Otto-von-Guericke Universität Magdeburg

Fakultät für Informatik

Masterarbeit

Reengineering einer Microservice-Architektur: Eine Fallstudie am PEGASOS-System

Autor:

Johannes Hauffe

24. Januar 2022

Betreuer:

Prof. Dr. rer. nat. habil. Gunter Saake
Otto-von-Guericke Universität Magdeburg

Dr. Jacob Krüger
Ruhr-Universität Bochum

M. Sc. Elias Kuiter
Otto-von-Guericke Universität Magdeburg

Dipl. Inf. Christoph Stepan
NEXUS/MARABU GmbH

Hauffe, Johannes:
Reengineering einer Microservice-Architektur:
Eine Fallstudie am PEGASOS-System
Masterarbeit, Otto-von-Guericke Universität Magdeburg, 2022.

Zusammenfassung

Microservices sind ein neuer Architekturstil, welcher in letzter Zeit immer mehr an Bedeutung gewann, sowohl in der Industrie, als auch im wissenschaftlichen Kontext. Dieser Architekturstil wurde bereits sehr erfolgreich in mehreren Unternehmen eingesetzt und viele Autoren verweisen auf die Erfolge dieses Architekturstils. Insbesondere im Hinblick auf die Skalierbarkeit und Modularität bestehen große Vorteile gegenüber klassischen Architekturstilen wie dem Monolith. Klassische Unternehmen, die einen solchen Monolith besitzen, überlegen, ob sie ihre vorhandene Software auf Microservices migrieren. Allerdings besteht Unwissenheit, spezielle Probleme der Domain oder organisatorische Schwierigkeiten, die eine Migration auf Microservices erschweren. Die Firma NEXUS/MARABU GmbH besitzt mit ihrem Produkt PEGASOS ein monolithisches System im Gesundheitssektor, wobei angedacht wird, dieses in Zukunft auf Microservices zu migrieren.

In dieser Arbeit versuchen wir, anhand der Software PEGASOS, eine Migration auf Microservices durchzuführen, um Probleme, Herausforderungen sowie deren Lösungen zu erörtern. Dazu arbeiten wir, auf der Grundlage einer Literaturrecherche, eine Methodik aus, mit der wir die Migration durchführen. Anhand der technischen Durchführung konnten wir einen Teil der Funktionalität von PEGASOS auf einen Microservice migrieren und stellten als Resultat unsere gewonnenen Erkenntnissen da. Die Erkenntnisse bestehen aus fünf Aspekten. Der erste Aspekt ist, dass die Aufteilung des Monoliths die komplexeste Aufgabe war. Das die Migration in kleinen Schritten erfolgen sollte ist der zweite Aspekt. An dritter Stelle ist das Vorhandensein von praktischen Erfahrungen über Microservices im Entwicklungsteam von Vorteil. Der vierte Aspekt behandelt die Testabdeckung der Software, dass diese zum Nachweis der Integrität, während der Migration, von Vorteil ist. Der letzte Aspekte besagt, dass Abhängigkeiten im Monolith vor der Migration untersucht werden sollte, z.B. im Bezug auf den Datenbankzugriff. Diese Erkenntnisse wurden mit anderen wissenschaftlichen Studien verglichen und es wurden Gemeinsamkeiten festgestellt.

Abschließend wurden Schwachstellen der Arbeit erörtert. Wir haben als Außenstehender ohne praktische Erfahrungen und ohne den Entwickler von NEXUS/MARABU die Migration an dem System vollzogen, wodurch organisatorische Probleme nicht berücksichtigt wurden sind. Außerdem wurde nur ein sehr kleiner Teil der Software migriert, wodurch offen bleibt, welche Probleme bei einer größeren Migration auftreten oder inwieweit unsere Erkenntnisse qualitativ verwertbar sind. Für zukünftige Arbeiten sehen wir diese Arbeit als Grundlage für eine weitere Migration an mit der Überprüfung, inwieweit unsere Erkenntnisse auf andere Systeme übertragbar sind und dort angewendet werden können.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1. Einführung

Am Anfang der Softwareentwicklung verwendeten Entwickler vorrangig eine Architekturform zum Entwickeln: den Monolith. Diese Architekturform besagt, dass die gesamte Funktionalität einer Software in einer einzigen ausführbaren Einheit zusammengefasst ist. Wiederverwendung und Vermeidung von Redundanz waren Kernmerkmale der Softwarearchitektur, sodass vorrangig bestehende Programme erweitert worden sind, anstelle einer Neuentwicklung. Immer neue Anforderungen an die Software führte zu einem immer größeren und komplexeren Wachstum des Monoliths. Auf der einen Seite war dies auch keine schlechte Herangehensweise, dennoch veränderte sich die Situation auf dem Markt, sodass ein Monolith nicht mehr in der Lage war, aktuelle Herausforderungen umzusetzen. Die zu entwickelnden Softwareprodukte wurden immer personalisierter, mussten deutlich mehr Anforderungen von verschiedenen Zielgruppen umsetzen und gleichzeitig sollte die Software kürzere Releasezyklen haben.

In Bezug auf die immer neuen Anforderungen an die Software entwickelten sich neue Softwarearchitekturen, um eine Alternative gegenüber des Monoliths anzubieten. Als Beispiel entwickelte sich der schichtbasierte Monolith, welcher die Funktionalität der Software in einzelne Schichten unterteilt. Eine Schicht übernimmt die Anzeige der Daten oder steuert Nutzerinteraktionen, eine zweite die Auswertung der Daten und eine dritte den Zugriff auf die Datenbank.

Im Zuge der Vernetzung von Computern und Programmen und der Verfügbarkeit von immer mehr Daten und Funktionalitäten im Web, entwickelten sich servicebasierte Architekturen heraus. Die Funktionalitäten sind in diesen Architekturen auf mehrere eigene Services aufgeteilt und über das Netzwerk zur Verfügung gestellt. Solch ein Architekturstil erscheint von Vorteil, weil dadurch die Skalierbarkeit, Robustheit und Zuverlässigkeit verbessert werden kann, die in einem immer mehr wachsenden Monolith schwieriger, aufwendiger und teurer wurde .

Seit ungefähr 2014 ist in dem Kontext ein neuer Begriff für einen Architekturstil im Umlauf, der sich Microservices nennt. Dabei existieren sehr viele kleine Services, wovon jeder eine bestimmte Aufgabe im Netzwerk übernimmt. Die Services kommunizieren, für die Umsetzung einer Funktionalität, über ein Kommunikationsprotokoll

miteinander. Dieser Architekturstil zeichnet sich durch eine sehr hohe Skalierbarkeit, Modularität und Anpassungsfähigkeit für neue Funktionalitäten aus. Firmen wie Netflix und Spotify nutzen mittlerweile sehr erfolgreich das Prinzip der Microservices. Wobei Vorreiter in diesem Gebiet Netflix gilt.

Für Unternehmen stellt sich die Frage, ob sie zu dem Architekturstil der Microservices wechseln sollten und welche Herausforderungen und Chancen mit einer Migration verbunden sind. Es gibt Firmen, die ein Produkt besitzen, welches schon seit über 10 Jahren entwickelt wird und eine monolithische Struktur aufweist. Die NEXUS/-MARABU GmbH ist eine dieser Firmen, die mit ihrem Produkt PEGASOS eine monolithische Software für die digitale Verwaltung von Patientenakten im Gesundheitssektor zur Verfügung stellen und herausfinden möchten, was die technischen Herausforderungen bei der Migration auf Microservices sind.

