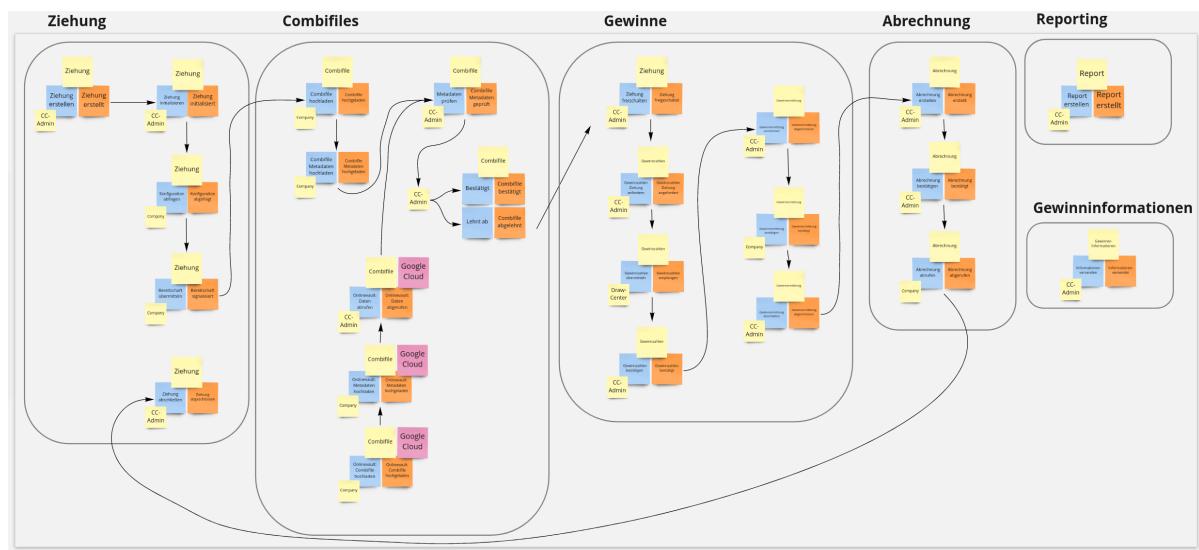


Abbildung 7.4: Ergebnisse des Event Stormings mit identifizierten Subdomänen

Um die erste Aufteilung weiter zu evaluieren, wird im Folgenden das Tool Service Cutter eingesetzt werden, um auf Basis des bestehenden Systems Microservice-Kandidaten zu ermitteln.

7.4.2 Service Cutter

Service Cutter ist ein Tool, welches auf einer abstrakten Beschreibung des Systems eine Dekomposition vornimmt. Dazu wird als minimale Eingabe ein ER-Modell des Systems benötigt. Da eine bestehende Software vorliegt, kann diese analysiert werden, um die Struktur des Systems zu extrahieren. Hierzu werden ebenfalls die zuvor angewendeten Programme Structure101 und Sonargraph genutzt. Diese erleichtern die Identifikation der vorhandenen Modellklassen und die Interaktionen sowie Beziehungen untereinander. Die anschließende Erstellung des Modells bleibt allerdings eine manuelle Aufgabe.

Um qualitativ hochwertigere Dekompositionsvorschläge zu generieren, benötigt Service Cutter weitere Informationen über die Software. Diese werden in Form von Use-Cases bereitgestellt, die die Zusammenhänge und Abhängigkeiten einzelner Komponenten beschreiben.

Service Cutter erwartet die bereitgestellten Informationen im JSON-Format, was die manuelle Erstellung der Daten erschwert und umständlich macht [90]. Um diesen Vorgang zu erleichtern, wird *Context Mapper*³ [147] genutzt. Dabei handelt es sich um ein Framework, welches im Kern eine *domänenspezifische Sprache* (DSL) für die Modellierung von DDD Anwendungen bereitstellt. Darüber hinaus enthält das Framework einige weitere Tools für das strategische Design und die Dekomposition in Microservices. Zu den Funktionalitäten von Context Mapper gehört ebenfalls die Generierung der von Service Cutter benötigten Eingabedaten auf Basis der DSL. Dies vereinfacht die Modellierung erheblich.

Auf Grundlage des existierenden EJP-CC wurde das bestehende Modell extrahiert und in der Context Mapper DSL modelliert. Ein Ausschnitt der Modellierung ist in Listing 7.1 dargestellt.

```
BoundedContext Eurojackpot {
    Aggregate Draw {
        Entity Draw {
            aggregateRoot
            long id
            int drawNumber
            Date drawDate
            - List<CompanyDrawState> companyDrawStates
            - DrawState drawState
        }
        ...
    }
    ...
}
```

Listing 7.1: Ausschnitt der Modellierung der bestehenden Entitäten mit Context Mapper

³<https://contextmapper.org/>

Technisch gesehen ist das ER-Modell bereits ausreichend, um eine erste Dekomposition vorzunehmen. Um bessere Ergebnisse zu erzielen, werden ebenfalls Use-Cases mithilfe der Context Mapper DSL erfasst. Listing 7.2 zeigt die Modellierung eines Use-Cases der EJP-CC Anwendung. Dabei handelt es sich um einen Anwendungsfall, bei dem eine Kombinationsdatei von einer Gesellschaft hochgeladen wird.

```
UseCase UploadCombifile {
    actor "Company"
    interactions
        read "Draw",
        read "Company",
        create "Combifile",
        "calculate" "Metadata" for "Combifile",
        update "CompanyDrawState"
}
```

Listing 7.2: Modellierung eines Use-Cases mithilfe von Context Mapper

Die Beschreibung des Use-Cases enthält den Aktor und die beteiligten Entitäten, welche gelesen, erstellt oder aktualisiert werden. Mithilfe dieser Beschreibungen kann Service Cutter die Kopplung zwischen verschiedenen Komponenten berücksichtigen und Microservice-Kandidaten generieren, die möglichst unabhängig voneinander sind.

Aus der modellierten Domäne werden mithilfe von Context Mapper die Eingabedaten im von Service Cutter erwarteten JSON-Format generiert. Verschiedene Dekompositionsoptionen können anschließend ausgewertet werden.

Service Cutter bietet die Möglichkeit verschiedene Kopplungskriterien zu gewichten. So können etwa semantische Ähnlichkeit oder Konsistenz unterschiedlich gewichtet werden und somit zu verschiedenen Dekompositionen führen. Die subjektive Gewichtung des Graphen führt teilweise zu Kritik, da Beziehungen zwischen Komponenten durch die persönliche Einschätzung der Entwickler verfälscht werden können [29]. Gleichzeitig ermöglicht es die Generierung, Evaluation und Diskussion verschiedener Aufteilungen unter Priorisierung unterschiedlicher Kopplungskriterien.

Darüber hinaus lassen sich verschiedene Clusteralgorithmen anwenden, welche sowohl deterministischer als auch nicht-deterministischer Natur sind. Im Rahmen der Fallstudie konnte dabei festgestellt werden, dass insbesondere die Anwendung des Epidemic Label Propagation Algorithmus [94] (im Kontext von Service Cutter auch *Leung-Algorithmus*) zu tendenziell groben Service-Schnitten führt.

Abbildung 7.5 zeigt einen Service-Schnitt unter Verwendung des Leung-Algorithmus. In diesem Fall wurden lediglich zwei Microservices ermittelt. Service A übernimmt dabei die

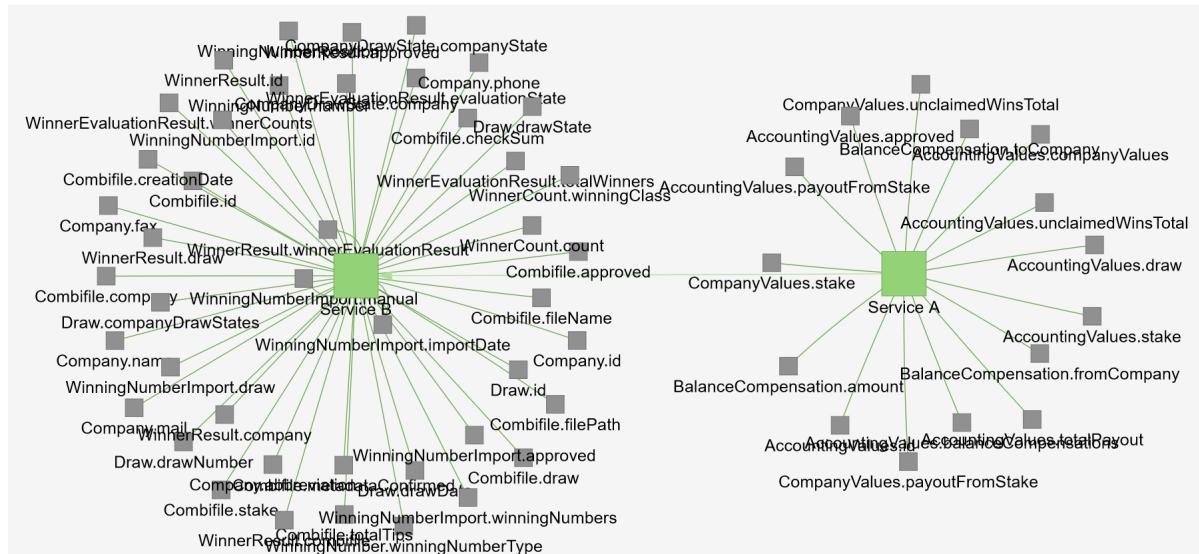


Abbildung 7.5: Ermittelte Microservices unter Verwendung des Leung-Algorithmus

Abrechnung, während der zweite Microservice - Service B - für alle weiteren Geschäftsfähigkeiten verantwortlich ist. Der zweite Service ist somit für viele verschiedene Aufgaben zuständig und wurde somit innerhalb des Teams als zu grob klassifiziert.

Bessere Erkenntnisse wurden unter Anwendung der Markov- [93] und Chinese-Whispers-Algorithmen [96] gewonnen. Abbildung 7.6 zeigt die von Service Cutter ermittelte Aufteilung der Microservices unter Verwendung des Markov-Algorithmus. In diesem Fall wurden sechs Microservices ermittelt, womit eine wesentlich feingranularere Aufteilung vorliegt.

Die Auswertung der Dekomposition zeigt, dass dynamisch erzeugte Daten wie Reports und Informationsmails nicht als eigenständige Microservices ermittelt wurden. Im Gegensatz dazu propagiert Service Cutter die feinere Aufteilung zweier Microservices, welche bereits im Rahmen des DDD identifiziert wurden.

Der zuvor identifizierte Gewinne-Service wurde in diesem Fall in zwei kleinere Microservices unterteilt. Ein Service, welcher ausschließlich für die Gewinnzahlen verantwortlich ist und darüber hinaus ein weiterer Service, welcher ausschließlich für die Gewinnermittlung verantwortlich ist.

Weiter enthält der generierte Service-Schnitt einen separaten Microservice, welcher ausschließlich für die Verwaltung der Gesellschaften verantwortlich ist.

Beide neu eingeführten Services wurden im Entwicklerteam diskutiert und für sinnvoll erachtet, da sowohl die Gesellschaften als auch die Gewinnzahlen als eigenständige Subdomäne betrachtet werden können und ein separater Microservice somit zielführend ist.

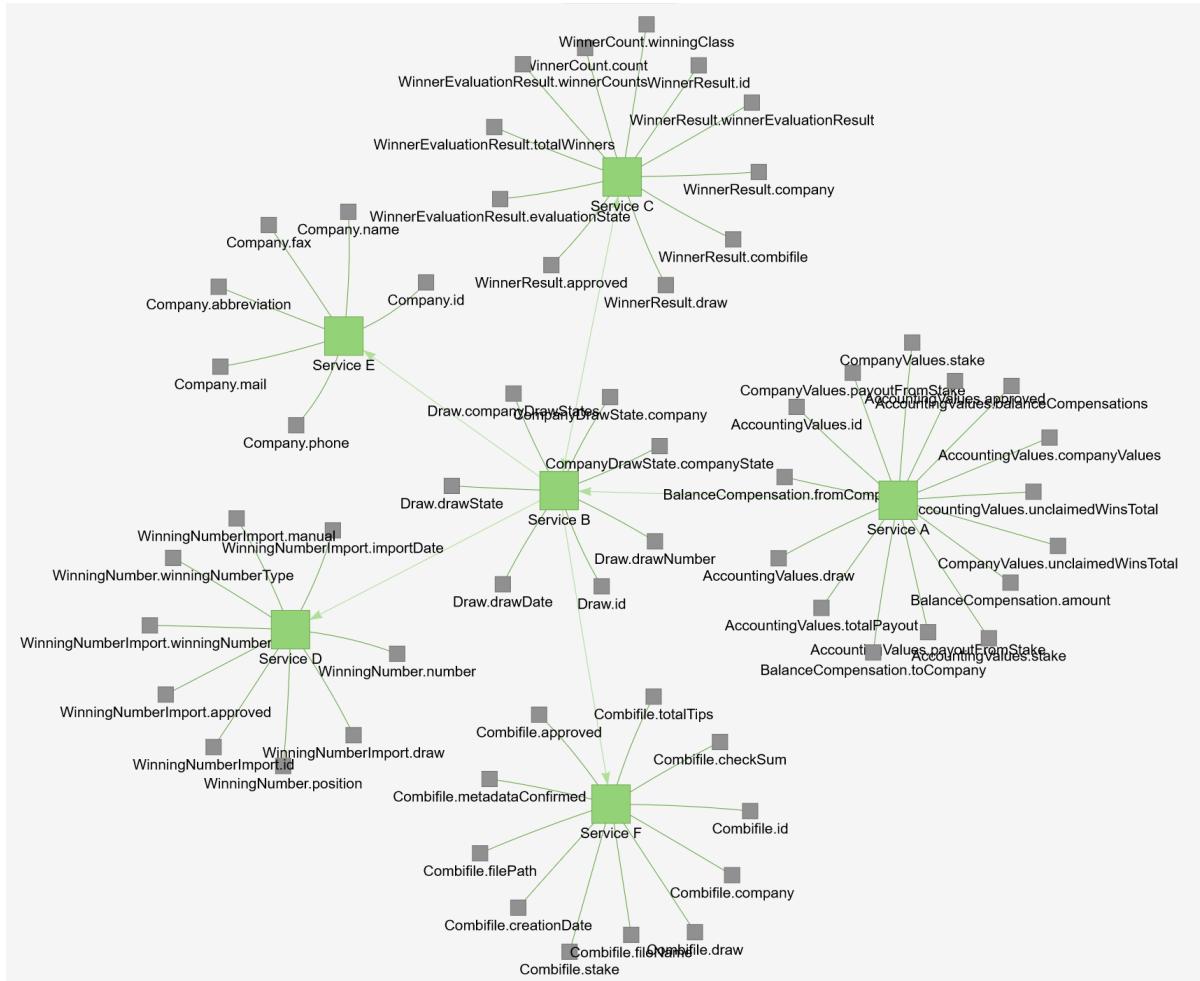


Abbildung 7.6: Ermittelte Microservices unter Verwendung des Markov-Algorithmus

7.4.3 Identifizierte Microservices

Im Rahmen der Konzeption wurden sowohl Methodiken des DDD als auch ein unterstützendes Tool verwendet. Beide Methoden wurden dabei als gewinnbringend erachtet, um ein gemeinsames Verständnis der Domäne zu erhalten, Kontextgrenzen zu identifizieren und verschiedene Service-Schnitte zu visualisieren.

Insbesondere die Visualisierung verschiedener Schnitte und die durch Service Cutter dargestellten Abhängigkeiten der identifizierten Services wurden dabei innerhalb des Teams als wertvolle Diskussionsgrundlage empfunden.

Durch Kombination der Erkenntnisse des DDD und der Anwendung des Service Cutters wurden letztlich acht Microservices identifiziert:

Gesellschaften: Anlage und Verwaltung der Gesellschaften

Ziehungen: Erstellung und Verwaltung der Ziehungen

Kombinationsdateien: Entgegennahme und Prüfung der Kombinationsdateien

Gewinnzahlen:	Entgegennahme und Freigabe der Gewinnzahlen
Gewinnermittlungen:	Ermittlung der Gewinner jeder Gesellschaft
Abrechnungen:	Erstellung der Abrechnungen
Reporting:	Erstellung verschiedener Reports
Gewinninformationen:	Versenden von Informationsmails, etwa an die Presse

7.5 Umsetzung

Das folgende Unterkapitel beschreibt die Implementation der Microservice-Architektur. Neben den identifizierten Services werden in einer Microservice-Architektur typischerweise weitere Komponenten benötigt, um die Kommunikation und Verwaltung der Services zu gewährleisten.