Zielstellung der Arbeit

Über Microservices und ihrem Aufbau ist mittlerweile viel bekannt. Allerdings fehlt es an praktischen Leitlinien oder an einer allgemeinen Vorgehensweise, wie ein bestimmter Monolith auf Microservices migriert werden kann. An unserem Fallbeispiel PEGASOS führen wir eine Migration auf Microservices durch, um die Probleme und Herausforderungen herauszufinden, die mit solch einer Migration zusammenhängen. Schlussendlich leiten wir Erkenntnisse aus unserer Umsetzung ab, vergleichen diese mit den Erfahrungen aus anderen Fallstudien und geben eine Empfehlung für die Migration von PEGASOS ab.

Gliederung der Arbeit

Im zweiten Kapitel beschreiben wir das System PEGASOS und dessen Domain. Wir beschreiben dabei die Architektur von PEGASOS, dessen Besonderheiten und den Einsatzbereich der Software. Weitergehend widmen wir uns der terminologischen Klärung des Begriffes Microservices, der Beschreibung dieses Architekturstils und den allgemeinen Methodiken der Migration zu Microservices.

Aufbauend auf diesen Grundlagen erarbeiten wir uns in dem dritten Kapitel eine Methodik für unser Vorgehen. Dazu führen wir eine Literaturrecherche durch, mit dessen Ergebnissen wir eine entsprechende Vorgehensweise für die Migration von PEGASOS zu Microservices definieren.

Mithilfe der ausgearbeiteten Methodik erfolgt im vierten Kapitel schließlich die Durchführung der Migration an unserem Fallbeispiel des PEGASOS Systems. Dabei erläutern wir die technische Umsetzung wie wir vorgegangen sind, stellen Probleme und Chancen heraus, die mit der Migration einhergingen, und erklären, wie wir auf Herausforderungen während der Umsetzung reagierten.

Im fünften Kapitel werden anschließend die Resultate der Durchführung ausgewertet. Dabei stellen wir aus unserer Migration gewonnene Erkenntnisse dar und geben einen Ausblick auf die weitere Migration der Software PEGASOS. Zusätzlich erfolgt eine Diskussion möglicher methodischer Probleme und Threats of Validity.

Zum Abschluss fassen wir im sechsten Kapitel die Ergebnisse der Arbeit zusammen und geben einen Ausblick für potentielle zukünftige Forschungsarbeiten.

2. Grundlagen

Der erste Teile dieses Kapitel beschreibt den Begriff Microservices, wie dessen Definition lautet und welche Vor- und Nachteile sich mit dessen Softwarearchitektur ergeben. Die Erläuterung zu grundlegenden Techniken, wie die Migration von einem Monolith zu Microservices durchgeführt werden kann, erfolgt ebenfalls in dem ersten Teil. Die Beschreibung der Domain PEGASOS, dessen Einsatzgebiet sowie dessen Funktionalität wird in dem zweiten Teil dieses Kapitel beschrieben. Dabei gehen wir auf die derzeitige Softwarearchitektur ein, welche Probleme damit verbunden sind und welche Lösungen von Seitens der Firma NEXUS/MARABU, die die Software PEGASOS entwickelt, angestrebt werden.

2.1 Microservices

Der Begriff Microservices erlangte erstmals während eines Workshops für Softwarearchitekten im Mai 2011 eine größere Bedeutung und erschien in einem Vortrag von James Lewis[1] und Fred George[2] Eingang in die Literatur. Dabei wurde das, was Microservices beschreiben, schon länger in der Industrie eingesetzt. Es existierte allerdings keine einheitliche Definition was Microservices überhaupt sind, welche Eigenschaften sie aufweisen und wie die Architektur aufgebaut ist. Martin Fowler und James Lewis [3] definieren Microservices als einen neuen Architekturstil sowie als eine Herangehensweise, wie eine einzelne Applikation programmiert wird, die aus vielen kleinen Services besteht, wovon jeder in einem eigenen Prozess läuft und über einen leichtgewichteten Mechanismus kommuniziert. Die Autoren erwähnen ebenfalls, dass die Services immer in Bezug auf die Businesslogik programmiert sind, dass jeder Service für sich eine bestimmte Aufgabe bzw. einen bestimmten Bereich der Applikation übernimmt. Thones [2015] vertritt in diesem Zusammenhang die Auffassung, dass jeder Service unabhängig von anderen Diensten gebaut, getestet, skaliert und auf einem System ausgeführt werden kann.

[1] http://2012.33degree.org/talk/show/67 (Zuletzt geprüft am 29.06.2021)

[2] https://www.slideshare.net/fredgeorge/micro-service-architecure (Zuletzt geprüft am 29.06.2021)

[3] [Web]https://martinfowler.com/articles/microservices.html (Zuletzt geprüft am 06.10.2021)

Im Vergleich mit bestehenden Architekturmuster fällt eine gewisse Ähnlichkeit mit der serviceorientierten Architektur auf. Diese beschreibt laut dem OASIS SOA-RM Technical Committee [2006] ein Paradigma, welches zur Organisation und Nutzung verteilter Funktionalitäten, die von verschiedenen Besitzern kontrolliert werden, genutzt wird. Bezüglich, dass Microservices einen eigenen Architekturstil bilden oder nur eine Erweiterung der serviceorientierten Architektur sind, existieren verschiedene Meinungen. Martin Fowler und James Lewis[4] erwähnen in ihrem Beitrag Microservice als einen eigenen Architekturstil. Newman [2015] dagegen erwähnt, dass Microservices ein richtiger Ansatz ist um die serviceorientierten Architektur umzusetzen. Adrian Cockroft von Netflix, einer der ersten Firmen die Microservices im großen Maßstab umgesetzt haben, hält dagegen und bezeichnet Microservices als eine fein granulare serviceorientierten Architektur[5]. Spätere Literatur kommt zu dem Schluss, dass sehr viele Prinzipien von Microservices denen von serviceorientierter Architektur ähneln und entsprechend die serviceorientierte Architektur um einzelne Methoden erweitern [Zimmermann, 2017].

2.1.1 Vorteile von Microservices

Um zu verstehen, was die Vorteile von Microservices gegenüber anderen Architekturstilen sind, ist ein Blick auf klassische Architekturstile, wie den Monolithen, notwendig. Laut Newman [2021] spricht man von einem Monolith, wenn „die gesamte Funktionalität eines Systems gemeinsam deployt werden“ [Newman, 2021] muss und sich im Prinzip der gesamte Quellcode des Programms in einem einzelnen Prozess befindet. Die Vorteile und Herausforderungen, die sich daraus ergeben, bilden sich größtenteils aus dem Fakt, dass der gesamte Quellcode ebenfalls an einem Ort ist und sich somit potentiell mehr Entwickler, während der Bearbeitung, in die Quere kommen können. Bird et al. [2011] hat in Studien nachgewiesen, dass je mehr Entwickler an einem Programmabschnitt arbeiten, häufiger Fehler auftreten, als in Abschnitte, wo die Zuständigkeit eindeutiger geregelt ist. Auch können die Module eines Monolithen nicht unabhängig voneinander ausgeführt werden[Dragoni et al., 2017]. Dieses führt zu Problemen, dass ein immer größer werdender Monolith, eine immer geringere Wartbarkeit aufweist, einzelne Module des Monoliths sich schwieriger testen lassen und die Skalierbarkeit des Systems sich nur auf das Gesamtsystem betrachtet durchführen lässt.