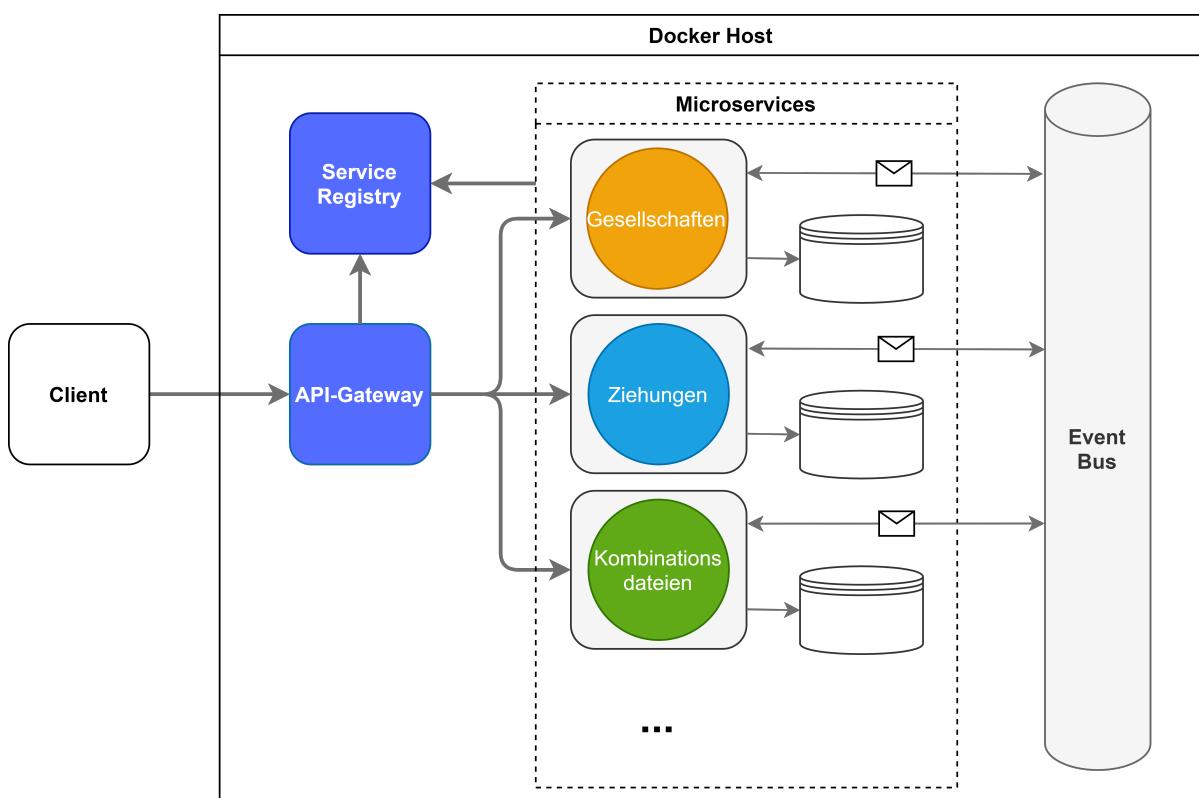


Abbildung 7.7: Architektur des EJP-CC auf Basis von Microservices

Abbildung 7.7 zeigt die Microservice-Architektur, welche in Form eines Prototyps umgesetzt wurde. Aus Gründen der Übersichtlichkeit werden lediglich drei der acht identifizierten Microservices abgebildet. Alle weiteren Services sind analog in die Architektur eingebunden.

Die Architektur besteht dabei aus folgenden Komponenten:

Service Registry

Muss ein Service mit einem weiteren Service kommunizieren, so ist es notwendig, dass dieser die IP-Adresse und den Port des Zielservices kennt. In Microservice-Architekturen kann es eine große Anzahl an Services geben. Darüber hinaus können im Rahmen der Skalierung dynamisch neue Instanzen erzeugt und gelöscht werden. Eine feste Kodierung von IP-Adressen und Ports in der Konfiguration eines Services ist somit keine Option.

Um Services gegenseitig erreichbar zu machen, existiert deshalb eine Service Registry. Diese dient als zentrale Verwaltung für die Zieladressen aller Microservices. Beim Start eines Microservices, registriert sich dieser unter Angabe der eigenen IP-Adresse und des Port bei der Service Registry. Andere Services können die Service Registry dann bei Bedarf anfragen, unter welcher Adresse und Port ein bestimmter Service aktuell erreichbar ist.

Im entwickelten Prototyp wird die Service Registry *Eureka*⁴ verwendet, welche von Netflix als Open-Source-Projekt entwickelt wird.

API-Gateway

Das API-Gateway stellt den zentralen Einstiegspunkt für alle Anfragen der Clients dar. Mögliche Clients sind in diesem Fall etwa die teilnehmenden Gesellschaften. Diese benötigen keine Kenntnisse über die interne Architektur der Anwendung. Es ist somit nicht gewollt, dass eine Gesellschaft verschiedene Microservices selbst anspricht. Aus diesem Grund wurde ein API-Gateway eingeführt, welches als Proxy dient. Es nimmt Anfragen entgegen und leitet diese an den jeweils zuständigen Microservice weiter. Um die aktuelle Lokation eines Microservices zu ermitteln, fragt das API-Gateway die Service Registry an.

Diese Komponente wird mithilfe des *Spring Cloud Gateway*⁵ Projekts umgesetzt. Regeln, auf deren Basis ein Routing vorgenommen wird, lassen sich mit geringem Aufwand extern konfigurieren.

Listing 7.3 zeigt die Konfiguration einer Gateway-Route. Alle Anfragen, deren Pfad mit dem Muster `/companies/**` übereinstimmt, werden entsprechend an den Gesellschaften-Microservice weitergeleitet. Mithilfe des API-Gateways lässt sich somit ein zentraler Einstiegspunkt für alle Clients bereitstellen und die interne Komplexität des Systems kann verborgen werden. Darüber hinaus kann das Gateway weitere Aufgaben wie die Authentifizierung übernehmen.

Microservices

Die acht Microservices sind für die Bearbeitung der ermittelten Geschäftsfähigkeiten verantwortlich. Jeder Service übernimmt dabei nur eine geringe Anzahl an Aufgaben innerhalb einer definierten Subdomäne. Die Eigenständigkeit der Services sorgt dafür, dass auch bei Ausfall eines Services weite Teile des Systems weiter nutzbar sind. Um die Kopplung der

⁴<https://github.com/Netflix/eureka>

⁵<https://spring.io/projects/spring-cloud-gateway>

```
application.yml
```

```

1  spring:
2    cloud:
3      gateway:
4        routes:
5          - id: companies
6            uri: lb://companies
7            predicates:
8              - Path=/companies/**

```

Listing 7.3: Konfiguration einer Gateway-Route

Microservices zu minimieren, besitzt jeder Service seine eigene Datenhaltung. Aufgrund bestehender Kenntnisse wurde für die Implementierung der Microservices das *Spring Boot*⁶ Framework verwendet. Dieses bietet viele Integrationen für die Entwicklung von Microservices, eine automatische Konfiguration sowie einen integrierten Webserver und ist somit eine weit verbreitete Wahl für die Implementierung von Microservices [6], [148].

Event Bus

Einige der Microservices sind auf die gleichen Daten angewiesen. Etwa benötigen fast alle Services Kenntnisse über die bestehenden Ziehungen, welche vom Ziehungen-Service verwaltet werden.

Die erste Möglichkeit diese Informationen zu erhalten, ist die direkte Punkt-zu-Punkt-Kommunikation. Andere Services fragen bei Bedarf die Informationen zu einer Ziehung vom Ziehungen-Service ab. Während diese Vorgehensweise leicht umzusetzen ist, sorgt sie gleichzeitig für eine starke Kopplung der Services untereinander. Fällt der Ziehungen-Service aus, sind alle weiteren Services, welche Ziehungsinformationen benötigen, betroffen.

Um diese Situation zu vermeiden, ist eine Duplikation der benötigten Daten in einer Microservice-Architektur gängige Praxis. Jeder Microservice hält seine eigenen Ziehungsinformationen, wobei lediglich die benötigten Attribute gespeichert werden. Änderungen an den Ziehungsdaten werden über einen Event Bus publiziert. Jeder Service kann die von ihm gewünschten Events konsumieren und seinen eigenen Datenbestand entsprechend anpassen. Diese eventbasierte Form der Kommunikation in Verbindung mit der eigenen Datenhaltung eines jeden Services sorgt dafür, dass jeder Microservice maximal unabhängig ist und seine Geschäftsfähigkeit ohne synchrone Kommunikation mit anderen Services erfüllen kann.

Als Plattform nutzt der Prototyp *Apache Kafka*⁷.

⁶<https://spring.io/projects/spring-boot>

⁷<https://kafka.apache.org/>

```
@Service
@Transactional
public class KafkaInboxListener implements KafkaInboxEventHandler {
    @Autowired
    private DrawRepository drawRepository;

    @KafkaListener(topics = "${kafka.ejpcc.topic}")
    public void incomingEvent(AbstractEvent event) {
        event.accept(this);
    }

    @Override
    public void handleDrawEvent(DrawEvent event) {
        if (event.getEventType() == EventType.DELETE) {
            drawRepository.deleteById(event.getDraw().getId());
        } else {
            drawRepository.save(event.getDraw());
        }
    }
}
```

Listing 7.4: Verarbeitung eines Domänen-Events durch einen Microservice

Listing 7.4 zeigt die Verarbeitung eines Events, welches die Ziehung betrifft. Je nach Typ des Events wird die eigene Kopie der Ziehung entweder gelöscht oder aktualisiert. Dabei ist zu beachten, dass im Rahmen des EJP-CC nicht nur Events veröffentlicht werden, welche die Ziehung betreffen. Jeder Service ist in der Lage, eigene Events zu veröffentlichen und verschiedene Events zu konsumieren. Bei der Identifizierung der verschiedenen Events erweisen sich die Erkenntnisse des Event Stormings (vgl. Abbildung 7.4) als hilfreich. Dieses beschäftigt sich explizit mit den Events, die in einer Domäne auftreten.

Durch den Aufbau einer eventbasierten Architektur können somit lose gekoppelte Services erstellt werden. Im Rahmen des Prototyps konnte auf diese Weise auf synchrone Kommunikation zwischen den Microservices vollständig verzichtet werden. Jeder Microservice ist im Besitz aller Daten, die er für die Bearbeitung einer Anfrage benötigt und kann diese somit vollkommen unabhängig verarbeiten.

Darüber hinaus ist zu beachten, dass für die Komponenten der Architektur auch andere Technologien eingesetzt werden können. Im produktiv angestrebten Kubernetes-Umfeld lassen sich etwa Funktionen des API-Gateways mit einem *Ingress Controller*⁸ umsetzen. Der grundätzliche Aufbau der Microservice-Architektur wird dadurch allerdings nicht beeinflusst.

⁸<https://kubernetes.io/docs/concepts/services-networking/ingress-controllers/>

7.6 Bewertung der Variabilität

Die Verarbeitung einer Ziehung wird in Abschnitt 7.1 im Kontext der Eurojackpot Lotterie beschrieben. Die grundsätzlichen Komponenten der Verarbeitung wie Ziehungen, Kombinationsdateien, Gewinnermittlungen und Abrechnungen sind allerdings auch im Kontext weiterer Spielarten vorhanden. So steht etwa eine analoge Verarbeitung des klassischen Lotto 6aus49 im Raum. Die Bewertung der Variabilitätsstrategien soll dabei helfen, geeignete Vorgehensweisen zu identifizieren, um die erstellten Microservices auch im Kontext anderer Spielarten nutzbar zu machen.

In diesem Zusammenhang zeigt sich vorab eine positive Eigenschaft von Microservices. Durch die Erstellung von kleinen Einheiten, welche nur eine Geschäftsfähigkeit erfüllen, wird implizit bereits eine hohe Wiederverwendbarkeit erreicht [4], [24], [28]. Darüber hinaus kann für jeden Microservice eine eigene Variabilitätsstrategie verfolgt werden. Während einige Services unter Umständen keine Unterschiede in Bezug auf unterschiedliche Spielarten aufweisen, kann für Services mit einem hohen Grad an Variabilität eine andere Vorgehensweise gewählt werden.

Im Folgenden soll anhand von zwei Beispielen eine geeignete Variabilitätsstrategie unter Berücksichtigung der entwickelten Bewertungskriterien ermittelt werden. Dazu werden die beiden Services, die für die Kombinationsdateien und die Abrechnungen verantwortlich sind, herangezogen und geprüft, inwieweit diese im Kontext anderer Spielarten genutzt werden können.

7.6.1 Kombinationsdateien

Der Microservice für die Kombinationsdateien verantwortet die Annahme der Dateien, welche die Tipps der teilnehmenden Gesellschaften enthalten. Darüber hinaus findet die Berechnung des Gesamteinsatzes und einer Prüfsumme statt, welche mit den ermittelten Daten der Gesellschaft übereinstimmen müssen. Die Nutzung dieses Services in einem weiteren Spielkontext wird im Folgenden untersucht.

Grad der Variabilität

In Bezug auf die Kombinationsdateien ist ein geringer Grad der Variabilität vorhanden. In allen Fällen liegt eine komprimierte Datei vor, welche die Tipps einer Gesellschaft enthält. Die Zeilennummer repräsentiert dabei die getippte Zahlenkombination, der Wert in einer Zeile gibt an, wie oft die entsprechende Kombination getippt wurde.

Sowohl das Dateiformat als auch der Aufbau einer Datei sind somit unabhängig von der Spielart gleich gehalten. Unterschiede bestehen in der Anzahl der möglichen Zahlenkombinationen

und damit in der erwarteten Zeilenanzahl der Kombinationsdatei. Auch der Einsatz pro Tipp kann sich zwischen unterschiedlichen Spielarten unterscheiden.

Die Annahme, Verarbeitung und Bestätigung der Kombinationsdateien läuft allerdings unabhängig von der Spielart nach dem gleichen Muster ab. Lediglich einige definierte Attribute unterscheiden sich je Spielart. Der Grad der Variabilität kann somit insgesamt als gering eingestuft werden. Das Konzept besagt in diesem Fall, dass sich in Fällen mit geringer Variabilität die Verarbeitung mehrerer Spielvarianten in einer Codebasis mithilfe von Feature Flags oder einer externen Konfiguration anbietet.

Wartbarkeit

Die Wartbarkeit ist zu einem Großteil von der Komplexität der Codebasis abhängig. Die Komplexität der Codebasis wiederum wird insbesondere durch den Grad der Variabilität und die gewählte Variabilitätsstrategie beeinflusst (vgl. Unterabschnitt 6.3.3).

Im Falle der Feature Flags beziehungsweise der externen Konfiguration werden alle Produktvarianten innerhalb der gleichen Codebasis verwaltet. Um die Komplexität der Codebasis gering zu halten, ist in diesen Fällen Voraussetzung, dass sich alle Produktvarianten auch in Zukunft gleich verhalten. Avisierte Änderungen, die nur einen Teil der Varianten betreffen, führen anderenfalls zu einer Erhöhung der Komplexität und somit zu einer verminderter Wartbarkeit des Microservices.

Im Falle der Kombinationsdateien besteht innerhalb des Projektteams die Meinung, dass eine unterschiedliche Verarbeitung der Kombinationsdateien für verschiedene Spielarten als sehr unwahrscheinlich gilt. Insofern besteht hohes Vertrauen, dass mögliche Änderungen an der Fachlichkeit der Kombinationsdateien alle Produktvarianten in gleichem Maße betreffen. Die Verarbeitung von Kombinationsdateien verschiedenster Spielarten in einer Codebasis hat darüber hinaus den Vorteil, dass Änderungen - im Gegensatz etwa zu Code-Duplikationen - nur einmalig angewendet werden müssen. Aus Sicht der Wartbarkeit ist somit eine Umsetzung mittels Feature Flags oder extern konfigurierten Instanzen ebenfalls denkbar.

Performance

Da es sich bei der Microservice-Architektur um ein verteiltes System handelt, können Antwortzeiten insbesondere dann erhöht sein, wenn für die Erfüllung einer Geschäftsfähigkeit mehrere Services involviert sind. In diesen Fällen wird eine Kommunikation über das Netzwerk notwendig, was zu erhöhten Latenzen führt. Insbesondere bei hohen Anforderungen an die Performance soll somit auf frequente Inter-Service Kommunikation - wie etwa im Falle eines geteilten Services (vgl. Abschnitt 5.5) - verzichtet werden.

Im Rahmen der Verarbeitung von Kombinationsdateien existieren keine Anforderungen in Bezug auf hohe Performance oder Echtzeitverarbeitung. Etwaige Latenzen, welche durch

einen zusätzlichen Serviceaufruf entstehen, können somit vernachlässigt werden. Die gesamte Ziehungsverarbeitung ist darüber hinaus ein Prozess, welcher durch manuelle Prüfungen und Bestätigungen geprägt ist. Hochgeladene Dokumente und berechnete Daten werden in jedem Schritt manuell geprüft, bestätigt und mit der unabhängigen zweiten Implementierung des EJP-CC abgeglichen. Da zusätzliche Netzwerklatenzen somit toleriert werden können, existieren aus Sicht der Performance keine Limitationen.

Verfügbarkeit

Bei dem EJP-CC handelt es sich um eine kritische Anwendung (vgl. Unterabschnitt 7.3.3), sodass eine hohe Verfügbarkeit gewährleistet sein muss. Dies gilt auch für alle weiteren Spielarten, deren Verarbeitung in Zukunft denkbar ist.

Im Rahmen der Verfügbarkeit sind zwei Fälle zu unterscheiden. Die Verfügbarkeit kann eingeschränkt werden, indem etwa ein geteilter Service aufgrund zu hoher Last nicht alle Anfragen bearbeiten kann. Dies stellt einen potenziellen Ausfallpunkt dar, der alle abhängigen Services beeinflusst. Ein Ausfall aufgrund von hoher Last ist im Rahmen der vorliegenden Anwendung nicht zu erwarten. Es existiert für alle Spielarten lediglich eine begrenzte Anzahl an Gesellschaften, sodass mit einer überschaubaren Anzahl an Clients zu rechnen ist, welche auf einen Service zugreifen.

Darüber hinaus wird eine produktive Inbetriebnahme der Anwendung in Kubernetes avisiert. Dieses unterstützt eine automatische Skalierung einzelner Pods, was die Verfügbarkeit weiter erhöht.

Ein zweiter Aspekt in Bezug auf die Verfügbarkeit ist die Anzahl der Produktvarianten, welche innerhalb einer Microservice-Implementierung verwaltet wird. Ein Fehler in der Variabilität einer Spielart kann im Falle von Feature Flags etwa dazu führen, dass der Service ausfällt und alle weiteren Spielarten betroffen sind. Um diesbezüglich eine hohe Verfügbarkeit zu erreichen, sieht das Konzept vor, dass Variabilitäten einzelner Produktvarianten in eigenen Codebasen implementiert werden. Somit existiert für jede Produktvariante ein spezifischer Microservice. Der Ausfall einer Spielart hat damit keine Auswirkungen auf die Microservices anderer Spielarten.

Unter Berücksichtigung der Verfügbarkeit ergibt es somit Sinn, unterschiedliche Spielarten in verschiedenen Microservice-Implementationen bereitzustellen. Dies ist insbesondere dann der Fall, wenn ein hoher Grad der Variabilität vorliegt. Bei einem geringen Grad der Variabilität und somit einer nahezu identischen Verarbeitung unterschiedlicher Spielarten kann dieser Punkt vernachlässigt werden. Letzteres trifft auch auf den Kombinationsdateien-Service zu. Eine hohe Verfügbarkeit lässt sich somit unter Berücksichtigung geeigneter Skalierungsmaßnahmen mit allen Variabilitätsstrategien erreichen.

Kosten

In Bezug auf die Kosten sieht das Konzept eine Berücksichtigung der Wartungs- und Bereitstellungskosten vor. Die Wartungskosten stehen in direktem Zusammenhang mit der zuvor bewerteten Wartbarkeit. Die Bereitstellungskosten ergeben sich aus der erwarteten Last und der Anzahl der bereitgestellten Services. Eine hohe Anzahl an Services sorgt somit in der Regel für höhere Betriebskosten. Müssen diese Kosten minimiert werden, kann es somit sinnvoll sein, mehrere Produktvarianten innerhalb eines Services zu verarbeiten [141].

Im Falle des EJP-CC herrscht keine Konkurrenzsituation mit weiteren Unternehmen des freien Marktes. WestLotto erhält für die Koordination und Verwaltung der Ziehungen eine feste Vergütung durch alle teilnehmenden Gesellschaften. Während die anfallenden Kosten die Vergütung nicht übersteigen sollen, existiert gleichzeitig kein Kostendruck durch einen Wettbewerb mit anderen Unternehmen. Eine Minimierung der Kosten ist somit immer erstrebenswert, gleichzeitig sind durch die vorhandene feste Vergütung gewisse Freiheitsgrade vorhanden.

Darüber hinaus ist die Anzahl der Microservices und die erwartete Last aufgrund der geringen Anzahl an Clients ebenfalls ein Indiz für tendenziell geringe Betriebskosten.

Eine gute Wartbarkeit mit entsprechend langfristig geringen Wartungskosten wird im Kontext des vorliegenden Projektes daher als bedeutsamer eingeschätzt.

7.6.2 Abrechnungen

Der Abrechnungen-Microservice ist verantwortlich für die Berechnung der auszuzahlenden Beträge zwischen den teilnehmenden Gesellschaften. Im Folgenden soll betrachtet werden, welche Variabilitätsstrategie sich eignet, um die Abrechnung im Kontext einer neuen Spielart - in diesem Beispiel Lotto 6aus49 - zu vollziehen. Dazu werden erneut die im Konzept entwickelten Kriterien ausgewertet.

Performance, Verfügbarkeit & Kosten

Bei Bewertung dieser Kriterien kann auf die Ergebnisse aus Unterabschnitt 7.6.1 zurückgegriffen werden. Die dort gewonnenen Erkenntnisse beziehen sich nicht ausschließlich auf den Kombinationsdateien-Microservice, sondern gelten analog für den überwiegenden Teil aller Microservices.

Als Beispiel sei hier die hohe Kritikalität genannt. Diese betrifft die gesamte Anwendung. Eine hohe Verfügbarkeit muss dementsprechend für alle Services der Architektur gewährleistet sein.

Grad der Variabilität

Große Unterschiede zu dem für die Kombinationsdateien verantwortlichen Service liegen im vorhandenen Grad der Variabilität vor. Sowohl im Kontext von Eurojackpot als auch im Kontext von Lotto 6aus49 werden im Rahmen der Abrechnung ähnliche Berechnungen vorgenommen. Der Microservice berechnet in beiden Fällen die Höhe der Ausschüttung in jeder Gewinnklasse. Mithilfe der Daten der Gewinnermittlung wird berechnet, welchen Betrag jede Gesellschaft ihren Kunden auszahlen muss. Aus dem Auszahlungsbetrag und den Einsätzen der Gesellschaft ergibt sich die Differenz, die eine Gesellschaft entweder zahlen muss, oder ausgezahlt bekommt. Darauf basierend werden Ausgleichszahlungen berechnet, die die Gesellschaften untereinander leisten müssen, sodass alle Gewinne ausgezahlt werden können. Während der grobe Ablauf sich in beiden Spielarten ähnelt, liegen dennoch einige Variabilitäten vor:

- **Garantierter Jackpot:** Im Falle von Eurojackpot wird in der höchsten Gewinnklasse eine Ausschüttung von mindestens 10 Millionen Euro garantiert. Liegen die Spieleinsätze unterhalb der Summe, welche ausgezahlt wird, muss die Differenz durch die teilnehmenden Gesellschaften getragen werden. Um das so entstehende Risiko finanzieller Verluste zu minimieren, wurde der sogenannte Booster-Fonds eingeführt. Mit jeder Ziehung fließen 9 % der Spieleinsätze in diesen Fonds. Übersteigt die garantiert zugesicherte Auszahlung die Spieleinsätze, so wird die Differenz durch den Booster-Fonds ausgeglichen. Der Fonds dient somit als Puffer und verringert das Risiko, dass die teilnehmenden Gesellschaften mit Eigenkapital Auszahlungen vornehmen müssen.
Im Falle von Lotto 6aus49 ist kein garantierter Jackpot und somit auch kein Booster-Fonds vorhanden. Verrechnungen werden lediglich zwischen den Gesellschaften vorgenommen. Dieser Aspekt sorgt für große Unterschiede in der Abrechnung beider Spielarten.
- **Vorhandenes Treuhandkonto:** Eurojackpot und Lotto 6aus49 unterscheiden sich darüber hinaus in der Verwaltung von nicht ausgezahlten Gewinnen. Ist etwa die höchste Gewinnklasse nicht besetzt, so erhöht sich der Jackpot in der folgenden Ziehung. Die Spieleinsätze, die aufgrund unbesetzter Gewinnklassen nicht ausgezahlt werden müssen, werden im Eurojackpot-Umfeld von allen Gesellschaften auf ein Treuhandkonto überwiesen. Im Gegensatz dazu existiert im Lotto 6aus49 Umfeld kein Treuhandkonto, welches im Rahmen der Abrechnung verwaltet und berücksichtigt werden muss. Stattdessen bleiben alle Gesellschaften im Besitz des Geldes und es wird lediglich eine Buchführung über die nicht ausgezahlten Gewinne geführt. Auch dieser Aspekt hat entsprechend Auswirkungen auf die jeweiligen Abrechnungen.
- **Ausgleich von Quotenanomalien:** Eine Quote beschreibt, welcher Geldbetrag jedem Gewinner innerhalb einer Gewinnklasse ausgezahlt wird. Sie ergibt sich aus dem

Gesamtbetrag, welcher in der Gewinnklasse ausgeschüttet wird, dividiert durch die Anzahl der Gewinner. Es kann dabei der Fall eintreten, dass die Quote einer niedrigeren Gewinnklasse die Quote einer höheren Gewinnklasse übersteigt. Dies wird als Quoten-anomalie bezeichnet und führt dazu, dass ein Spieler mit weniger richtigen Tipps mehr Geld erhielt als ein Spieler mit mehr richtigen Tipps. Um diesen Zustand zu vermeiden, werden im Falle von auftretenden Quotenanomalien Verrechnungen vorgenommen, die sicherstellen, dass die Quote einer niedrigeren Gewinnklasse die Quote einer höheren Gewinnklasse nicht übersteigt. Die Berechnung dieses Ausgleichs unterscheidet sich in Eurojackpot und Lotto 6aus49.

- **Rundung von Beträgen:** Im Rahmen der Abrechnung müssen Rundungen vorgenommen werden. Ein Jackpot von 10 Millionen Euro etwa lässt sich nicht restlos auf drei Gewinner aufteilen. Um allen Gewinnern einer Gewinnklasse den exakt gleichen Betrag auszuzahlen, wird deshalb eine Abrundung vorgenommen. Die entsprechende Differenz (auch *Rundungsspitze*) wird im Rahmen von Eurojackpot dem Booster-Fonds zugeführt. Im Rahmen von Lotto 6aus49 werden die Rundungsspitzen dem Einsatz der nächsten Ziehung zugeführt.

Es lässt sich somit festhalten, dass ein sehr hoher Grad an Variabilität besteht. Das Konzept besagt in diesem Fall, dass die Variabilitäten der einzelnen Produktvarianten in unterschiedlichen Codebasen implementiert werden sollen. Die Verwaltung beider Spielarten in der gleichen Codebasis, wie sie im Falle von Feature Flags und extern konfigurierbaren Instanzen vorgenommen wird, sorgt für eine erhöhte Komplexität der Anwendung.

Auf Basis dieses Bewertungskriteriums wird somit eine Umsetzung mittels geteilter Services, geteilter Bibliotheken, Sidecars oder Code-Duplikationen empfohlen.

Wartbarkeit

Der Grad der Variabilität beeinflusst maßgeblich die Wartbarkeit der unterschiedlichen Strategien. Im vorliegenden Fall ist ein hoher Grad der Variabilität gegeben, der sich über alle Bereiche der Abrechnung erstreckt. Eine Umsetzung mittels Feature Flags oder extern konfigurierten Instanzen sorgt somit für eine hohe Komplexität der Codebasis, da die Variabilitäten verschiedenster Spielarten in einer Codebasis vereint werden. Dies erschwert wiederum die Wartbarkeit. Aus Sicht der Wartbarkeit ist eine Umsetzung mithilfe von Feature Flags oder der externen Konfiguration somit ebenfalls keine empfohlene Option.

Ein weiterer Aspekt, der die Wartbarkeit der verbleibenden Optionen unterscheidet, sind Änderungen an Gemeinsamkeiten der Codebasis. Im Falle von geteilten Services sowie geteilten Bibliotheken müssen Änderungen lediglich einmalig vorgenommen werden. Liegt eine Code-Duplikation vor, so müssen Änderungen, welche alle Spielarten betreffen, in jeder Codebasis vorgenommen werden, was für erhöhte Aufwände sorgt.

Berücksichtigt man darüber hinaus Nachteile wie hohe Kopplung, die durch geteilte Bibliotheken eingeführt werden (vgl. Abschnitt 5.4), so stellt aus Sicht der Wartbarkeit ein geteilter Service beziehungsweise ein Sidecar eine geeignete Möglichkeit dar, um gemeinsame Routinen verschiedener Spielarten innerhalb einer Codebasis zu kapseln.

7.7 Konzept für das Hinzufügen weiterer Spielvarianten

Im Rahmen einer angedachten Erweiterung weiterer Spielarten wurden exemplarisch zwei Microservices betrachtet. Zu diesem Zweck wurden die im Konzept entwickelten Bewertungskriterien angewendet.

In Bezug auf den Kombinationsdateien-Microservice spricht der geringe Grad der Komplexität für eine Umsetzung mithilfe von Feature Flags oder extern konfigurierten Instanzen. Aufgrund der großen Gemeinsamkeiten lässt sich somit auch eine gute Wartbarkeit gewährleisten, da Änderungen lediglich an einer Codebasis vorgenommen werden müssen.

Die Performance spielt eine eher untergeordnete Rolle und stellt somit keine Einschränkung in Bezug auf die Auswahl einer Variabilitätsstrategie dar.

Das Verarbeiten mehrerer Spielvarianten innerhalb einer Codebasis stellt aufgrund des sehr geringen Grads der Variabilität kein Problem für die Verfügbarkeit dar. Eine technisch hohe Verfügbarkeit kann darüber hinaus mit geeigneten Skalierungsmaßnahmen erreicht werden, die Kernbestandteile von Orchestrierungs-Tools wie Kubernetes sind. In Bezug auf die Verfügbarkeit lässt sich argumentieren, dass konfigurierbare Instanzen einen Vorteil gegenüber Feature Flags haben, da jede Spielart durch eine eigenständige Microservice-Instanz bearbeitet wird und somit eine höhere Unabhängigkeit vorhanden ist.

Die Bereitstellungskosten konnten als sekundäres Kriterium identifiziert werden. Für wichtiger werden die Wartungskosten erachtet, welche wiederum besonders vom Grad der Variabilität abhängen.

Insgesamt wurde auf Grundlage der Bewertung die Umsetzung mithilfe von extern konfigurierten Instanzen als sinnvollste Option erachtet. Gegenüber Feature Flags bieten Sie einen Vorteil, der im Rahmen des Konzeptes noch nicht berücksichtigt wurde: die Trennung von Mandanten. Verschiedene Spielarten werden von unterschiedlichen Gesellschaften unterstützt. Während es sich bei Eurojackpot um eine Lotterie handelt, an der mehrere europäische Länder teilnehmen, sind im Falle von Lotto 6aus49 lediglich deutsche Gesellschaften beteiligt. Je nach Spielart liegen somit unterschiedliche Mandanten vor. Die Daten verschiedener Mandanten und Spielarten müssen strikt voneinander getrennt werden. Dies ist insbesondere im Falle von Feature Flags eine Herausforderung, die die Einführung geeigneter Muster erforderlich macht [149], da mehrere Mandanten und Spielarten innerhalb einer Service-Instanz verwaltet werden.

Eine separate Microservice-Instanz mit eigener Datenhaltung, die lediglich für die Verwaltung einer spezifischen Spielart und deren Mandanten zuständig ist, stellt daher eine Option dar, die mit geringerem Aufwand umsetzbar ist.