Die Vorteile von Microservices lassen sich hauptsächlich von den Eigenschaften ableiten, dass diese klein, eigenständig, unabhängig und auf eine bestimmte Aufgabe spezialisiert sind [Newman, 2015]. Dragoni et al. [2017] ergänzt in diesem Zusammenhang den Begriff Bounded Context, welches von Evans und Fowler [2019], in seinem Buch Domain-Driven-Design, eingeführt wurde. Domain-Driven-Design postuliert, dass ähnliche Funktionalitäten in einen Service zusammengefasst sind. Der gesamte Funktionsumfang einer Applikation ist somit nicht in einem einzelnen Service gebündelt, sondern auf mehrere Services aufgeteilt, wovon jeder eine bestimmte Aufgabe und Funktionalität übernimmt. Die Vorteile, die sich daraus ergeben, sind folgende auf der Grundlage von Newman [2015]:

[4] [Web]https://martinfowler.com/articles/microservices.html
[5] [Web]https://martinfowler.com/articles/microservices.html

Verschiedene Technologien

Jeder Microservice arbeitet für sich eigenständig und unabhängig und ist daher auch nicht an eine bestimmte Technologie gebunden. Somit kann eine Technologie, Programmiersprache oder Framework ausgewählt werden, die für die Aufgabe des Services am geeignetsten erscheint. Eine Allzwecktechnologie, die im Monolith für jegliche Funktionalität eingesetzt werden muss, ist für einzelne Aufgaben der Anwendung zufriedenstellend, für andere wiederum nicht. Bei dem Einsatz von Microservices kann jeder Service eine andere Technologie verwenden und somit die Vorteile einer Programmiersprache oder Frameworks nutzen, die andere nicht mit sich bringen. Außerdem kann schnell auf neue Technologien umgestellt werden, da der Umbau, bedingt durch die kleine Größe eines Microservice, schneller möglich ist als in einem Monolith. Auch die Hürde, diesen Umbau umzusetzen ist geringer.

Belastbarkeit und Skalierung

Microservices bieten eine besondere Gewährleistung, wenn es um die Belastbarkeit und einfache Skalierung von Services geht. Die Eigenständigkeit und Isolierung der Services voneinander folgt aus diesen Eigenschaften. Fällt ein Service des Systems aus, führt dies nicht zu einem Totalausfall des gesamten Systems, weil die anderen Services weiterhin aktiv sind. Im Gegensatz zu einem Monolith, wobei unter Umständen das ganze System nicht mehr reagiert. Ähnlich verhält sich das Prinzip in Bezug auf die Skalierung. Die Skalierung eines Monoliths geschieht durch der mehrmaligen Ausführung des Prozesses, wodurch allerdings Funktionalitäten mitskaliert werden, die Rechenleistung und Speicher belegen, welche sie nicht benötigen. Meistens benötigen nur bestimmte Funktionen einer Software viel Rechenleistung, wodurch die Skalierungsform des Monoliths keine geeignete Strategie ist. Innerhalb der Microservices können einzelne Services mehrmals ausgeführt werden, wodurch einerseits nur die Funktionalität verstärkt wird, die einen höheren Bedarf benötigt und anderseits Ressourcen und Kapazitäten gespart werden.

Komfortables Deployment und Austauschbarkeit

Änderungen an einem Monolith ziehen immer ein komplettes neues Bauen und deployen der gesamten Software mit sich, auch wenn nur wenige Zeilen Code geändert wurden. Änderungen an einer Software betreffen meistens nur einen bestimmten Aufgabenbereich und somit nur einen einzelnen Microservice. Die anderen Services des Systems bleiben von der Änderung unberücksichtigt und können weiterhin aktiv im System bleiben, währenddessen der zu Ändernde Service dem System neu zur Verfügung gestellt wird. Ebenso betrifft dies die Austauschbarkeit eines Services, welcher ohne großen Aufwand, im Vergleich zu einem Monolith, durchgeführt werden kann. Im Monolith ist das Austauschen eines Moduls mit einem erneuten bauen der Software verbunden, wodurch das System für die Zeit nicht zur Verfügung steht. Dagegen kann innerhalb von Microservices ein Services ausgetauscht werden ohne dem Ausfall von Funktionalitäten, die die anderen Services zur Verfügung stellen.

Betriebliche Abstimmung und modularer Aufbau

Eine große und komplexe Software benötigt eine gute Organisation, insbesondere wenn ein großes Entwicklerteam und eine große Codebasis vorhanden sind. Microservices

können dabei helfen, die Unternehmensstruktur auf die Entwicklung der Software anzupassen, indem Entwicklerteams auf die Services dahingehend aufgeteilt werden, dass sie die größtmögliche Produktivität erreichen. Darüber hinaus ermöglicht der modulare Aufbau der Microservice, einen Einsatz für die verschiedensten Aufgaben, sodass dahingehend nicht für ein bestimmtes Betriebssystem oder Gerät entwickelt wird, sondern die Codebasis für diverse Endgeräte wiederverwendet werden kann.

Weitere Vorteile von Microservices

Die vorher genannten Vorteile stammen von Newman [2015] und finden sich auch in der Literatur von anderen Autoren wieder. Zhang [2017] erwähnt den Vorteil des modularen Aufbaus, die unabhängige Entwicklung und den Einsatz von verschiedenen Technologie, die bei der Entwicklung von Microservices zutrage kommen, sowie des schnelleren Deployments und Skalierbarkeit. Ebenso ist die Skalierbarkeit heutzutage ein wichtiger Faktor, warum Microservices immer mehr Anwendung finden [Dragoni et al., 2018]. Ebenfalls lösen Microservices viele Probleme, die mit den immer mehr wachsenden Monolithen aufkamen [Dragoni et al., 2017, 2018]. In der Industrie und Literatur finden sich ebenfalls die genannten Vorteile wieder, unter anderem in Bezug auf die Eigenständigkeit, die Flexibilität, die lose Kopplung und der technologischen Freiheit von Microservices [Soldani et al., 2018]. Darüber hinaus erwähnt Soldani et al. [2018], dass die Fehlertoleranz, dass Nutzen einer separaten Datenbank pro Service und der Einsatz von Continuous Integration und Continuous Development mit Microservice ein Vorteil sind.

2.1.2 Nachteile von Microservices

Microservices bieten nicht nur Vorteile für die Software und das Unternehmen sondern können die Entwicklung einschränken. Newman [2021] erwähnt Situationen in denen Microservices eine schlechte Idee sind. Die erste Situation ist, wenn die Domain des Kunden sehr unklar ist und keine Erfahrungen und Erkenntnisse darüber vorhanden sind. Dies führt zu Systemgrenzen zwischen den Microservices, die nicht der Realität entsprechen und somit im Laufe des Entwicklungsprozesses wieder geändert werden müssen. Das hat wiederum erhöhte Entwicklungskosten zur Folge, um die Änderungen an den Systemgrenzen zwischen den Microservices aufzulösen.

Eine zweite Situation ist, wenn der administrative Prozess der Software beim Kunden oder hauptsächlich beim Kunden stattfindet. Der Einsatz von Microservices erhöht die Komplexität in dem operativen Betrieb der Software [Newman, 2021]. Die Software besteht dann nicht mehr aus einer einzigen ausführbaren Datei oder einem einzelnen Installationsprogramm sondern aus vielen kleinen Paketen. Diese Pakete müssen unter Umständen in der richtigen Reihenfolge installiert werden und aufeinander zugreifen können. Wenn absehbar ist, dass die Kunden nicht die Expertise oder nicht die nötige Unterstützung zur Installation bekommen, dann sollte von Microservices abgesehen werden.