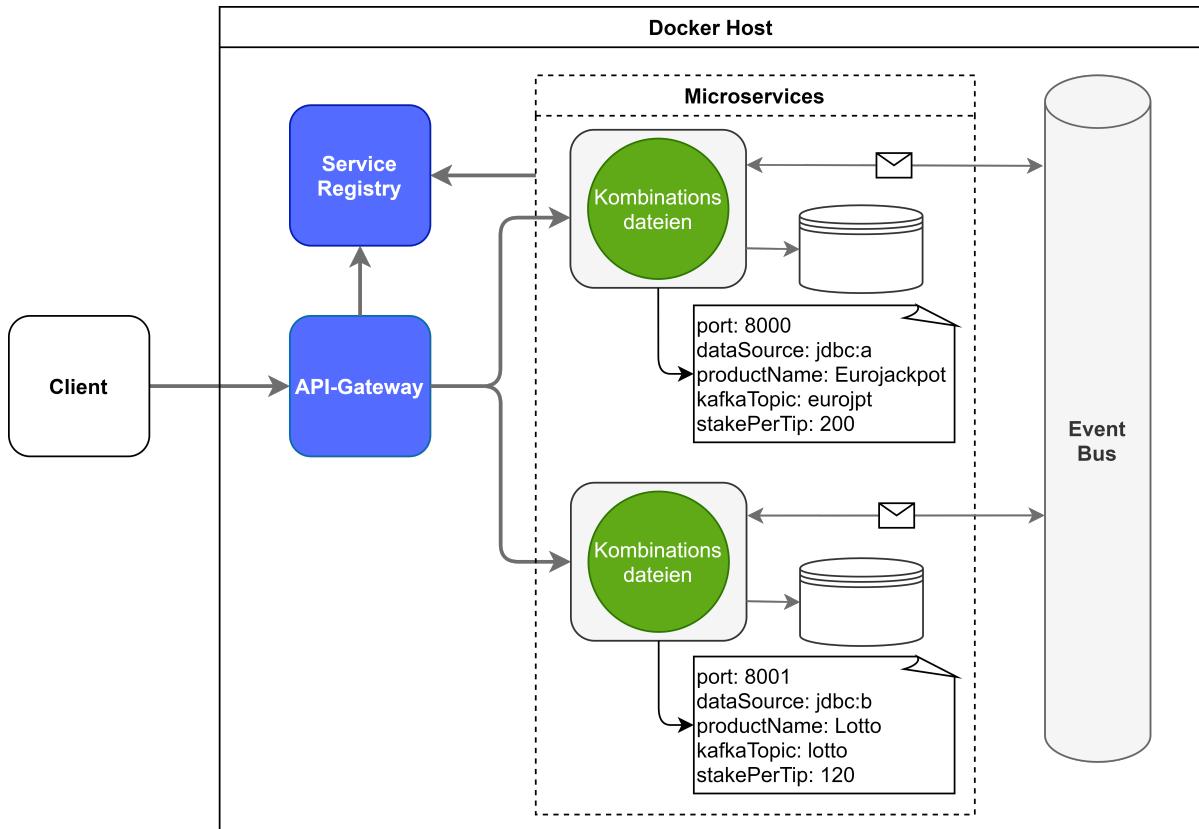


Abbildung 7.8: Microservice-Instanzen für die Verarbeitung von Eurojackpot- sowie Lotto 6aus49 Kombinationsdateien

Abbildung 7.8 zeigt, wie die Architektur erweitert wird, um sowohl Kombinationsdateien für die Spielart Eurojackpot als auch Lotto 6aus49 verarbeiten zu können. Die geringen Unterschiede der Service-Instanzen werden in der extern bereitgestellten Konfiguration modelliert. Dazu gehört etwa die Datenbankverbindung oder der Einsatz pro Tipp, der für die jeweilige Spielart anfällt. Um die Daten verschiedener Spielarten auch innerhalb des Event Buses zu trennen, verwenden beide Instanzen darüber hinaus ein eigenes Topic. Alle Events lassen sich somit eindeutig einer Spielart zuordnen und weitere Services der Architektur sind in der Lage Events zu spezifischen Spielarten zu konsumieren.

Im Falle des zweiten betrachteten Beispiels handelt es sich um den Abrechnungen-Microservice. Die Bewertung der Kriterien Performance, Kosten und Verfügbarkeit entspricht denen des Kombinationsdateien-Services. Der entscheidende Unterschied besteht in dem vorliegenden Grad der Variabilität. Während sich die grundsätzliche Verarbeitung für Eurojackpot und Lotto 6aus49 ähnelt, bestehen im Detail zahlreiche Unterschiede, welche die gesamte Abrechnung beeinflussen. Es ist entsprechend ein sehr hoher Grad der Variabilität vorhanden, der die Verwaltung beider Spielarten in einer Codebasis nicht sinnvoll macht. In diesem Fall bietet es

sich an, die verschiedenen Spielarten in unterschiedlichen Microservices zu verwalten. Dies spricht für eine Umsetzung mittels geteilter Services, Sidecars, geteilter Bibliotheken oder Code-Duplikationen.

Aus Sicht der Wartbarkeit konnte festgestellt werden, dass sich eine Verwaltung von Gemeinsamkeiten in einem separaten Service eignet, um Änderungen lediglich einmalig vornehmen und bereitstellen zu müssen.

Demgegenüber konnte im Rahmen der Fallstudie ein weiterer Aspekt identifiziert werden, welcher im Konzept bisher keine Berücksichtigung gefunden hat: die erwartete Änderungshäufigkeit der Gemeinsamkeiten. Gemeinsamkeiten in einer Codebasis zu kapseln hat den Vorteil, dass Änderungen nur einmalig vorgenommen werden müssen. Im Rahmen von Microservices führt dies allerdings zu einer Kopplung mehrerer Services sowie der Einführung neuer Services, was stets kritisch zu hinterfragen ist. In diesem Fall wurde etwa festgestellt, dass Änderungen an Gemeinsamkeiten der Abrechnung nur äußerst selten zu erwarten sind. Die jeweiligen Abrechnungen der beiden Spielarten sind fest spezifiziert und wurden etwa im Falle von Eurojackpot seit mehr als zehn Jahren nicht verändert. Dies ist ein Argument für die Anwendung einer Code-Duplikation. In diesem Fall müssen eventuell anfallende Änderungen zwar mehrfach angewendet werden, die Wahrscheinlichkeit, dass dies regelmäßig auftritt, wird allerdings als sehr niedrig eingeschätzt.

Unter Berücksichtigung dieses Aspektes wurde letztlich entschieden, den Abrechnungs-Service für die neue Spielart Lotto 6aus49 auf Basis einer Code-Duplikation zu entwickeln. Beide Services sind auf diese Weise komplett eigenständig und maximal entkoppelt. Auch bestehen keine Abhängigkeiten zu weiteren Services. Während dieses Vorgehen dem DRY-Prinzip widerspricht, konnte auch im Rahmen dieser Arbeit bestätigt werden, dass Code-Duplikationen in einer Microservice-Architektur ein valides Mittel sind, um die Unabhängigkeit mehrerer Services zu gewährleisten.

Die auf diese Weise erweiterte Architektur ist in Abbildung 7.9 dargestellt. Im Rahmen der Fallstudie wurden exemplarisch zwei Microservices betrachtet. Das Konzept erlaubt es, in ähnlicher Weise auch für die verbleibenden Services der Architektur eine geeignete Variabilitätsstrategie auszuwählen.

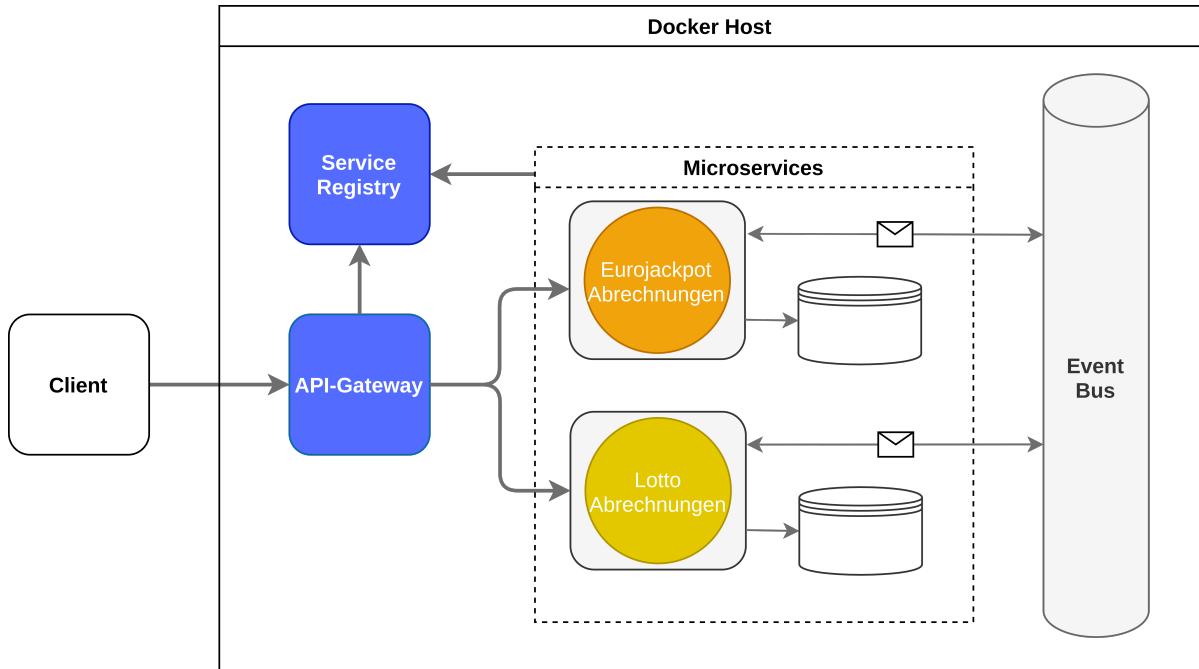


Abbildung 7.9: Separate Microservices für die Verarbeitung von Eurojackpot- sowie Lotto 6aus49 Abrechnungen

7.8 Ergebnisse

Auf Basis des Konzepts wurden Methoden des DDD und modellbasierte Ansätze als geeignete Strategie für die Dekomposition des EJP-CC identifiziert. Als modellbasierter Ansatz ist die Entscheidung - insbesondere aufgrund der eingeschränkten Verfügbarkeit von Tools - auf Service Cutter gefallen. Unter Anwendung der beiden Vorgehensweisen wurden acht Microservices ermittelt. Dabei konnten sowohl das DDD als auch Service Cutter zum Ergebnis der Dekomposition beitragen und als wertvolle Diskussionsgrundlage dienen.

Die Microservices wurden im Rahmen eines Prototyps implementiert und evaluiert. Durch die gewählte Aufteilung der Microservices konnten einige Vorteile erreicht werden. Jeder Microservice übernimmt ausschließlich Aufgaben einer Subdomäne. Stellt eine teilnehmende Gesellschaft etwa eine Kombinationsdatei zur Verfügung, so ist ausschließlich der Kombinationsdateien-Service an der Bearbeitung der Anfrage beteiligt. Ausfälle anderer Services beeinflussen somit nicht diesen Teil der Anwendung. Auf diese Weise wird die Fehlertoleranz des Gesamtsystems erhöht, was insbesondere für eine kritische Anwendung wie das EJP-CC einen großen Vorteil bedeutet.

Für die lose Kopplung der Services sorgt auch der verwendete Event Bus. Alle Services veröffentlichen lediglich Nachrichten und können bei Bedarf die Nachrichten anderer Services konsumieren. Durch die Umsetzung dieser eventbasierten Architektur konnte im Rahmen des Prototyps vollständig auf eine synchrone Kommunikation zwischen Services verzichtet

werden, was sowohl die Fehlertoleranz als auch die Skalierbarkeit erhöht. Außerdem lassen sich neue Services mit wenig Aufwand in die Architektur aufnehmen, ohne dass bestehende Services angepasst werden müssen. Die neuen Microservices werden dazu lediglich an den Event Bus angebunden und können anschließend die Events aller bestehenden Services empfangen und verarbeiten.

Ein weiterer Vorteil, welcher mit der Microservice-Architektur erreicht werden konnte, ist die unabhängige Skalierung einzelner Softwarekomponenten. Statt - wie im Falle einer monolithischen Anwendung - die gesamte Software mehrfach bereitzustellen, können die Services je nach Bedarf unabhängig voneinander skaliert werden, was eine effizientere Ressourcennutzung ermöglicht [4], [40].

Durch die lose Kopplung und die kleinen Codebasen der ermittelten Microservices kann darüber hinaus eine hohe Wartbarkeit gewährleistet werden. Die Services können somit unabhängig voneinander weiterentwickelt und bereitgestellt werden.

Im Rahmen der prototypischen Umsetzung konnte somit gezeigt werden, dass die gewählte Aufteilung der Services die Vorteile einer Microservice-Architektur (vgl. Abschnitt 2.2) erfüllt. Dies spricht für die durch das Konzept ermittelte Migrationsstrategie.

Dennoch bleibt zu beachten, dass es sich bei der Microservice-Architektur um ein verteiltes System handelt, welches entsprechende Herausforderungen mit sich bringt. Während die eigenständige Datenhaltung eines jeden Microservices etwa die Unabhängigkeit der Services erhöht, werden gleichzeitig Redundanzen eingeführt. Durch die verteilte Datenhaltung herrscht außerdem keine strenge Konsistenz, welche das *Atomicity, Consistency, Isolation, Durability* (ACID) Prinzip erfüllt. Stattdessen wird eine schlussendliche Konsistenz (engl. *eventual consistency*) zugesichert. Geschäftsprozesse, welche mehrere Services umspannen, erfordern somit die Einführung geeigneter Vorgehensweisen wie das Saga-Muster [99, S. 114 ff.], um eine anhaltende Konsistenz der Daten zu erreichen. Darüber hinaus müssen in einem verteilten System Herausforderungen in Form von Netzwerklatenzen und potenziellen Ausfällen berücksichtigt werden [31].

Auch mit Hinblick auf die Unterstützung weiterer Spielarten konnte durch die Anwendung des Konzepts eine geeignete Variabilitätsstrategie entwickelt werden. Als Vorteil der Architektur erweist sich hierbei vor allem, dass Microservices eine hohe Wiederverwendbarkeit ermöglichen [2], [4], [28] und für jeden Service eine unterschiedliche Strategie gewählt werden kann. Im Falle des Kombinationsdateien-Services wurde etwa die externe Konfiguration als geeignete Vorgehensweise identifiziert. Der Service benötigt keine tiefgehenden Kenntnisse über die verschiedenen Spielarten und der geringe Grad der Variabilität lässt sich mühelos extern konfigurieren. Der Service kann somit ohne zusätzlichen Implementierungsaufwand in unterschiedlichen Kontexten genutzt werden.

Im Gegensatz dazu zeigt die Fallstudie auch, dass die Einhaltung des DRY-Prinzips in einer

Microservice-Architektur nicht immer sinnvoll ist. Um eine lose Kopplung zu erreichen und die Vorteile der Microservice-Architektur beizubehalten, wurde im Rahmen des Abrechnungs-Services eine Code-Duplikation als sinnvolles Mittel identifiziert.

8 Diskussion

Im Rahmen der folgenden Diskussion werden die gesammelten Ergebnisse dieser Arbeit betrachtet und damit die initial gestellten Forschungsfragen beantwortet. Die Arbeit beschäftigt sich mit verschiedenen Vorgehensweisen für die Migration monolithischer Anwendungen hin zu einer Microservice-Architektur. Zu diesem Zweck wurden verschiedene Techniken und Vorgehensweisen identifiziert und analysiert. Darüber hinaus wurden Möglichkeiten evaluiert, wie sich Variabilität in einer Microservice-Architektur umsetzen lässt, sodass sich eine Software mit moderatem Aufwand in mehreren Kontexten nutzen lässt.

Welche Möglichkeiten und Technologien existieren, um die Migrationen eines Monolithen in eine Microservice-Architektur zu unterstützen?

Kapitel 4 beschreibt die ermittelten Techniken, die in Bezug auf die Dekomposition und das Vorgehensmodell identifiziert werden konnten. Das DDD wird von vielen Autoren für die Modellierung einer Anwendungsdomäne und die Identifizierung geeigneter Microservices propagiert [33, S. 28 ff.], [46, S. 78 ff.], [92], [99, S. 54 ff.] und gilt als der am weitesten verbreitete Ansatz [6], [101].

Da Analyse und Modellierung einer komplexen Domäne allerdings zeitaufwendig sind und dies in der Regel noch manuell vorgenommen wird [17], [57], werden ebenfalls Technologien entwickelt, welche diesen Prozess unterstützen sollen (vgl. Tabelle 6.1). Auf Basis von Design-Artefakten, bestehender Codebasis und dynamischen Laufzeitdaten werden typischerweise Cluster-Algorithmen genutzt, um zusammenhängende Einheiten zu identifizieren.

Insgesamt konnte der von Fritzsch et al. [40] und Li et al. [29] beschriebene Mangel an etablierten Vorgehensweisen und Tools bestätigt werden. Nur wenige Methoden bieten eine ausgereifte Unterstützung von Tools an. Darüber hinaus existieren in der Regel Beschränkungen auf Programmiersprachen und Erwartungshaltungen an genutzte Frameworks. Die Dekomposition ist in der Praxis daher in der Regel noch ein manueller Vorgang, wobei eine Unterstützung durch Tools durchaus erwünscht ist [17]. Ein weiterer Grund für die mangelnde Verbreitung von unterstützenden Technologien ist, dass das Wissen über deren Existenz oft nicht vorhanden ist [57].

Auch in Bezug auf den Prozess wurden mehrere Vorgehensweisen identifiziert. Dazu gehören die Erweiterungsstrategie, das Strangler-Muster und die Neuentwicklung. Im Rahmen der

Erweiterungsstrategie werden lediglich neue Funktionalitäten als Microservices entwickelt. Das Legacy-System bleibt in diesem Fall bestehen und kommuniziert lediglich über eine Schnittstelle mit den neu entwickelten Microservices.

In Kontrast dazu steht die vollständige Neuentwicklung der Anwendung. In diesem Fall besteht keine Verbindung zwischen altem und neuem System. Die neue Microservice-Architektur wird von Grund auf neu aufgebaut und löst den Monolithen im Rahmen eines Cut-over-Prozesses vollständig ab.

Um das Risiko eines Cut-over-Prozesses zu minimieren, sind darüber hinaus iterative Vorgehensweisen, wie das Strangler-Muster verbreitet [58], [61]. In diesem Fall werden Funktionalitäten schrittweise aus dem Monolithen in Microservices extrahiert.

Welche Techniken existieren, um Gemeinsamkeiten und Variabilitäten innerhalb einer Microservice-Architektur zu verwalten?

Im Rahmen des konkreten Projektes ist neben der Verarbeitung von Eurojackpot auch eine Verarbeitung weiterer Spielarten angedacht. In diesem Kontext wurde untersucht, wie sich Gemeinsamkeiten und Unterschiede innerhalb einer Microservice-Architektur verwalten lassen, sodass weitere Spielarten mit moderatem Aufwand ebenfalls verarbeitet werden können. Kapitel 5 beschreibt die identifizierten Vorgehensweisen in Bezug auf die Umsetzung von Variabilität in einer Microservice-Architektur. Um Gemeinsamkeiten und Unterschiede verschiedener Produktvarianten in einer Microservice-Architektur zu verwalten, wurden Feature Flags, Code-Duplikationen, externe Konfigurationen, geteilte Services, das Sidecar-Muster und geteilte Bibliotheken identifiziert und analysiert.

Darüber hinaus werden in Kapitel 5 auch Software-Produktlinien für die Umsetzung von Variabilitäten betrachtet. Im Verlauf der Arbeit wurde diesbezüglich allerdings bewusst eine Einschränkung auf den vorliegenden Anwendungsfall vorgenommen: eine geringe Anzahl von Produkten, welche zur Konzeptionszeit bekannt sind. Eine Ad-hoc-Komposition von Produktvarianten, wie sie etwa im Rahmen von Software-Produktlinien ermöglicht werden soll [72], [150], [151], [152], [153], wurde im Rahmen des Konzeptes bewusst ausgeschlossen. Auch tiefe Anpassungen einer Software zur Laufzeit [154], [155], [156] liegen somit außerhalb des Kontexts dieser Arbeit.

Welche der identifizierten Möglichkeiten können im Falle der EJP-CC Migration sinnvoll eingesetzt werden?

Die reine Identifizierung von Technologien ist nicht ausreichend, um die EJP-CC Migration zu vollziehen. Daher gilt es mithilfe eines Konzeptes eine geeignete Auswahl zu ermitteln. Bestehende Arbeiten, welche sich mit der Dekomposition von Monolithen beschäftigen, geben lediglich einen Überblick über die vorhandenen Methoden [1], [59] oder beschränken sich auf die Anwendbarkeit als einziges Entscheidungskriterium [40], [60]. Die Anwendbarkeit

allein kann aber nur ein Aspekt bei der Bewertung sein, ob ein Dekompositionsansatz sinnvoll eingesetzt werden kann.

Auch in Bezug auf die Prozessstrategien konnte ein Mangel an Entscheidungshilfen festgestellt werden. Während einige Arbeiten eine iterative Vorgehensweise präferieren, um Risiken zu minimieren [107], wird aus der Praxis auch berichtet, dass die besten Erfahrungen im Rahmen von Neuentwicklungen gemacht wurden. Ein Teilnehmer der Studie von Fritzsch et al. [57] gibt dazu an: „Most productive and successful Microservices projects that I know are greenfield developments“.

Daraus lässt sich keine eindeutige Vorgehensweise für bestimmte Szenarien ableiten.

Dies konnte auch im Bereich der Variabilität beobachtet werden. Besonders in diesem Feld konnte ein Mangel an Literatur, welche sich mit der Verwaltung von Gemeinsamkeiten und Unterschieden innerhalb einer Microservice-Architektur auseinandersetzt, festgestellt werden. Lediglich einige wenige Arbeiten beschreiben in der Praxis genutzte Ansätze [22], [23], [70], keine der Arbeiten zeigt jedoch, in welchen Situationen sich welche Technologie anbietet. Es fehlt somit eine detaillierte Betrachtung, welche Vor- sowie Nachteile beschreibt und geeignete Anwendungsfälle aufzeigt. Diese Lücke kann durch die vorliegende Arbeit geschlossen werden.

Das in Kapitel 6 erarbeitete Konzept beschreibt mehrere Bewertungskriterien, anhand derer sich Dekompositions-, Prozess- und Variabilitätsstrategien evaluieren und auswählen lassen. Diese wurden ermittelt, indem untersucht wurde, in welchen Bereichen relevante Unterschiede zwischen den identifizierten Ansätzen bestehen.

In Bezug auf die Dekomposition wurde nach Anwendung des Konzeptes ein Vorgehen mit Methoden des DDD gewählt, welches durch ein modellbasiertes Tool - in diesem Fall Service Cutter - unterstützt wurde. Während in der Praxis teilweise die Meinung vertreten wird, dass Tools einen komplexen Prozess wie die Dekomposition eines Monolithen nicht unterstützen können [57], konnte im Rahmen der Fallstudie das Gegenteil gezeigt werden. Service Cutter hat in diesem Fall auf Basis eines abstrakten ER-Modells mit zusätzlichen Use-Cases nützliche Ergebnisse geliefert, die als Diskussionsgrundlage positiv wahrgenommen wurden. Die Kombination beider Techniken führte zu einer Aufteilung von acht Microservices, welche entkoppelt voneinander lediglich eine Geschäftsfähigkeit ausführen. Die Analyse der Dekompositionsansätze mit anschließender Auswahl hat es ermöglicht, die Dekomposition in einem nachvollziehbaren und strukturierten Prozess zu vollziehen. Dies stellt eine Verbesserung zur Praxis dar, in welcher die Dekomposition noch oft unstrukturiert vorgenommen wird [20].

Ähnliche Erfahrungen konnten im Rahmen der Auswahl einer Prozessstrategie gewonnen werden. Hier konnte gezeigt werden, dass nicht alle Bewertungskriterien immer für den gleichen Ansatz sprechen. Während aus Sicht der Schnittstellenstabilität etwa eine Neuentwicklung sinnvoll ist, spricht die hohe Kritikalität für eine iterative Umsetzung. In diesen Fällen muss

eine Abwägung getroffen werden. Diese erlaubt es jedoch, sich mit Vor- und Nachteilen verschiedener Ansätze zu beschäftigen und somit eine nachvollziehbare Entscheidung zu treffen. Im Falle der EJP-CC Migration konnte so die Neuentwicklung als geeignete Vorgehensweise identifiziert werden.

Für das Hinzufügen weiterer Spielarten konnte in der Fallstudie festgestellt werden, dass die Microservice-Architektur eine hohe Wiederverwendbarkeit ermöglicht. Statt - wie im Falle einer monolithischen Anwendung - eine Variabilitätsstrategie für die gesamte Anwendung zu implementieren, können je nach Microservice unterschiedliche Ansätze gewählt werden. Im Falle des EJP-CC konnte etwa ermittelt werden, dass sich Variabilitäten innerhalb des Kombinationsdateien-Services gut über eine externe Konfiguration modellieren lassen. Der Service lässt sich somit ohne zusätzliche Implementationen in mehreren Kontexten nutzen. Für andere Services können je nach Kontext entsprechend andere Vorgehensweisen gewählt werden. Das Konzept hat auch hier geholfen, die Unterschiede abzuwagen und eine reproduzierbare Entscheidung zu treffen.

Wie kann der Entscheidungsprozess für die Auswahl der identifizierten Technologien unterstützt werden?

Während die durch das Konzept ermittelte Auswahl von Vorgehensweisen zur erfolgreichen Entwicklung des Prototyps geführt hat, konnte die Fallstudie auch zeigen, dass nicht alle Aspekte im Konzept berücksichtigt werden konnten und in der Praxis weitere Faktoren eine Rolle spielen. In Bezug auf die Dekomposition eines Monolithen ist zusätzlich zu den ermittelten Kriterien etwa der Grad der angestrebten Modernisierung ein entscheidender Faktor. Während grundsätzlich alle Softwaremigrationen durch eine Modernisierung motiviert sind [58], existieren dennoch einige Unterschiede im Ausmaß. Die vollständige Abkehr der Kommunikation über E-Mails führt im Falle des EJP-CC etwa zur Vereinfachung vieler Geschäftsprozesse. In diesem Fall sind somit nicht nur Technologien selbst betroffen, auch die fachlichen Abläufe innerhalb der Anwendung erfahren Veränderungen. Dies wiederum erschwert eine Aufteilung der bestehenden Quellcodebasis, da Quellsystem und avisierter Zielarchitektur stark voneinander abweichen und ist somit ein Indiz gegen eine Verwendung von statischer und dynamischer Analyse.

Im Rahmen der Auswahl einer Prozessstrategie wurde festgestellt, dass auch Typ und Anzahl der Schnittstellennutzer eine Rolle spielen. Eine iterative Umstellung der Schnittstelle, welche fortlaufende Änderungen und Zertifizierungstests in den Clients aller Gesellschaften erfordert, verursacht zu viel Aufwand, weshalb eine Neuentwicklung der Anwendung mit einmaligem Cut-over-Prozess beschlossen wurde.

Die Auswahl einer Variabilitätsstrategie hat im Rahmen der Fallstudie gezeigt, dass neben den entwickelten Kriterien ebenfalls die erwartete Änderungshäufigkeit entscheidend ist.

Ist davon auszugehen, dass sich Gemeinsamkeiten selten bis nie ändern werden, so ist die Einführung von geteilten Services oder Bibliotheken nicht zielführend. Sowohl durch geteilte Bibliotheken als auch durch geteilte Services entstehen Abhängigkeiten und Kopplungen zwischen Microservices. Der Mehrwert, Änderungen lediglich einmalig anwenden zu müssen, wiegt die entstehende Kopplung in diesen Fällen nicht auf. Somit konnte diese Arbeit auch Code-Duplikationen als valides Mittel in einer Microservice-Architektur identifizieren.

Die Erkenntnisse der Fallstudie können somit genutzt werden, um das initial erstellte Konzept zu erweitern und weitere Bewertungskriterien zu berücksichtigen. Eine aktualisierte Übersicht der Bewertungskriterien ist in Abbildung 8.1 dargestellt.

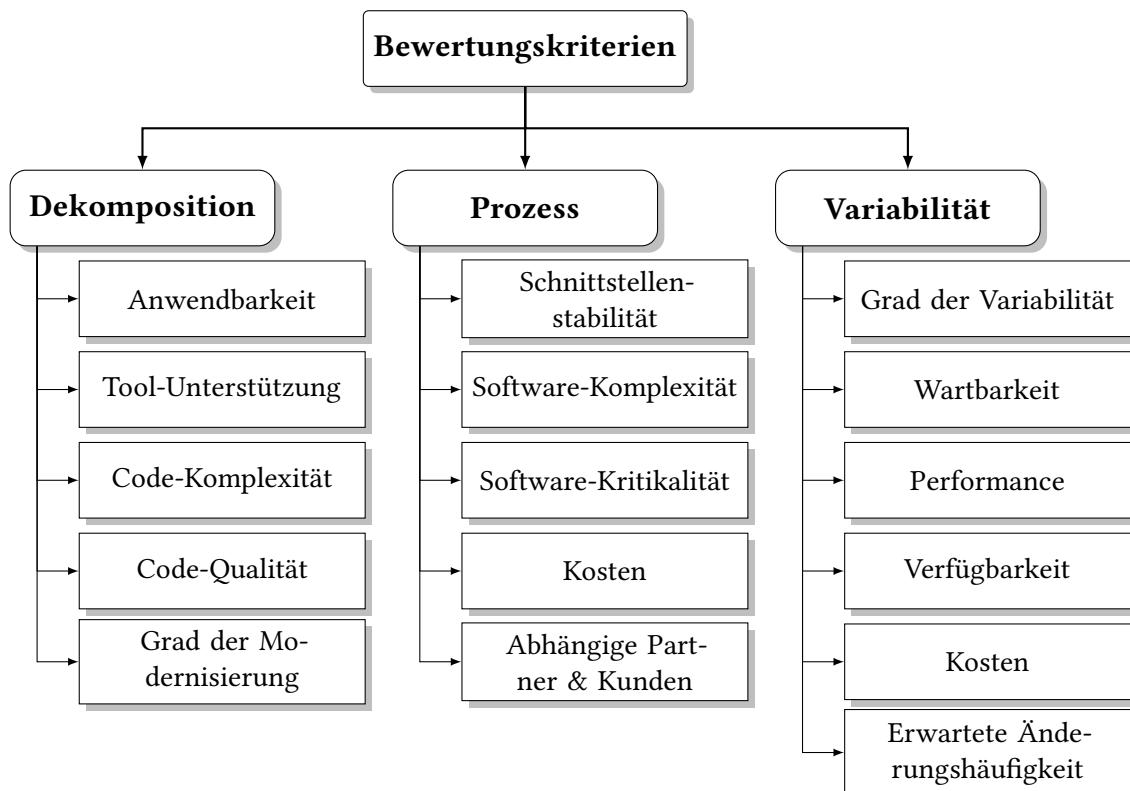


Abbildung 8.1: Bewertungskriterien für die Auswahl von Dekompositions-, Prozess- und Variabilitätsstrategien nach Erkenntnisgewinn der Fallstudie

Während sich das Konzept als geeignetes Mittel für die strukturierte und begründete Auswahl verschiedener Technologien erwiesen hat, lassen sich dennoch Aspekte für die künftige Forschung identifizieren.

Eine denkbare Erweiterung des Konzeptes ist etwa die Entwicklung von Metriken für die entwickelten Bewertungskriterien. Mit definierten Metriken lässt sich leichter bestimmen, unter welchen Voraussetzungen etwa die Qualität oder Komplexität der Anwendung als hoch oder niedrig bewertet wird. Die Einteilung in die einzelnen Kategorien kann somit durch objektive Kriterien unterstützt werden und basiert nicht allein auf der Einschätzung des

Anwenders des Konzepts. Dies erhöht die Nachvollziehbarkeit und Reproduzierbarkeit des Vorgehens.

Ein weiterer Aspekt zukünftiger Forschung kann die Einordnung der identifizierten Variabilitätsstrategien sein. Diese können mit weiteren Techniken wie den Software-Produktlinien oder Methoden der tiefen Anpassbarkeit verglichen werden. Die Einordnung und der Vergleich der verschiedenen Vorgehensweisen kann in der Praxis dabei helfen, abzuschätzen, in welchen Anwendungsfällen sich welche Strategie eignet. Im Rahmen dieser Arbeit wurde bewusst eine Einschränkung auf eine beschränkte Anzahl von bekannten Produktvarianten vorgenommen.

Zuletzt kann in Bezug auf die identifizierten Variabilitätsstrategien ebenfalls untersucht werden, inwiefern sich diese auf eine Mandantenfähigkeit auswirken. Im Rahmen des EJP-CC liegen etwa mehrere Mandanten in Form von Gesellschaften vor. Werden weitere Spielarten wie Lotto 6aus49 in die Architektur aufgenommen, sollen nicht alle Mandanten Zugriff auf die Daten dieser Spielart haben. In Zusammenhang mit einer Mandantenfähigkeit gilt es zusätzliche Herausforderungen zu berücksichtigen. Dazu gehören etwa Datenisolation, Ressourcenverteilung, gesetzliche Richtlinien und Bereitstellungen der Software ohne Ausfallzeiten [141]. Die Auswirkungen dieser Aspekte auf die entwickelten Variabilitätsstrategien liegen außerhalb des Fokus dieser Arbeit und sind somit Bestandteil zukünftiger Forschung.

9 Fazit und Ausblick

Bei der Migration monolithischer Softwaresysteme in eine Microservice-Architektur handelt es sich um einen komplexen Vorgang [1], [14], [15], der in der Praxis oft unstrukturiert vorgenommen wird [20], [40], [101]. Obwohl sich Microservices in der Industrie immer weiter verbreiten, fehlen Richtlinien, welche die Dekomposition unterstützen können.

Um eine Migration strukturiert und nachvollziehbar durchführen zu können, wurden im Rahmen dieser Arbeit mehrere Beiträge geleistet. Zunächst wurde identifiziert, welche Methoden und Technologien existieren, um die Dekomposition eines Monolithen zu unterstützen. Das Wissen über die Existenz unterstützender Tools ist in der Industrie teils nicht vorhanden [40]. In diesem Kontext konnten 19 Arbeiten ermittelt werden, welche sich in die Kategorien einteilen lassen, die auch von Fritzsch et al. [40] und Ponce et al. [1] ermittelt wurden: die statische Analyse, die dynamische Analyse und die modellbasierte Analyse. Darüber hinaus ist auch das Domain-driven Design ein weit verbreiteter Ansatz für die Modellierung von Microservices.