2.1.3 Migrationstechniken

Die Migration eines Monoliths auf Microservices ist eine komplexe Angelegenheit und nicht jede Methodik lässt sich auf jeden Fall anwenden. Dennoch existieren

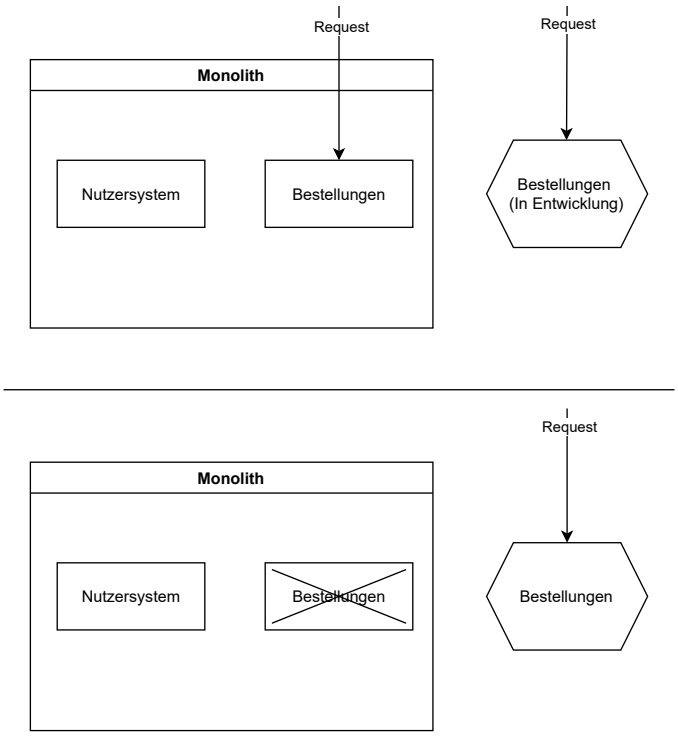

Abbildung 2.1: Die neue Implementierung der Bestellungen werden parallel zu alten Implementierung beschrieben. Erst nachdem die neue Implementierung den gesamten Funktionsumfang besitzt und die Anfragen der Clients auf den Service umgeleitet sind, kann die alte Implementierung im Monolith entfernt werden.

Vorgehensweise und Techniken, die sich bei einer Migration als nützlich erwiesen haben [Carrasco et al., 2018]. Eine bekannte Technik beschreibt Newman [2021] und nennt sich Strangler-Fig-Application. Die ursprüngliche Idee dieser Technik beschrieb zuerst Martin Fowler[6] und besagt, dass eine neue Implementierung zuerst von der alten Funktionalität unterstützt wird, bis zu dem Punkt, wo die neue Implementierung die alte Funktionalität nicht mehr als Stütze benötigt. In der Abbildung 2.1 soll die Funktionalität des Moduls *Bestellungen* in einen neuen Service implementiert werden, welcher sich nicht im Monolith befindet. Während der Migration ist die Implementierung zweimal im Netzwerk vorhanden, einerseits im Monolith und anderseits in dem separaten Service. Die Nutzer nutzen vorrangig die Funktionalität des Monoliths, solange die Implementierung des neuen Services sich noch in der Entwicklung befindet. Einzelne Aufrufe an den Monolith können dabei testweise an den neuen Service geschickt werden, um die neue Implementierung zu testen. Erst nachdem die gesamte Funktionalität im neuen Service getestet und bereitgestellt ist, wird die alte Funktionalität im Monolith entfernt, da diese nicht mehr verwendet wird.

Eine weitere Technik, die sich bei einer Migration anbietet, ist *Branch by Abstraction*. Diese wurde, in dem Kontext von Microservices, von Newman [2021] beschrieben und

[6] https://martinfowler.com/bliki/StranglerFigApplication.html

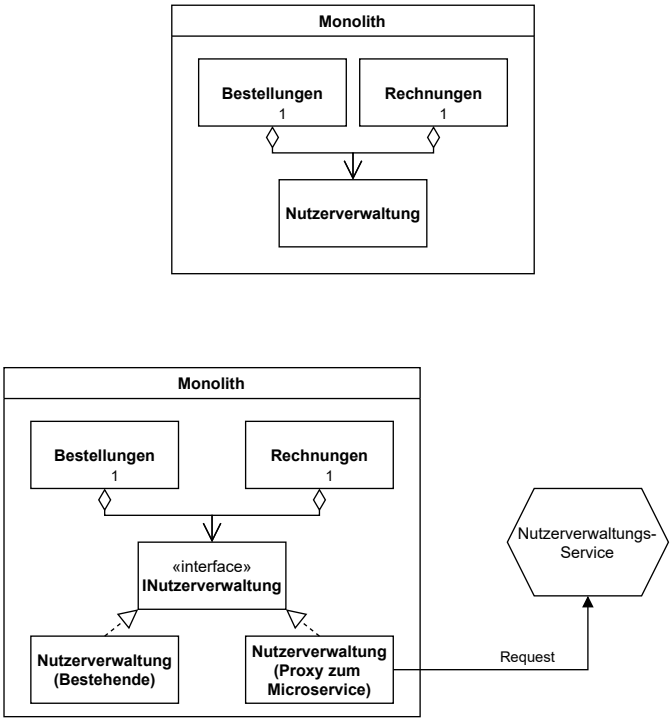

Abbildung 2.2: Durch die Einführung einer Abstraktionsschicht der Nutzerverwaltung können zwei Implementierungen im System vorhanden sein. Durch die Abstraktionsschicht ist es möglich, inkrementelle Veränderungen an der Software vorzunehmen ohne das die Abhängigkeiten im Monolith geändert werden müssen.

von Martin Fowler[7] 2014 erstmals formuliert. Diese Technik eignet sich für Module, die innerhalb des Monoliths enger gekoppelt sind und durch ihre Abhängigkeit, schwerer zu extrahieren sind. In der Abbildung 2.2 ist die Technik an einem Bestellsystem dargestellt. In dem Monolith soll die Nutzerverwaltung auf einen Microservice extrahiert werden, welche allerdings in Abhängigkeit zu dem Bestellsystem und den Rechnungen steht. Die Technik *Branch by Abstraction* besagt in dieser Struktur, dass zuerst eine Abstraktionsschicht eingeführt und anschließend zwei Implementierungen vorhanden sind. Eine Implementierung davon besitzt die Funktionalitäten der Nutzerverwaltung, während die zweite Implementierung die Anfragen an einen Microservice weiterleitet. Nun können die Funktionalitäten inkrementell von der bestehenden Implementierung in den Microservice extrahiert werden. Nach Abschluss der Extraktion muss nur die neue Implementierung im Monolith genutzt werden und die bestehende Funktionalität kann gelöscht werden, da diese im Microservice implementiert wurde. Diese Technik ermöglicht weitere Methodiken wie *Feature-Toggles*[8] oder *Parallel-Run*[9].