Während die Arbeiten von Fritzsch et al. [40] und Bajaj et al. [60] einen Überblick über existierende Ansätze geben und versuchen den Entscheidungsprozess zu unterstützen, beschränken sich beide Arbeiten lediglich auf die Anwendbarkeit verschiedener Ansätze. Diese Arbeit erweitert die bestehende Forschung, indem mehrere Bewertungskriterien entwickelt wurden, die die Auswahl einer geeigneten Migrationsstrategie zulassen. Neben der Dekomposition wurde in diesem Rahmen auch der Prozess, also die Vorgehensweise im Rahmen der Implementierung, betrachtet. Hier herrscht in der Praxis Uneinigkeit, ob ein iteratives Vorgehen oder eine Neuentwicklung besser geeignet ist. Das vorliegende Konzept hilft dabei, verschiedene Bewertungskriterien zu evaluieren, abzuwagen und eine nachvollziehbare Entscheidung zu treffen. Dies stellt eine Verbesserung gegenüber den in der Praxis oft unstrukturierten Vorgehensweisen dar.

Das entwickelte Konzept konnte im Rahmen einer Fallstudie evaluiert werden und hat zu der strukturierten Ermittlung von acht Microservices beigetragen. Diese wurden im Rahmen eines Prototyps implementiert und sind in der Lage jeweils eine Geschäftsfähigkeit unabhängig zu bearbeiten.

Ein weiterer Beitrag zum Forschungsstand ist die Berücksichtigung von Variabilitäten in einer Microservice-Architektur. Die Studien von Wang et al. [22] und Carvalho et al. [23] bestätigen, dass in der Praxis oft mehrere Produktvarianten unterstützt werden müssen. Auch in diesem Bereich fehlt es allerdings an festen Vorgehensweisen und Richtlinien, wie eine geeignete Technologie für die Verwaltung von Gemeinsamkeiten und Unterschieden in einer Microservice-Architektur ausgewählt werden kann. Diese Arbeit hat im Rahmen des Konzeptes auch bezüglich der Variabilität mehrere Bewertungskriterien entwickeln können, die im Rahmen der Fallstudie ebenfalls evaluiert werden konnten. Auch hier konnte eine strukturierte Vorgehensweise etabliert werden, um weitere Spielarten in die Architektur zu integrieren.

Darüber hinaus dient die Arbeit als Ausgangspunkt für weitere Forschung. Die Evaluation der Bewertungskriterien beruht aktuell auf der subjektiven Einschätzung des Anwenders, der etwa die Komplexität oder Qualität des bestehenden Systems einschätzt. Eine Verbesserung des Konzeptes stellt somit die Entwicklung von Metriken dar, auf deren Basis sich das Konzept objektiver und reproduzierbarer anwenden lässt.

Im Rahmen der Variabilität konnte die Mandantenfähigkeit als wichtiger Aspekt identifiziert werden, welche außerhalb des Kontexts dieser Arbeit liegt. Diesbezüglich lässt sich weiter auswerten, welche Folgen die Herstellung einer Mandantenfähigkeit auf die entwickelten Variabilitätsstrategien und deren Auswahl hat.

Literaturverzeichnis

- [1] F. Ponce, G. Márquez und H. Astudillo, „Migrating from Monolithic Architecture to Microservices: A Rapid Review“, in *2019 38th International Conference of the Chilean Computer Science Society (SCCC)*, 2019, S. 1–7.
- [2] N. Dragoni, S. Giallorenzo, A. L. Lafuente et al., „Microservices: Yesterday, Today, and Tomorrow“, in *Present and Ulterior Software Engineering*, M. Mazzara und B. Meyer, Hrsg., Cham: Springer International Publishing, 2017, S. 195–216.
- [3] M. Fowler und J. Lewis. „Microservices: A Definition of This New Architectural Term“, martinfowler.com. (2014), Adresse: <https://martinfowler.com/articles/microservices.html> (besucht am 05. 10. 2021).
- [4] F. Auer, V. Lenarduzzi, M. Felderer und D. Taibi, „From Monolithic Systems to Microservices: An Assessment Framework“, *Information and Software Technology*, Jg. 137, S. 106 600, 2021.
- [5] Y. Gong, K. Chen, F. Gu und F. Wang, „The Design and Implementation of a Campus Web Information System Based on Micro-Service Architecture“, *Journal of Physics: Conference Series*, Jg. 1629, Nr. 1, S. 012 044, 2020.
- [6] A. Balalaie, A. Heydarnoori und P. Jamshidi, „Migrating to Cloud-Native Architectures Using Microservices: An Experience Report“, in *Advances in Service-Oriented and Cloud Computing*, Ser. Communications in Computer and Information Science, A. Celesti und P. Leitner, Hrsg., Bd. 567, Cham: Springer International Publishing, 2016, S. 201–215.
- [7] A. Balalaie, A. Heydarnoori und P. Jamshidi, „Microservices Architecture Enables DevOps: Migration to a Cloud-Native Architecture“, *IEEE Software*, Jg. 33, Nr. 3, S. 42–52, 2016.
- [8] A. Balalaie, A. Heydarnoori, P. Jamshidi, D. A. Tamburri und T. Lynn, „Microservices Migration Patterns“, *Software-Practice & Experience*, Jg. 48, Nr. 11, S. 2019–2042, 2018.
- [9] P. Di Francesco, I. Malavolta und P. Lago, „Research on Architecting Microservices: Trends, Focus, and Potential for Industrial Adoption“, in *2017 IEEE International Conference on Software Architecture (ICSA)*, Göteborg, Schweden: IEEE, 2017, S. 21–30.
- [10] T. Mauro. „Adopting Microservices at Netflix: Lessons for Architectural Design“, NGINX. (2015), Adresse: <https://www.nginx.com/blog/microservices-at-netflix-architectural-best-practices/> (besucht am 15. 10. 2021).
- [11] P. Calcado. „Building Products at SoundCloud –Part I: Dealing with the Monolith“, Soundcloud. (2014), Adresse: <https://developers.soundcloud.com/blog/building-products-at-soundcloud-part-1-dealing-with-the-monolith> (besucht am 15. 10. 2021).
- [12] S. Ihde und K. Parikh. „From a Monolith to Microservices + REST: The Evolution of LinkedIn’s Service Architecture“, InfoQ. (2015), Adresse: <https://www.infoq.com/presentations/linkedin-microservices-urn/> (besucht am 15. 10. 2021).

- [13] C. Munns, „I Love APIs 2015: Microservices at Amazon“. (2015), Adresse: <https://de.slideshare.net/apigee/i-love-apis-2015-microservices-at-amazon-54487258> (besucht am 15. 10. 2021).
- [14] H. H. S. da Silva, G. de F. Carneiro und M. P. Monteiro, „An Experience Report from the Migration of Legacy Software Systems to Microservice Based Architecture“, in *16th International Conference on Information Technology-New Generations (ITNG 2019)*, Ser. Advances in Intelligent Systems and Computing, S. Latifi, Hrsg., Bd. 800, Cham: Springer International Publishing, 2019, S. 183–189.
- [15] S. Eski und F. Buzluca, „An Automatic Extraction Approach: Transition to Microservices Architecture from Monolithic Application“, in *Proceedings of the 19th International Conference on Agile Software Development: Companion*, Ser. XP ’18, New York, NY, USA: ACM Press, 2018, S. 1–6.
- [16] I. Pigazzini, F. Arcelli Fontana und A. Maggioni, „Tool Support for the Migration to Microservice Architecture: An Industrial Case Study“, in *Software Architecture*, Ser. Lecture Notes in Computer Science, T. Bures, L. Duchien und P. Inverardi, Hrsg., Bd. 11681, Cham: Springer International Publishing, 2019, S. 247–263.
- [17] D. Taibi und K. Systä, „A Decomposition and Metric-Based Evaluation Framework for Microservices“, in *Cloud Computing and Services Science*, Ser. Communications in Computer and Information Science, D. Ferguson, V. Méndez Muñoz, C. Pahl und M. Helfert, Hrsg., Bd. 1218, Cham: Springer International Publishing, 2020, S. 133–149.
- [18] G. Mazlami, J. Cito und P. Leitner, „Extraction of Microservices from Monolithic Software Architectures“, in *2017 IEEE International Conference on Web Services (ICWS)*, 2017, S. 524–531.
- [19] G. Kecskemeti, A. C. Marosi und A. Kertesz, „The ENTICE Approach to Decompose Monolithic Services into Microservices“, in *2016 International Conference on High Performance Computing & Simulation (HPCS)*, Innsbruck, Österreich: IEEE, 2016, S. 591–596.
- [20] C. Schröer, F. Kruse und J. Marx Gómez, „A Qualitative Literature Review on Microservices Identification Approaches“, in *Service-Oriented Computing*, S. Dustdar, Hrsg., Ser. Communications in Computer and Information Science, Cham: Springer International Publishing, 2020, S. 151–168.
- [21] S. Newman, *Building Microservices: Designing Fine-Grained Systems*, 2. Aufl. Sebastopol, CA, USA: O’Reilly Media, 2021, 586 S.
- [22] Y. Wang, H. Kadiyala und J. Rubin, „Promises and Challenges of Microservices: An Exploratory Study“, *Empirical Software Engineering*, Jg. 26, Nr. 4, S. 1–44, 2021.
- [23] L. Carvalho, A. Garcia, W. K. G. Assunção, R. Bonifácio, L. P. Tizzei und T. E. Colanzi, „Extraction of Configurable and Reusable Microservices from Legacy Systems: An Exploratory Study“, in *Proceedings of the 23rd International Systems and Software Product Line Conference - Volume A*, Paris, Frankreich: ACM Press, 2019, S. 26–31.
- [24] A. Megargel, V. Shankararaman und D. K. Walker, „Migrating from Monoliths to Cloud-Based Microservices: A Banking Industry Example“, in *Software Engineering in the Era of Cloud Computing*, Ser. Computer Communications and Networks, M. Ramachandran und Z. Mahmood, Hrsg., Cham: Springer International Publishing, 2020, S. 85–108.

- [25] C. Richardson. „Pattern: Monolithic Architecture“, microservices.io. (2018), Adresse: <http://microservices.io/patterns/monolithic.html> (besucht am 04.02.2022).
- [26] M. Mazzara, N. Dragoni, A. Buccharone, A. Giaretta, S. T. Larsen und S. Dustdar, „Microservices: Migration of a Mission Critical System“, *IEEE Transactions on Services Computing*, Jg. 14, Nr. 5, S. 1464–1477, 2021.
- [27] Y. Wei, Y. Yu, M. Pan und T. Zhang, „A Feature Table Approach to Decomposing Monolithic Applications into Microservices“, in *12th Asia-Pacific Symposium on Internetware*, Singapur: ACM Press, 2020, S. 21–30.
- [28] J.-P. Gouigoux und D. Tamzalit, „From Monolith to Microservices: Lessons Learned on an Industrial Migration to a Web Oriented Architecture“, in *2017 IEEE International Conference on Software Architecture Workshops (ICSAW)*, 2017, S. 62–65.
- [29] S. Li, H. Zhang, Z. Jia et al., „A Dataflow-Driven Approach to Identifying Microservices from Monolithic Applications“, *Journal of Systems and Software*, Jg. 157, S. 110 380, 2019.
- [30] J. Cito, P. Leitner, T. Fritz und H. C. Gall, „The Making of Cloud Applications: An Empirical Study on Software Development for the Cloud“, in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, Ser. ESEC/FSE 2015, New York, NY, USA: ACM Press, 2015, S. 393–403.
- [31] Z. Ren, W. Wang, G. Wu et al., „Migrating Web Applications from Monolithic Structure to Microservices Architecture“, in *Proceedings of the Tenth Asia-Pacific Symposium on Internetware*, Peking, China: ACM Press, 2018, S. 1–10.
- [32] M. E. Conway, „How Do Committees Invent?“, *Datamation*, Jg. 14, Nr. 4, S. 28–31, 1968.
- [33] S. Newman, *Monolith to Microservices: Evolutionary Patterns to Transform Your Monolith*, 1. Aufl. Sebastopol, CA, USA: O'Reilly Media, 2019, 255 S.
- [34] IBM Cloud Team. „SOA vs. Microservices: What's the Difference?“, IBM. (2021), Adresse: <https://www.ibm.com/cloud/blog/soa-vs-microservices> (besucht am 19.09.2021).
- [35] L. Baresi und M. Garriga, „Microservices: The Evolution and Extinction of Web Services?“, in *Microservices: Science and Engineering*, A. Buccharone, N. Dragoni, S. Dustdar et al., Hrsg., Cham: Springer International Publishing, 2020, S. 3–28.
- [36] O. Zimmermann, „Microservices Tenets: Agile Approach to Service Development and Deployment“, *Computer Science - Research and Development*, Jg. 32, Nr. 3, S. 301–310, 2017.
- [37] C. Pautasso, O. Zimmermann, M. Amundsen, J. Lewis und N. Josuttis, „Microservices in Practice, Part 1: Reality Check and Service Design“, *IEEE Software*, Jg. 34, Nr. 1, S. 91–98, 2017.
- [38] M. Fowler. „BoundedContext“, martinfowler.com. (2014), Adresse: <https://martinfowler.com/bliki/BoundedContext.html> (besucht am 16.10.2021).
- [39] E. Evans, *Domain-Driven Design: Tackling Complexity in the Heart of Software*. Boston, MA, USA: Addison-Wesley, 2004, 529 S.

- [40] J. Fritzsch, J. Bogner, A. Zimmermann und S. Wagner, „From Monolith to Microservices: A Classification of Refactoring Approaches“, in *Software Engineering Aspects of Continuous Development and New Paradigms of Software Production and Deployment*, Ser. Lecture Notes in Computer Science, J.-M. Bruel, M. Mazzara und B. Meyer, Hrsg., Bd. 11350, Cham: Springer International Publishing, 2019, S. 128–141.
- [41] N. Kratzke und P.-C. Quint, „Understanding Cloud-Native Applications after 10 Years of Cloud Computing - A Systematic Mapping Study“, *Journal of Systems and Software*, Jg. 126, S. 1–16, 2017.
- [42] D. Merkel, „Docker: Lightweight Linux Containers for Consistent Development and Deployment“, *Linux Journal*, Jg. 2014, Nr. 239, 2014.
- [43] Q. Zhang, L. Liu, C. Pu, Q. Dou, L. Wu und W. Zhou, „A Comparative Study of Containers and Virtual Machines in Big Data Environment“, in *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*, San Francisco, CA, USA: IEEE, 2018, S. 178–185.
- [44] W. Felter, A. Ferreira, R. Rajamony und J. Rubio, „An Updated Performance Comparison of Virtual Machines and Linux Containers“, in *2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, Philadelphia, PA, USA: IEEE, 2015, S. 171–172.
- [45] C. Anderson, „Docker“, *IEEE Software*, Jg. 32, Nr. 3, S. 102–105, 2015.
- [46] M. Bruce und P. A. Pereira, *Microservices in Action*. Shelter Island, NY, USA: Manning Publications, 2019, 366 S.
- [47] Docker, Inc. „Docker Overview“, Docker Documentation. (2021), Adresse: <https://docs.docker.com/get-started/overview/> (besucht am 08. 11. 2021).
- [48] S. Grunert. „Demystifying Containers - Part I: Kernel Space“, Medium. (2019), Adresse: <https://medium.com/@saschagrunert/demystifying-containers-part-i-kernel-space-2c53d6979504> (besucht am 09. 11. 2021).
- [49] Docker, Inc. „Orientation and Setup“, Docker Documentation. (2021), Adresse: <https://docs.docker.com/get-started/> (besucht am 31. 01. 2022).
- [50] B. Burns, B. Grant, D. Oppenheimer, E. Brewer und J. Wilkes, „Borg, Omega, and Kubernetes: Lessons Learned from Three Container-Management Systems over a Decade“, *ACM Queue*, Jg. 14, Nr. 1, S. 70–93, 2016.
- [51] B. Burns, J. Beda und K. Hightower, *Kubernetes: Up and Running: Dive into the Future of Infrastructure*, 2. Aufl. Peking, China: O'Reilly Media, 2019, 255 S.
- [52] J. Spaleta. „How Kubernetes Works“, Cloud Native Computing Foundation. (2019), Adresse: <https://www.cncf.io/blog/2019/08/19/how-kubernetes-works/> (besucht am 10. 11. 2021).
- [53] The Kubernetes Authors. „What Is Kubernetes?“ (2021), Adresse: <https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/> (besucht am 09. 11. 2021).
- [54] A. Krause, C. Zirkelbach, W. Hasselbring, S. Lenga und D. Kroger, „Microservice Decomposition via Static and Dynamic Analysis of the Monolith“, in *2020 IEEE International Conference on Software Architecture Companion (ICSA-C)*, Salvador, Brasilien: IEEE, 2020, S. 9–16.

- [55] H. Zhang, S. Li, Z. Jia, C. Zhong und C. Zhang, „Microservice Architecture in Reality: An Industrial Inquiry“, in *2019 IEEE International Conference on Software Architecture (ICSA)*, 2019, S. 51–60.
- [56] V. Velepucha und P. Flores, „Monoliths to Microservices - Migration Problems and Challenges: A SMS“, in *2021 Second International Conference on Information Systems and Software Technologies (ICI2ST)*, 2021, S. 135–142.
- [57] J. Fritzsch, J. Bogner, S. Wagner und A. Zimmermann, „Microservices Migration in Industry: Intentions, Strategies, and Challenges“, in *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, Cleveland, OH, USA: IEEE, 2019, S. 481–490.
- [58] P. Di Francesco, P. Lago und I. Malavolta, „Migrating Towards Microservice Architectures: An Industrial Survey“, in *2018 IEEE International Conference on Software Architecture (ICSA)*, 2018.
- [59] H. C. da Silva Filho und G. de Figueiredo Carneiro, „Strategies Reported in the Literature to Migrate to Microservices Based Architecture“, in *16th International Conference on Information Technology-New Generations (ITNG 2019)*, Ser. Advances in Intelligent Systems and Computing, S. Latifi, Hrsg., Bd. 800, Cham: Springer International Publishing, 2019, S. 575–580.
- [60] D. Bajaj, U. Bharti, A. Goel und S. C. Gupta, „A Prescriptive Model for Migration to Microservices Based on SDLC Artifacts“, *Journal of Web Engineering*, 2021.
- [61] M. Fowler. „StranglerFigApplication“, [martinfowler.com](https://martinfowler.com/bliki/StranglerFigApplication.html). (2019), Adresse: <https://martinfowler.com/bliki/StranglerFigApplication.html> (besucht am 27. 10. 2021).
- [62] A. Hunt und D. Thomas, *The Pragmatic Programmer: From Journeyman to Master*. Reading, MA, USA: Addison-Wesley, 2000, 321 S.
- [63] J. Stenberg. „Don’t Share Code Between Microservices“, InfoQ. (2015), Adresse: <https://www.infoq.com/news/2015/01/microservices-sharing-code/> (besucht am 04. 09. 2021).
- [64] C. F. „Many Reasons to Not Share Code between Microservices“, Medium. (2017), Adresse: <https://christophef.medium.com/sharing-code-between-microservices-b576221820b2> (besucht am 04. 09. 2021).
- [65] M. Bialecki. „Why Duplication Isn’t Always a Bad Thing in Micro-Services“, Michal Bialecki Blog. (2019), Adresse: <https://www.michalbielecki.com/2019/02/08/why-duplication-isnt-always-a-bad-thing-in-micro-services/> (besucht am 04. 09. 2021).
- [66] E. Wolff, *Microservices: Flexible Software Architecture*. Boston, MA, USA: Addison-Wesley, 2017.
- [67] S. Kaasten. „Reducing Microservice Overhead with Shared Libraries“, CircleCI. (2021), Adresse: <https://circleci.com/blog/reducing-microservice-overhead-with-shared-libraries/> (besucht am 04. 09. 2021).
- [68] M. Richards, *Microservices AntiPatterns and Pitfalls*. Sebastopol, CA, USA: O’Reilly Media, 2016.
- [69] P. Hauer. „Don’t Share Libraries among Microservices“, Philipp Hauer’s Blog. (2020), Adresse: <https://phauer.com/2016/dont-share-libraries-among-microservices/> (besucht am 04. 09. 2021).

- [70] S. S. de Toledo, A. Martini und D. I. K. Sjøberg, „Improving Agility by Managing Shared Libraries in Microservices“, in *Agile Processes in Software Engineering and Extreme Programming – Workshops*, M. Paasivaara und P. Kruchten, Hrsg., Ser. Lecture Notes in Business Information Processing, Cham: Springer International Publishing, 2020, S. 195–202.
- [71] D. Taibi und V. Lenarduzzi, „On the Definition of Microservice Bad Smells“, *IEEE Software*, Jg. 35, Nr. 3, S. 56–62, 2018.
- [72] M. A. Naily, M. R. A. Setyautami, R. Muschevici und A. Azurat, „A Framework for Modelling Variable Microservices as Software Product Lines“, in *Software Engineering and Formal Methods*, A. Cerone und M. Roveri, Hrsg., Ser. Lecture Notes in Computer Science, Cham: Springer International Publishing, 2018, S. 246–261.
- [73] E. B. Johnsen, R. Hähnle, J. Schäfer, R. Schlatte und M. Steffen, „ABS: A Core Language for Abstract Behavioral Specification“, in *Formal Methods for Components and Objects*, Ser. Lecture Notes in Computer Science, B. K. Aichernig, F. S. de Boer und M. M. Bonsangue, Hrsg., Bd. 6957, Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, S. 142–164.
- [74] L. P. Tizzei, M. Nery, V. C. V. B. Segura und R. F. G. Cerqueira, „Using Microservices and Software Product Line Engineering to Support Reuse of Evolving Multi-tenant SaaS“, in *Proceedings of the 21st International Systems and Software Product Line Conference - Volume A*, Sevilla, Spanien: ACM Press, 2017, S. 205–214.
- [75] R. Kazman, S. G. Woods und S. J. Carrière, „Requirements for Integrating Software Architecture and Reengineering Models: CORUM II“, in *Proceedings of the Working Conference on Reverse Engineering (WCRE'98)*, Ser. WCRE '98, IEEE Computer Society, 1998, S. 154.
- [76] R. Heinrich, A. van Hoorn, H. Knoche et al., „Performance Engineering for Microservices: Research Challenges and Directions“, in *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering Companion*, Ser. ICPE '17 Companion, New York, NY, USA: ACM Press, 2017, S. 223–226.
- [77] D. M. Blei, A. Y. Ng und M. I. Jordan, „Latent Dirichlet Allocation“, *The Journal of Machine Learning Research*, Jg. 3, S. 993–1022, 2003.
- [78] I. Saidani, A. Ouni, M. W. Mkaouer und A. Saied, „Towards Automated Microservices Extraction Using Muti-objective Evolutionary Search“, in *Service-Oriented Computing*, Ser. Lecture Notes in Computer Science, S. Yangui, I. Bouassida Rodriguez, K. Drira und Z. Tari, Hrsg., Bd. 11895, Cham: Springer International Publishing, 2019, S. 58–63.
- [79] A. Levcovitz, R. Terra und M. T. Valente. „Towards a Technique for Extracting Microservices from Monolithic Enterprise Systems“. arXiv: 1605.03175 [cs.SE]. (2016), Adresse: <http://arxiv.org/abs/1605.03175> (besucht am 12. 10. 2021).
- [80] M. Brito, J. Cunha und J. Saraiva, „Identification of Microservices from Monolithic Applications through Topic Modelling“, in *Proceedings of the 36th Annual ACM Symposium on Applied Computing*, Ser. SAC '21, New York, NY, USA: ACM Press, 2021, S. 1409–1418.
- [81] O. Al-Debagy und P. Martinek, „A Microservice Decomposition Method Through Using Distributed Representation of Source Code“, *Scalable Computing: Practice and Experience*, Jg. 22, Nr. 1, S. 39–52, 2021.

- [82] K. Deb, S. Agrawal, A. Pratap und T. Meyarivan, „A Fast Elitist Non-dominated Sorting Genetic Algorithm for Multi-objective Optimization: NSGA-II“, in *Parallel Problem Solving from Nature PPSN VI*, Ser. Lecture Notes in Computer Science, M. Schoenauer, K. Deb, G. Rudolph et al., Hrsg., bearb. von G. Goos, J. Hartmanis und J. van Leeuwen, Bd. 1917, Berlin, Heidelberg: Springer Berlin Heidelberg, 2000, S. 849–858.
- [83] A. K. Kalia, J. Xiao, R. Krishna, S. Sinha, M. Vukovic und D. Banerjee, „Mono2Micro: A Practical and Effective Tool for Decomposing Monolithic Java Applications to Microservices“, in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, Athen, Griechenland: ACM Press, 2021, S. 1214–1224.
- [84] D. Taibi und K. Systä, „From Monolithic Systems to Microservices: A Decomposition Framework Based on Process Mining“, in *Proceedings of the 9th International Conference on Cloud Computing and Services Science*, Iraklion, Kreta, Griechenland: SciTePress - Science and Technology Publications, 2019, S. 153–164.
- [85] D. Taibi, V. Lenarduzzi und C. Pahl, „Microservices Anti-patterns: A Taxonomy“, in *Microservices*, A. Bucchiarone, N. Dragoni, S. Dustdar et al., Hrsg., Cham: Springer International Publishing, 2020, S. 111–128.
- [86] W. Jin, T. Liu, Y. Cai, R. Kazman, R. Mo und Q. Zheng, „Service Candidate Identification from Monolithic Systems Based on Execution Traces“, *IEEE Transactions on Software Engineering*, Jg. 47, Nr. 5, S. 987–1007, 2021.
- [87] A. K. Kalia, J. Xiao, C. Lin et al., „Mono2Micro: An AI-based Toolchain for Evolving Monolithic Enterprise Applications to a Microservice Architecture“, in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, Ser. ESEC/FSE 2020, New York, NY, USA: ACM Press, 2020, S. 1606–1610.
- [88] T. Matias, F. F. Correia, J. Fritzsch, J. Bogner, H. S. Ferreira und A. Restivo, „Determining Microservice Boundaries: A Case Study Using Static and Dynamic Software Analysis“, in *Software Architecture*, Ser. Lecture Notes in Computer Science, A. Jansen, I. Malavolta, H. Muccini, I. Ozkaya und O. Zimmermann, Hrsg., Bd. 12292, Cham: Springer International Publishing, 2020, S. 315–332.
- [89] J. B. MacQueen, „Some Methods for Classification and Analysis of MultiVariate Observations“, in *Proc. of the Fifth Berkeley Symposium on Mathematical Statistics and Probability*, L. M. L. Cam und J. Neyman, Hrsg., Bd. 1, University of California Press, 1967, S. 281–297.
- [90] M. Gysel, L. Kölbener, W. Giersche und O. Zimmermann, „Service Cutter: A Systematic Approach to Service Decomposition“, in *Service-Oriented and Cloud Computing*, M. Aiello, E. B. Johnsen, S. Dustdar und I. Georgievski, Hrsg., Ser. Lecture Notes in Computer Science, Cham: Springer International Publishing, 2016, S. 185–200.
- [91] L. Baresi, M. Garriga und A. De Renzis, „Microservices Identification Through Interface Analysis“, in *Service-Oriented and Cloud Computing*, Ser. Lecture Notes in Computer Science, F. De Paoli, S. Schulte und E. Broch Johnsen, Hrsg., Bd. 10465, Cham: Springer International Publishing, 2017, S. 19–33.
- [92] M. Fowler. „DDD_Aggregate“, martinfowler.com. (2013), Adresse: https://martinfowler.com/bliki/DDD_Aggregate.html (besucht am 22. 10. 2021).
- [93] S. van Dongen, „Graph Clustering by Flow Simulation“, University of Utrecht, 2000.

- [94] I. X. Y. Leung, P. Hui, P. Liò und J. Crowcroft, „Towards Real-Time Community Detection in Large Networks“, *Physical Review E*, Jg. 79, Nr. 6, S. 066 107, 2009.
- [95] M. Girvan und M. E. J. Newman, „Community Structure in Social and Biological Networks“, *Proceedings of the National Academy of Sciences of the United States of America*, Jg. 99, Nr. 12, S. 7821–7826, 2002.
- [96] C. Biemann, „Chinese Whispers - an Efficient Graph Clustering Algorithm and Its Application to Natural Language Processing Problems“, in *Proceedings of Textgraphs: The First Workshop on Graph Based Methods for Natural Language Processing*, New York, NY, USA: ACM Press, 2006, S. 73–80.
- [97] P. Kolb, „Experiments on the Difference Between Semantic Similarity and Relatedness“, in *Proceedings of the 17th Nordic Conference of Computational Linguistics (NODALIDA 2009)*, Odense, Dänemark: Northern European Association for Language Technology (NEALT), 2009, S. 81–88.
- [98] J. Ramos, „Using Tf-Idf to Determine Word Relevance in Document Queries“, in *Proceedings of the First Instructional Conference on Machine Learning*, Bd. 242, 2003, S. 29–48.
- [99] C. Richardson, *Microservices Patterns: With Examples in Java*. Shelter Island, NY, USA: Manning Publications, 2019, 490 S.
- [100] M. Fowler. „EvansClassification“, martinfowler.com. (2005), Adresse: <https://martinfowler.com/bliki/EvansClassification.html> (besucht am 23. 10. 2021).
- [101] S. Tyszberowicz, R. Heinrich, B. Liu und Z. Liu, „Identifying Microservices Using Functional Decomposition“, in *Dependable Software Engineering. Theories, Tools, and Applications*, Ser. Lecture Notes in Computer Science, X. Feng, M. Müller-Olm und Z. Yang, Hrsg., Bd. 10998, Cham: Springer International Publishing, 2018, S. 50–65.
- [102] S. C. North, „Drawing Graphs with NEATO“, *NEATO User manual*, Jg. 11, Nr. 1, 2004.
- [103] B. Wu, D. Lawless, J. Bisbal et al., „Legacy Systems Migration-a Method and Its Tool-Kit Framework“, in *Proceedings of Joint 4th International Computer Science Conference and 4th Asia Pacific Software Engineering Conference*, IEEE, 1997, S. 312–320.
- [104] K. Finnigan, *Enterprise Java Microservices*. Shelter Island, NY, USA: Manning Publications, 2019, 253 S.
- [105] A. Megargel, V. Shankararaman und T. P. FAN, „SOA Maturity Influence on Digital Banking Transformation“, *IDRBT Journal of Banking Technology*, Jg. 2, Nr. 2, S. 1, 2018.
- [106] M. Kalske, N. Mäkitalo und T. Mikkonen, „Challenges When Moving from Monolith to Microservice Architecture“, in *Current Trends in Web Engineering*, Ser. Lecture Notes in Computer Science, I. Garrigós und M. Wimmer, Hrsg., Bd. 10544, Cham: Springer International Publishing, 2018, S. 32–47.
- [107] C.-Y. Fan und S.-P. Ma, „Migrating Monolithic Mobile Application to Microservice Architecture: An Experiment Report“, in *2017 IEEE International Conference on AI & Mobile Services (AIMS)*, Honolulu, HI, USA: IEEE, 2017, S. 109–112.
- [108] A. F. A. A. Freire, A. F. Sampaio, L. H. L. Carvalho, O. Medeiros und N. C. Mendonça, „Migrating Production Monolithic Systems to Microservices Using Aspect Oriented Programming“, *Software: Practice and Experience*, Jg. 51, Nr. 6, S. 1280–1307, 2021.

- [109] G. Kiczales, J. Lamping, A. Mendhekar et al., „Aspect-Oriented Programming“, in *ECOOP’97 – Object-Oriented Programming*, M. Akşit und S. Matsuoka, Hrsg., Ser. Lecture Notes in Computer Science, Berlin, Heidelberg: Springer, 1997, S. 220–242.
- [110] Y. Dubinsky, J. Rubin, T. Berger, S. Duszynski, M. Becker und K. Czarnecki, „An Exploratory Study of Cloning in Industrial Software Product Lines“, in *2013 17th European Conference on Software Maintenance and Reengineering*, 2013, S. 25–34.
- [111] Pivotal Software. „Externalized Configuration“, Spring Boot Reference Documentation. (2021), Adresse: <https://docs.spring.io/spring-boot/docs/2.6.0-SNAPSHOT/reference/htmlsingle/#features.external-config> (besucht am 05. 11. 2021).
- [112] R. Krull. „Microservices – Don’t Create Shared Libraries“, Medium. (2017), Adresse: <https://medium.com/standard-bank/microservices-dont-create-shared-libraries-2e803b033552> (besucht am 30. 10. 2021).
- [113] M. Fowler, *Refactoring: Improving the Design of Existing Code*, 2. Aufl. Boston, MA, USA: Addison-Wesley, 2019, 418 S.
- [114] Microsoft. „Sidecar Pattern“, Microsoft Docs. (2021), Adresse: <https://docs.microsoft.com/en-us/azure/architecture/patterns/sidecar> (besucht am 07. 11. 2021).
- [115] B. Burns, *Designing Distributed Systems: Patterns and Paradigms for Scalable, Reliable Services*. Sebastopol, CA, USA: O’Reilly Media, 2018, 149 S.
- [116] P. Clements und L. Northrop, *Software Product Lines: Practices and Patterns*. Boston, MA, USA: Addison-Wesley, 2002, 563 S.
- [117] F. van der Linden, K. Schmid und E. Rommes, *Software Product Lines in Action: The Best Industrial Practice in Product Line Engineering*. Berlin, New York: Springer, 2007, 333 S.
- [118] S. Apel, D. Batory, C. Kästner und G. Saake, *Feature-Oriented Software Product Lines*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013.
- [119] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak und A. S. Peterson, „Feature-Oriented Domain Analysis (FODA) Feasibility Study“, Carnegie-Mellon University Software Engineering Institute, 1990.
- [120] J. Sincero, H. Schirmeier, W. Schröder-Preikschat und O. Spinczyk, „Is the Linux Kernel a Software Product Line“, in *Proceedings of the International Workshop on Open Source Software and Product Lines (SPLC-OSSPL 2007)*, Kyoto, Japan, 2007.
- [121] H. Lackner, M. Thomas, F. Wartenberg und S. Weissleder, „Model-Based Test Design of Product Lines: Raising Test Design to the Product Line Level“, in *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation*, Cleveland, OH, USA: IEEE, 2014, S. 51–60.
- [122] M. Svahnberg, J. van Gurp und J. Bosch, „A Taxonomy of Variability Realization Techniques“, *Software: Practice and Experience*, Jg. 35, Nr. 8, S. 705–754, 2005.
- [123] R. Hähnle, M. Helvensteijn, E. B. Johnsen et al., „HATS Abstract Behavioral Specification: The Architectural View“, in *Formal Methods for Components and Objects*, Ser. Lecture Notes in Computer Science, B. Beckert, F. Damiani, F. S. de Boer und M. M. Bonsangue, Hrsg., Bd. 7542, Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, S. 109–132.