[7] https://martinfowler.com/bliki/BranchByAbstraction.html (Zuletzt geprüft am 23.01.2022)
[8] https://martinfowler.com/articles/feature-toggles.html (Zuletzt geprüft am 23.01.2022)
[9] https://engineering.zalando.com/posts/2021/11/parallel-run.html (Zuletzt geprüft am 23.01.2022)

2.2 Beschreibung der Domain PEGASOS

Die Durchführung dieser Fallstudie wird an der Software PEGASOS, welche die Firma NEXUS/MARABU entwickelt, durchgeführt. PEGASOS wurde mit der Idee konzipiert „sämtliche medizinischen und administrativen Informationen im Krankenhaus auf einer einheitlichen Plattform zusammenzuführen“[10]. Die Software dient als Enterprise Content Management-Plattform (ECM) zur digitalen Verwaltung von Daten und Dokumenten und bietet eine einheitliche Oberfläche, um „ein zeit- und standortunabhängiges Zusammenarbeiten von Mitarbeitern“[11] zu ermöglichen. Im Zuge der Digitalisierung des Gesundheitssektors bietet PEGASOS die Möglichkeit, insbesondere in Krankenhäusern und Kliniken, den medizinischen Anforderungen Genüge zu tun. Der Fokus liegt auf Standardisierungen, Informationsbereitstellung, Informationssicherheit, Datenschutz und Transparenz.

Die Software PEGASOS deckt in ihrem Funktionsumfang die vier wichtigsten Teile eines ECM Systems ab: Erfassen, Verwalten, Speichern und Bereitstellen von Informationen[12]. Beim Erfassen von Dokumenten bietet die Software einen zentralen sowie dezentralen Scanprozess an, eine Bilderfassungs-App, unterstützt digitale Signaturen und die Erkennung von Formularen, Bar- oder QR-Codes. Ebenso ist der Import von allen relevanten Datenformaten möglich und eine automatische Klassifizierung und Indexierung kann vorgenommen werden.

Die vorhandenen Daten können durch die integrierte Aktenverwaltung abgerufen, angelegt, verändert oder externen Systemen zur Verfügung gestellt werden[13]. Zusätzlich sind Notizen und Annotationen möglich, eine Office-Integration ist vorhanden und der Verwaltungsprozess unterstützt eine Version- und Revisionierung. Die Archivierung der Daten erfolgt gemäß der IDW PS 880, womit dem Kunden garantiert wird, dass „die elektronische Archivierung mit dem PEGASOS ECM-System den geltenden handels- und steuerrechtlichen Ordnungsmäßigkeitskriterien entspricht“[14]. Der Standard der IHE-Repository und IHE-Registry wird in dem ganzen Prozess ebenfalls unterstützt.

PEGASOS dient als zentrale Bereitstellungsplattform von Daten, in Form eines Windows- oder Webclient oder ist in das Krankenhausinformationssystem (KIS) mittels eines Plugins eingebunden. Über dem Client oder Plugin können die Nutzer auf die Daten zugreifen, nach Metadaten, Volltexten oder Notizen suchen sowie Zugriffsteuerungen und Protokollierungen vornehmen. Die Einbettung in bekannte Krankenhaussystem und Standards wie das IHE[15] oder das FHIR[16] ist gegeben.

[10] https://www.nexus-marabu.de/konzept.html (Zuletzt geprüft am 11.11.2021)
[11] https://www.nexus-marabu.de/konzept.html (Zuletzt geprüft am 11.11.2021)
[12] https://www.aiim.org/Resources/Glossary/Enterprise-Content-Management (Zuletzt geprüft am 11.11.2021)
[13] https://www.nexus-marabu.de/aktenverwaltung.html (Zuletzt geprüft am 11.11.2021)
[14] https://www.nexus-marabu.de/nachricht/pegasos-erhaelt-bescheinigung-fuer-idw-ps-880-konformitaet-638.html (Zuletzt geprüft am 11.11.2021)
[15] https://www.ihe-d.de/ (Zuletzt geprüft am 12.11.2021)
[16] https://hl7.org/FHIR/ (Zuletzt geprüft am 12.11.2021)

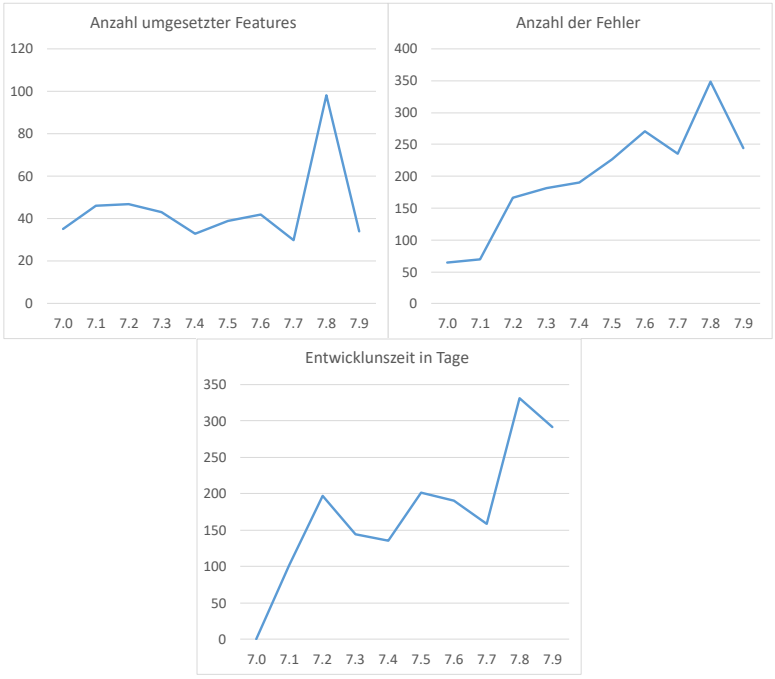

Abbildung 2.3: Der erste Graph zeigt die Anzahl der neuen Features, die in der jeweiligen Version enthalten waren. Die Anzahl der Fehler für jede Version sind in dem zweiten Graph visualisiert und letzte Graph zeigt die Anzahl der Tage an, wie lange die Entwicklungszeit gedauert hat.

2.2.1 Bisherige Architektur und Probleme mit PEGASOS

Auf der Softwarearchitekturebene ähnelt PEGASOS einer klassischen Client-Server Architektur[17], mit den Komponenten Client, Server und Datenbank. Der Server ist ein auf Java basierter Wildlfy Server, der die Logik der Dokumentenverwaltung beinhaltet. Hinter dem Server wird für das Storage-Management eine Datenbank zur Ablage von Objekten verwendet, die entweder Oracle oder Microsoft SQL Server als Technologien verwenden. Zur Ansicht der Daten stehen der Windowsclient, welcher mit der .NET Technologie entwickelt wird, ein Webclient, der sich noch in der Entwicklung befindet und die Plugins, welche in Office-Produkten, wie Microsoft-Word, integriert sind.

Damit wir die Probleme mit dem derzeitigen Entwicklungsprozess besser verstehen können untersuchten wir diesen. Die Entwicklung und Freigabe von neuen Versionen von PEGASOS geschieht anhand fester Versionen zu bestimmten Zeitpunkten. Die Zeitpunkte sind im Voraus geplant, die Freigabe einer neuen Version erfolgt erst, wenn alle Fehler behoben sind und die geplanten Features von der Qualitätssicherung überprüft wurden sind.

Während der Entwicklung einer neuen Version sind generell immer alle Komponenten, Server, Client und Datenbank, von PEGASOS betroffen. Zu jeder neuen Version werden Fehler gefixt und neue Features eingebaut. Stellt man die Entwicklungszeit der Versionen der letzten fünf Jahre, den gelösten Fehler und neu eingebauten Features gegenüber, so lässt sich folgendes erkennen. Die Entwicklungszeit ist im Mittel seit der Version 7.0.0 angestiegen, wobei sie zwischen den Versionen 7.2.0 und 7.7.0

[17] https://www.w3schools.in/what-is-client-server-architecture/ (Zuletzt geprüft am 21.01.2022)

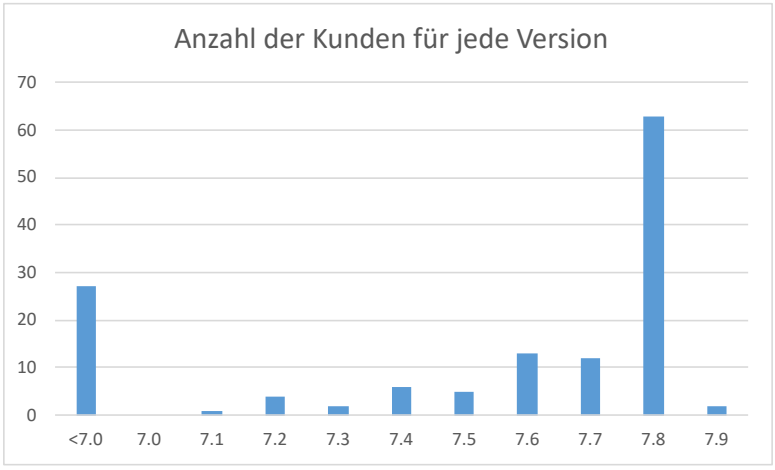

Abbildung 2.4: Die am häufigsten genutzte Version ist die 7.8 mit knapp 50% der Kunden, die zweithäufigste sind Versionen die vor der 7.0 entwickelt wurden sind mit 20% der Kunden und weitere 20% der Kunden nutzen die Versionen 7.7 oder 7.6. Die verbleibenden verteilen sich auf die restlichen Versionen von PEGASOS.

weitgehend um einen Median von 160 Tagen blieb und ab der Version 7.8.0 deutlich auf bis 300 Tage angestiegen ist. Die Anzahl der neuen Features in jeder Version ist weitgehend um die 40 konstant geblieben, wobei die Version 7.8.0 hierbei eine Ausnahme bildet. Bei den gelösten Fehler ist ein Anstieg erkennbar, dass mit jeder neuen Version immer mehr Fehler gelöst werden mussten. Wenn wir die Fehler und Features in Relation zu der Entwicklungszeit setzen, dann stellen wir fest, dass im Durchschnitt gesehen für jedes Feature immer mehr Zeit benötigt wird. In der Version 7.1.0 hat jedes Feature ungefähr zwei Tage benötigt, wogegen in der Version 7.9.0 schon fast neun Tage notwendig sind. Bei den eingebauten Fehler pro Entwicklungstag bis Version 7.7.0 ein Abwärtstrend erkennbar, dass bis dahin weniger Fehler pro Tag eingebaut worden. Ab der Version 7.8.0 ist der Trend wieder umgekehrt, dass ab da mehr Fehler pro Tag gefixt werden mussten.

Zusammenfassend wird aus diesen Statistiken erkennbar, dass immer mehr Zeit für jedes Release benötigt wird, die eingebauten Features konstant bleiben und mehr Fehler vorhanden sind. Dieser Trend wirkt sich längerfristig negativ auf die benötigten Kosten für ein Release aus, dem die Firma NEXUS / MARABU entgegen wirken möchte.

2.2.2 Problematik Kundendomäne

Die Kunden, die PEGASOS einsetzen und nutzen, sind größtenteils Krankenhäuser und Kliniken. Aus den Administrationsinformation der Firma ist erkennbar, dass diese nicht sehr häufig auf eine aktuelle Version aktualisiert werden. Ein Jahr nach dem Release der Version 7.8.0 nutzen gerade einmal knapp 50% der Kunden diese Version. Weitere 20% nutzen eine Version die maximal zwei Jahre alt ist. Versionen die maximal fünf Jahre alt sind bei 10% Kunden im Einsatz und die fehlenden 20% benutzen Versionen, die älter als fünf Jahre sind.

Die Gründe dafür sind vielseitig und sind nicht immer auf alle Kunden anwendbar. Einerseits ist ein Update auf eine neue Version von Seiten des Krankenhauses mit

hohem Aufwand verbunden, sodass dieser Schritt nur durchgeführt wird, wenn er absolut notwendig ist. Anderseits liegt die Komplexität auch auf Seiten der Software PEGASOS, dass die Datenmigration der Datenbank, Umstellung des Servers und Austauschen der Clients mit hohem Aufwand verbunden ist. Letztlich muss bei einem Update die Datenintegrität gesichert sein und der laufende Ablauf des Krankenhauses darf nicht gestört werden, sodass hier allgemein vorsichtig mit einem Update umgegangen wird.

Durch aufkommende Anforderungen sorgt dies zu der Problematik, dass auch sehr viele alte Versionen einen Patch benötigen, wenn ein kritischer Bug gefunden wird, neue rechtliche Anforderungen nicht erfüllt werden oder die Integration mit einem Fremdsystem nicht mehr funktioniert. Der Aufwand für die immer älter werdenden Versionen Patches nachzuliefern, kostet Personal- und Entwicklungsressourcen. Mitunter müssen die Entwickler sich in alten Quellcode einlesen, um die Anforderungen umzusetzen. Hinzu kommt immer die Gefahr, durch eine Änderung an der alten Version wieder neue Fehler einzubauen.

2.2.3 PEGASOS als geeignetes Fallbeispiel

Durch die Feststellung, dass immer mehr verschiedene Versionen beim Kunden gewartet werden müssen, dass die Entwicklungszeit sich für jedes Release verlängert und dass mehr Fehler pro Release entstehen, wurden in der Firma NEXUS/MARABU mehrere Lösungsversuche diskutiert. Eine Idee, die in diesem Prozess aufkam, ist der Einsatz von Microservices, wodurch man sich eine Verbesserung erhoffen könnte. Allerdings fehlt eine Grundexpertise und Analyse über die Fragen, inwieweit Microservices auf das Produkt PEGASOS anwendbar ist, ob Microservices die Probleme von PEGASOS im Kundeneinsatz lösen und wie überhaupt die Software des Monolithen auf Microservices umgebaut werden kann.

Unabhängig von dem Wunsch des Unternehmens Microservices einzusetzen ist PEGASOS ein geeignetes System, für das Herausfinden von Problemen und Herausforderungen, die mit einer Migration auf Microservices einhergehen. PEGASOS wird seit über 20 Jahren entwickelt und entstand somit in einer Zeit, in der der Monolith als Softwarearchitektur vorherrschend war. Bis heute hielt sich die monolithische Architektur der Software, sodass wir ein reines monolithisches System als Grundlage untersuchen können. Ebenfalls ist die Software PEGASOS ein Fall, wie sie in anderen Firmen vorkommen könnte. Heutzutage existieren monolithische Systeme, die in den letzten 20 Jahren entstanden und immer noch weiterentwickelt werden. Mit der Vernetzung, personalisierten Kundenanforderungen und erhöhten Rechenleistung müssen diese Monolithen immer mehr leisten und ein Weg auf Microservices scheint denkbar zu sein. Somit lässt sich die Migration am Fall PEGASOS auf klassische Monolithen beziehen, die aus den gleichen Bedingungen entstanden sind oder eine ähnliche Architektur aufweisen.