- [124] L. Nunes, N. Santos und A. Rito Silva, „From a Monolith to a Microservices Architecture: An Approach Based on Transactional Contexts“, in *Software Architecture*, T. Bures, L. Duchien und P. Inverardi, Hrsg., Ser. Lecture Notes in Computer Science, Cham: Springer International Publishing, 2019, S. 37–52.
- [125] O. Al-Debagy und P. Martinek, „A New Decomposition Method for Designing Microservices“, *Periodica Polytechnica Electrical Engineering and Computer Science*, Jg. 63, Nr. 4, S. 274–281, 2019.
- [126] M. Gysel, L. Kölbener und O. Zimmermann. „Service Cutter“. (2015), Adresse: <https://servicecutter.github.io/> (besucht am 19. 11. 2021).
- [127] M. Fowler. „DomainDrivenDesign“, martinfowler.com. (2020), Adresse: <https://martinfowler.com/bliki/DomainDrivenDesign.html> (besucht am 23. 10. 2021).
- [128] B. Foote und J. W. Yoder, „Big Ball of Mud“, in *Pattern Languages of Program Design*, N. Harrison, B. Foote und H. Rohnert, Hrsg., Bd. 4, Addison Wesley, 2000, S. 654–692.
- [129] Z. Dehghani. „How to Break a Monolith into Microservices“, martinfowler.com. (2018), Adresse: <https://martinfowler.com/articles/break-monolith-into-microservices.html> (besucht am 20. 11. 2021).
- [130] Tricentis. „Software Fail Watch: 5th Edition“. (2017), Adresse: <https://www.tricentis.com/wp-content/uploads/2019/01/Software-Fails-Watch-5th-edition.pdf>.
- [131] W. Cunningham, „The WyCash Portfolio Management System“, in *Addendum to the Proceedings on Object-oriented Programming Systems, Languages, and Applications (Addendum) - OOPSLA '92*, Vancouver, BC, Kanada: ACM Press, 1992, S. 29–30.
- [132] C. Lilienthal, *Sustainable Software Architecture: Analyze and Reduce Technical Debt*. Heidelberg: dpunkt, 2019, 297 S.
- [133] D. Bell, *Software Engineering for Students*, 4. Aufl. Harlow, England: Addison-Wesley, 2005, 423 S.
- [134] I. Sommerville, *Software Engineering*, 9. Aufl. Boston, MA, USA: Pearson, 2011, 773 S.
- [135] D. Taibi, V. Lenarduzzi und C. Pahl, „Processes, Motivations, and Issues for Migrating to Microservices Architectures: An Empirical Investigation“, *IEEE Cloud Computing*, Jg. 4, Nr. 5, S. 22–32, 2017.
- [136] M. R. A. Setyautami, H. S. Fadhlillah, D. Adianto, I. Affan und A. Azurat, „Variability Management: Re-Engineering Microservices with Delta-Oriented Software Product Lines“, in *Proceedings of the 24th ACM Conference on Systems and Software Product Line: Volume A*, Montreal, QC, Kanada: ACM Press, 2020, S. 1–6.
- [137] E. Ghabach, M. Blay-Fornarino, F. E. Khoury und B. Baz, „Clone-and-Own Software Product Derivation Based on Developer Preferences and Cost Estimation“, in *2018 12th International Conference on Research Challenges in Information Science (RCIS)*, Nantes, Frankreich: IEEE, 2018, S. 1–6.
- [138] J. Krüger und T. Berger, „Activities and Costs of Re-Engineering Cloned Variants into an Integrated Platform“, in *Proceedings of the 14th International Working Conference on Variability Modelling of Software-Intensive Systems*, Magdeburg: ACM Press, 2020, S. 1–10.
- [139] J. Echeverría, F. Pérez, J. I. Panach und C. Cetina, „An Empirical Study of Performance Using Clone & Own and Software Product Lines in an Industrial Context“, *Information and Software Technology*, Jg. 130, S. 106 444, 2021.

- [140] F. H. Vera-Rivera, C. Gaona und H. Astudillo, „Defining and Measuring Microservice Granularity—a Literature Overview“, *PeerJ Computer Science*, Jg. 7, e695, 2021.
- [141] C.-P. Bezemer und A. Zaidman, „Multi-Tenant SaaS Applications: Maintenance Dream or Nightmare?“, in *Proceedings of the Joint ERCIM Workshop on Software Evolution (EVOL) and International Workshop on Principles of Software Evolution (IWPSE)*, Antwerpen, Belgien: ACM Press, 2010, S. 88–92.
- [142] L. Moreau. „Saas Solutions - Multi-Tenant vs Multi-Instance Architectures“, Scaleway Blog. (2020), Adresse: <https://blog.scaleway.com/saas-multi-tenant-vs-multi-instance-architectures/> (besucht am 27. 11. 2021).
- [143] C. J. Guo, W. Sun, Y. Huang, Z. H. Wang und B. Gao, „A Framework for Native Multi-Tenancy Application Development and Management“, in *The 9th IEEE International Conference on E-Commerce Technology and The 4th IEEE International Conference on Enterprise Computing, E-Commerce and E-Services (CEC-EEE 2007)*, Tokio, Japan: IEEE, 2007, S. 551–558.
- [144] T. J. McCabe, „A Complexity Measure“, *IEEE Transactions on Software Engineering*, Jg. SE-2, Nr. 4, S. 308–320, 1976.
- [145] S. Hofer und H. Schwentner, *Domain Storytelling: A Collaborative, Visual, and Agile Way to Build Domain-Driven Software*, 1. Aufl. Boston, MA, USA: Addison-Wesley, 2021.
- [146] A. Brandolini, *Introducing EventStorming*, 1. Aufl. Leanpub, 2018.
- [147] S. Kapferer und O. Zimmermann, „Domain-Driven Architecture Modeling and Rapid Prototyping with Context Mapper“, in *Model-Driven Engineering and Software Development*, Ser. Communications in Computer and Information Science, S. Hammoudi, L. F. Pires und B. Selić, Hrsg., Bd. 1361, Cham: Springer International Publishing, 2021, S. 250–272.
- [148] W.-T. Lee und T.-C. Lu, „Developing a Microservices Software System with Spring Could – A Case Study of Meeting Scheduler“, in *Advances in E-Business Engineering for Ubiquitous Computing*, Ser. Lecture Notes on Data Engineering and Communications Technologies, K.-M. Chao, L. Jiang, O. K. Hussain, S.-P. Ma und X. Fei, Hrsg., Bd. 41, Cham: Springer International Publishing, 2020, S. 128–140.
- [149] A. Furda, C. Fidge, O. Zimmermann, W. Kelly und A. Barros, „Migrating Enterprise Legacy Source Code to Microservices: On Multitenancy, Statefulness, and Data Consistency“, *IEEE Software*, Jg. 35, Nr. 3, S. 63–72, 2018.
- [150] R. Capilla und N. Topaloglu, „Product Lines for Supporting the Composition and Evolution of Service Oriented Applications“, in *Eighth International Workshop on Principles of Software Evolution (IWPSE’05)*, Lissabon, Portugal: IEEE, 2005, S. 53–56.
- [151] S. T. Ruehl und U. Andelfinger, „Applying Software Product Lines to Create Customizable Software-as-a-Service Applications“, in *Proceedings of the 15th International Software Product Line Conference on - SPLC ’11*, München: ACM Press, 2011, S. 1–4.
- [152] M. A. Matar, R. Mizouni und S. Alzahmi, „Towards Software Product Lines Based Cloud Architectures“, in *2014 IEEE International Conference on Cloud Engineering*, Boston, MA, USA: IEEE, 2014, S. 117–126.

- [153] B. Benni, S. Mosser, J.-P. Caissy und Y.-G. Guéhéneuc, „Can Microservice-Based Online-Retailers Be Used as an SPL? A Study of Six Reference Architectures“, in *Proceedings of the 24th ACM Conference on Systems and Software Product Line: Volume A*, New York, NY, USA: ACM Press, 2020, S. 1–6.
- [154] F. Chauvel und A. Solberg, „Using Intrusive Microservices to Enable Deep Customization of Multi-tenant SaaS“, in *2018 11th International Conference on the Quality of Information and Communications Technology (QUATIC)*, Coimbra, Portugal: IEEE, 2018, S. 30–37.
- [155] H. Song, P. H. Nguyen und F. Chauvel, „Using Microservices to Customize Multi-Tenant SaaS: From Intrusive to Non-Intrusive“, in *Joint Post-Proceedings of the First and Second International Conference on Microservices (Microservices 2017/2019)*, L. Cruz-Filipe, S. Giallorenzo, F. Montesi, M. Peressotti, F. Rademacher und S. Sachweh, Hrsg., Ser. OpenAccess Series in Informatics (OASIcs), Bd. 78, Dagstuhl: Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2020, S. 1–18.
- [156] E. T. Nordli, P. H. Nguyen, F. Chauvel und H. Song, „Event-Based Customization of Multi-tenant SaaS Using Microservices“, in *Coordination Models and Languages*, Ser. Lecture Notes in Computer Science, S. Bliudze und L. Bocchi, Hrsg., Bd. 12134, Cham: Springer International Publishing, 2020, S. 171–180.

A Anhang

Priorities	
Cohesiveness Criteria	
Identity & Lifecycle Commonality	M
Semantic Proximity	M
Shared Owner	M
Latency	M
Security Contextuality	M
Compatibility Criteria	
Structural Volatility	XS
Consistency Criticality	XS
Availability Criticality	XS
Content Volatility	XS
Storage Similarity	XS
Security Criticality	XS
Constraints Criteria	
Consistency Constraint	M
Predefined Service Constraint	M
Security Constraint	M

Abbildung A.1: Kopplungskriterien des Tools Service Cutter

Tabelle A.1: Dekompositionsstrategien und vorhandene Tool-Unterstützung

Nr.	Titel	Tool-Unterstützung
1	Service Cutter: A Systematic Approach to Service Decomposition [90]	Ja
2	Microservices Identification Through Interface Analysis [91]	Prototyp
3	A Dataflow-Driven Approach to Identifying Microservices from Monolithic Applications [29]	Nein
4	A Feature Table Approach to Decomposing Monolithic Applications into Microservices [27]	Prototyp
5	A Decomposition and Metric-Based Evaluation Framework for Microservices [17]	Nein
6	From a Monolith to a Microservices Architecture: An Approach Based on Transactional Contexts [124]	Ja
7	A Microservice Decomposition Method Through Using Distributed Representation of Source Code [81]	Nein
8	Microservice Decomposition via Static and Dynamic Analysis of the Monolith [54]	Nein
9	Migrating Web Applications from Monolithic Structure to Microservices Architecture [31]	Nein
10	Tool Support for the Migration to Microservice Architecture: An Industrial Case Study [16]	Ja
11	Determining Microservice Boundaries: A Case Study Using Static and Dynamic Software Analysis [88]	Prototyp
12	Mono2Micro: A Practical and Effective Tool for Decomposing Monolithic Java Applications to Microservices [83]	Ja
13	Identification of Microservices from Monolithic Applications through Topic Modelling [80]	Prototyp
14	An Automatic Extraction Approach: Transition to Microservices Architecture from Monolithic Application [15]	Nein
15	A New Decomposition Method for Designing Microservices [125]	Nein
16	Towards a Technique for Extracting Microservices from Monolithic Enterprise Systems [79]	Nein
17	Extraction of Microservices from Monolithic Software Architectures [18]	Prototyp
18	Service Candidate Identification from Monolithic Systems Based on Execution Traces [86]	Nein
19	Towards Automated Microservices Extraction Using Multi-objective Evolutionary Search [78]	Nein

Tabelle A.2: Einschränkungen von Dekompositionsstrategien mit Tool-Unterstützung

Nr.	Titel	Einschränkungen
1	Service Cutter: A Systematic Approach to Service Decomposition [90]	-
2	Microservices Identification Through Interface Analysis [91]	-
4	A Feature Table Approach to Decomposing Monolithic Applications into Microservices [27]	Manuelle Extraktion von Funktionen und Entitäten notwendig
6	From a Monolith to a Microservices Architecture: An Approach Based on Transactional Contexts [124]	Nur Spring-Boot Anwendungen, die das Fénix Framework und Spring Data nutzen
10	Tool Support for the Migration to Microservice Architecture: An Industrial Case Study [16]	Voller Funktionsumfang nur für Java-EE Anwendungen
11	Determining Microservice Boundaries: A Case Study Using Static and Dynamic Software Analysis [88]	Nur Python-Anwendungen, die das Django-Framework nutzen
12	Mono2Micro: A Practical and Effective Tool for Decomposing Monolithic Java Applications to Microservices [83]	Nur Java-Anwendungen, Bestandteil der IBM WebSphere Hybrid Edition
13	Identification of Microservices from Monolithic Applications through Topic Modelling [80]	Nur Java-Anwendungen
17	Extraction of Microservices from Monolithic Software Architectures [18]	-

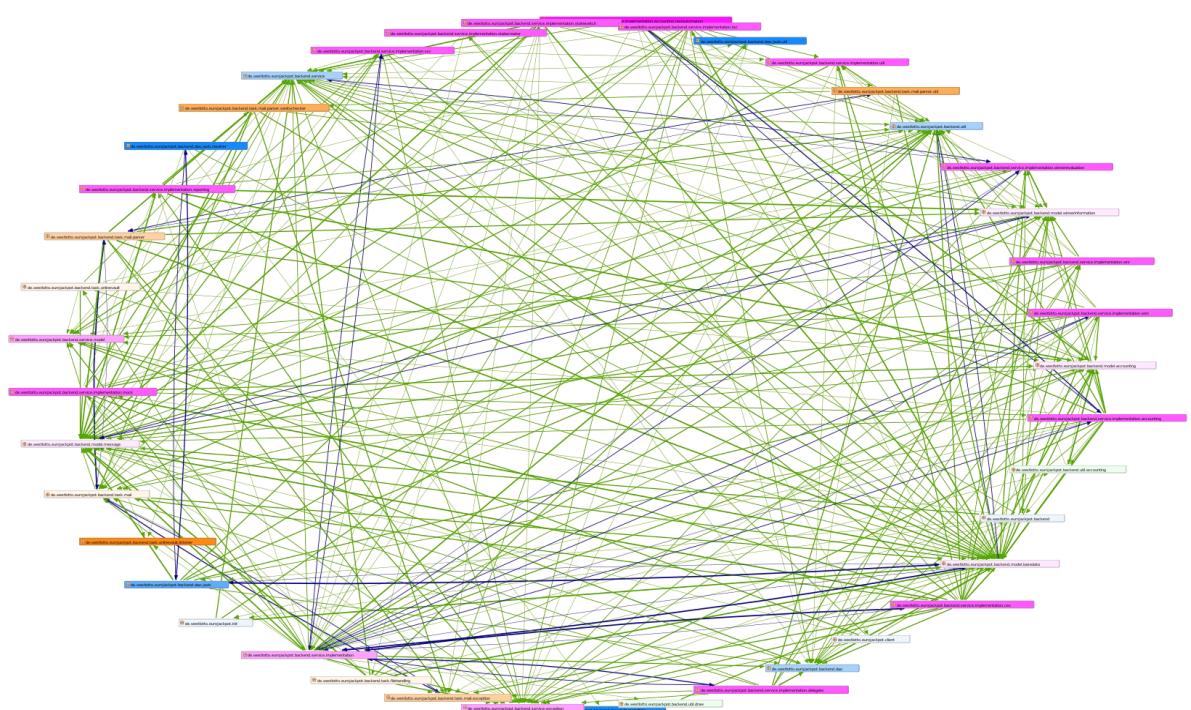


Abbildung A.2: Zyklische Abhangigkeitsgruppe in bestehender Codebasis

Eidesstattliche Erklärung

Ich versichere, die von mir vorgelegte Arbeit selbstständig verfasst zu haben. Alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten oder nicht veröffentlichten Arbeiten anderer entnommen sind, habe ich als entnommen kenntlich gemacht.

Sämtliche Quellen und Hilfsmittel, die ich für die Arbeit benutzt habe, sind angegeben. Die Arbeit hat in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen und ist noch nicht veröffentlicht worden. Ich bin mir bewusst, dass eine unwahre Erklärung rechtliche Folgen haben wird.

Ort, Datum

Unterschrift