2.3 Zusammenfassung

Dieses Kapitel betrachtete die terminologische Klärung des Begriffes Microservices, befasste sich mit den Vor- und Nachteilen und stellte allgemeine Vorgehensweisen für die Migration auf Microservices vor. Anschließend erfolgte die Beschreibung des

Systems PEGASOS, deren bisherige Architektur sowie die derzeitigen Probleme. Abschließend erklärten wir, warum wir das Systeme PEGASOS als Teil einer geeignete Fallstudie ansehen.

3. Methodik

Dieses Kapitel beschreibt die Methodik für die Migration der Software PEGASOS auf Microservices. Der Prozess ist dabei in drei Schritte unterteilt. Der erste Schritt ist eine Literaturrecherche zu bekannten Migrationsprozessen, die von einem Monolith zu Microservices umstellen. Die Resultate der Literaturrecherche werden für den nachfolgendem Schritt genutzt, worin die technische Migrationsmethodik für die Software PEGASOS beschrieben wird, welche Grundlage und Vorgehensweise für die Umsetzung ist. In dem dritten und letzten Schritt wird die Auswertung der Migration beschrieben. Darin werden die Ergebnisse der Umsetzung zusammengefasst und das Ergebnis, in Bezug auf der Leitfrage, präsentiert.

3.1 Literaturrecherche

Einige große Unternehmen wie Netflix[18] oder Spotify[19] setzen bereits erfolgreich Microservices ein, sodass viele Firmen, darunter NEXUS/MARABU, überlegen auf Microservices zu migrieren. Für eine Vorgehensweise, wie eine allgemeine monolithische Software, in unserem Fall PEGASOS, zu Microservices migriert werden kann, fehlen noch aussagekräftige wissenschaftliche Studien. Di Francesco et al. [2019] fanden heraus, dass in der wissenschaftlichen Literatur vorrangig spezielle Probleme und deren Lösungen präsentiert werden und es an grundlegender Forschung, wiederverwendbaren Praktiken und Erfahrungsberichten fehlt. Diese Beobachtung kommt größtenteils daher, dass Microservices zuerst in der Industrie entwickelt wurden und erst später im wissenschaftlichen Kontext untersucht wurden [Soldani et al., 2018]. Außerdem fehlen Studien, die Praktiken für die Migration zu Microservices wissenschaftlich untersuchen [Fritzsch et al., 2019b].

Aus diesen Aspekten heraus, dass in der Literatur vermehrt spezielle Probleme behandelt werden und es an aussagekräftigen allgemeinen Erfahrungsberichten fehlt, führen wir eine eigene Literaturrecherche durch. Die Literaturrecherche besitzt das Ziel, Grundlagen und ähnliche Erfahrungsberichte zu einer Migration von einem

[18] https://developpaper.com/design-and-analysis-of-netflix-microservice-architecture/
[19] https://www.slideshare.net/kevingoldsmith/microservices-at-spotify

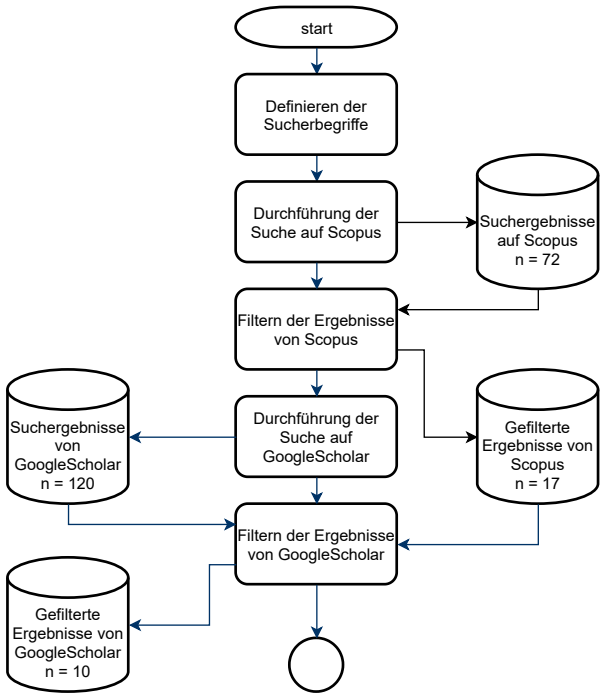

Abbildung 3.1: In diesem Ablaufplan ist die Literaturrecherche

Monolith zu Microservices zu finden, aus denen wir die Vorgehensweise für unsere Arbeit definieren. Der grobe Überblick der Vorgehensweise ist in Abbildung 3.1 dargestellt.

Definieren der Suchbegriffe

In dem ersten Schritt definierten wir für unsere Suche die Begriffe, wonach gesucht werden soll. In unserem Fall versuchen wir die Literatur zu finden, die sich mit der Migration von einem monolithischen System zu Microservices beschäftigt. Dafür definieren wir in erster Linie die beiden Suchbegriffe *Microservices* und *Monolith*. Zusätzlich möchten wir die Suche eingrenzen und bewusst die Quellen finden, die sich mit einer Migration, Refactoring oder mit dem Reengineering beschäftigen. Deswegen fügen wir die Begriffe *Migration*, *Refactoring* und *Reengineering* hinzu, da keine einheitliche Bezeichnung für den Prozess gegeben ist, ob dieser eine Migration ist, ein Refactoring Schritt oder ein Reengineering darstellt. Durch den Umstand, dass einige Quellen nur den Begriff Migration oder nur den Begriff Refactoring verwenden, führen wir drei voneinander unabhängige Suchen S1 - S3 durch. Als Suchmaschinen nutzen wir die Plattformen GoogleScholar[20] und Scopus[21].

S1 Diese Suche benutzt den Suchterm *Microservices Monolith Migration*.

S2 Diese Suche benutzt den Suchterm *Microservices Monolith Refactoring*.

S3 Diese Suche benutzt den Suchterm *Microservices Monolith Reengineering*.

[20] https://scholar.google.com/
[21] https://www.scopus.com/

Filterkriterien

Durch den Umstand, dass nicht jedes Suchergebnis von Relevanz ist, definieren wir die folgenden zwei Kriterien K1 und K2.

K1 Nähe zur Domain PEGASOS und Anwendbarkeit

K2 Ausführlichkeit der Vorgehensweise und deren Diskussion

Wir suchen auf zwei Plattform, Scopus und GoogleScholar, dadurch kann eine Studie sowohl bei Scopus, als auch bei GoogleScholar gefunden werden. Die Dopplungen werden entsprechend nur einmalig hinzugefügt.

Mit dem ersten Kriterium K1 definieren wir die Anwendbarkeit der gefundenen Literatur in Bezug auf unsere Domain mit PEGASOS. Literatur, die ein Problem für eine Domain behandelt, die keine oder nur zu einem sehr geringen Grad eine Verbindung zu unserer Domain besitzt, nehmen wir nicht mit in die weitere Untersuchung auf. Für Microservices existieren viele verschiedene Tools und Frameworks, z.B. Kubernetes[22], Docker Swarm[23] oder Amazon Elastic Container Service[24]. Studien oder Literatur, die die genannten oder andere spezielle Tools nutzen oder auf der Grundlage von speziellen Frameworks ihre Durchführung ausführen, nehmen wir nicht in unsere Betrachtung auf. Dies hat den Hintergedanke, dass wir sehr wenig davon auf unsere Domain mit PEGASOS übertragen können oder uns zu sehr auf eine bestimmte Technologie sowie Umsetzung einschränken.

Das zweite Kriterium ist folgendermaßen definiert. Wir wollen mithilfe der Literaturrecherche eine Vorgehensweise definieren, wie wir den Monolith PEGASOS auf Microservices migrieren können. Für die eigene Vorgehensweise müssen wir die Beweggründe, wie die jeweilige Studie ihre Migration definiert hat, genau nachvollziehen können. Wenn eine Studie den Fokus auf das Resultat und die eigentliche Migration nicht genauer beschreibt, betrachten wir die Studie nicht weiter, da wir ansonsten nicht genau nachvollziehen können, warum bestimmte Methoden oder Techniken angewandt wurden.

Durchführung der Suche auf Scopus

In diesem Schritt führen wir die Suche nach den oben definierten Suchbegriffe auf der Plattform Scopus durch und speichern uns alle Quellen für eine weitere Verarbeitung separat ab. Mit der Suche S1 fanden wir 37 Ergebnisse, mit der Suche S2 30 und die dritte Suche S3 ergab fünf Treffer. Durch diese geringe Anzahl entschieden wir uns, alle dieser Suchergebnisse in dem nächsten Schritt der Betrachtung und Filterung zu untersuchen.

Filtern der Ergebnisse von Scopus

Die insgesamt 72 Ergebnisse über die Suchen S1, S2 und S3 filtern wir in diesem Schritt auf der Grundlage der Kriterien K1 und K2. Dabei übernahmen wir schlussendlich die 17 Suchergebnisse die in der Tabelle 3.1 aufgelistet sind.

[22] kubernetes.io
[23] docs.docker.com/engine/swarm
[24] aws.amazon.com/ecs

Autor und Jahr	Inhalt
Almeida und Silva [2020]	Migrationsunterstützung mithilfe einer Komplexitätsmetrik, wodurch der Ablauf der monolithischen Software auf Microservices, mithilfe des SAGA Patterns [25] angewendet werden kann.
Brahneborg und Afzal [2020]	Analyse eines monolithischen Systems mithilfe der *Architectural Trade-off Analysis Method* kurz ATAM [26]. Die monolithische Software wird im Telekommunikationsbereich, bei der Übermittlung von SMSs, eingesetzt.
Carrasco et al. [2018]	Formulieren neun häufig aufkommende Probleme, wenn von einem Monolith zu Microservices migriert wird. Jeder dieser neun Probleme wird beschrieben und eine mögliche Lösung oder Best Practice dafür geschildert.
Delgado Preti et al. [2021]	Dieses Paper beschreibt die Vorgehensweise für die Migration eines Monoliths zu Microservices einer Software, die im behördlichen Ministerium eingesetzt wird. Der Fokus dieser Arbeit liegt auf dem Migrationsprozess der monolithischen Datenbank und deren Separierung für die Microservices.
Eski und Buzluca [2018]	Beschäftigen sich mit einem automatischen Ansatz, der aus einer bestehenden monolithischen Anwendung die Microservices extrahiert. Der Ansatz beruht auf einer statischen Codeanalyse und einer Graph Clustering Methode.
Firmansyah et al. [2019]	Beschreiben die Migration eines Monolithen im elektronischen Beschaffungssektor zu Microservices. Als Grundlage für die Extraktion der einzelnen Module aus dem Monolith nutzen sie das Domain Driven Design.
Fritzsch et al. [2019a]	Überprüften zehn verschiedene Techniken die beschreiben, wie mögliche Module für Microservices aus einem Monolith gefunden werden können. Daraus entwickelten sie eine Strategie, wie für einen bekannten Monolith die bestmögliche Strategie ausgewählt werden kann, um die möglichen Kandidaten für Microservices zu ermitteln.
Gonçalves et al. [2021]	Untersuchen die Entwicklungskosten und den Einfluss der Performance, wenn ein Monolith zu Microservices migriert wird. Der untersuchte Monolith ist dabei eine Website für die Humanwissenschaft, wo verschiedene Textabschnitte von Bücher gesucht und betrachtet werden können.
Kalske et al. [2018]	Dieses Paper versucht die Gründe der Unternehmen herauszufinden, warum diese ihren Monolith auf Microservices migrieren wollen. Außerdem erörtern sie die technischen und organisatorischen Herausforderungen, die während der Migration von einem Monolith zu Microservices auftreten.
Kazanavicius und Mazeika [2019]	Erörtern verschiedene Techniken und Herausforderungen, die für den Wechsel von einem Monolith zu Microservices auftreten. Die Techniken werden dafür ausführlich auf deren Vor- und Nachteile untersucht.
Kuryazov et al. [2020]	Adressiert die Probleme, die während der Migration von einem Monolith zu Microservice auftreten.
de Lauretis [2019]	Beschreiben eine Vorgehensweise zur Migration einer monolithischen Architektur zu Microservices.

[25] https://microservices.io/patterns/data/saga.html
[26] https://resources.sei.cmu.edu/library/asset-view.cfm?assetid=513908

Autor und Jahr	Inhalt
Mishra et al. [2018]	Bei der Migration von einem Monolith zu Microservices ist eine große Herausforderung, die richtigen Module des Monoliths zu finden, welche später einzelne Microservices werden. Diese Arbeit schlägt einen automatischen Ansatz vor, welcher als Resultat die Module vorschlägt, die zu Microservices migriert werden sollen.
Santos und Rito Silva [2020]	Die Autoren versuchen mit der Arbeit herauszufinden, wie hoch der Aufwand für eine Aufspaltung einer ACID Transaktion auf mehrere verteilte Transaktionen ist und ob ihre vorgeschlagene Komplexitätsmetrik für eine bessere Aufspaltung des Monoliths
Taibi et al. [2018]	Befragten mehrere Unternehmen, welche ihren Monolith auf Microservices umgestellt haben, um herauszufinden was die Motivation und Probleme der Migration waren.
Velepucha und Flores [2021]	Berichten über Probleme und Herausforderungen, die während der Migration zu Microservices auftreten, anhand vorhandener wissenschaftlicher Studien.
Yugopuspito et al. [2017]	Untersuchen anhand eines monolithischen Parkreservierungssystems die Migration hin zu Microservices. Dabei beschreiben sie die Schritte der Vorgehensweise, welche unternommen werden müssen, um das monolithische Design auf Microservices zu übertragen.

Tabelle 3.1: Übersicht über die gefundenen Literatur aus der Literaturrecherche auf Scopus

Durchführung der Suche auf GoogleScholar

Ähnlich zu der Durchführung der Suchplattform Scopus suchten wir mit den drei definierten Suchen S1, S2 und S3 auf der Plattform GoogleScholar. Mit allen drei Suchen fanden wir über 1.000 Resultate, sodass wir eine Entscheidung treffen mussten, welche Resultate wir näher betrachten und welche nicht. Wir entschieden uns dafür, dass wir die Abstrakte der Quellen durchlesen, in der Reihenfolge wie uns die Suchmaschine diese zur Verfügung stellt. Ab der dritten Ergebnisseite - ab dem 30. Eintrag - der Suchmaschine, traf auf immer mehr Quellen das Kriterium K2 zu, sodass diese Quellen keine weitere Bedeutung mehr für uns darstellen. Darüber hinaus sucht GoogleScholar im Volltext der gesamten Quellen, wodurch ab dem 30. Eintrag vermehrt Quellen gefunden wurden, die nicht mehr primär das gesuchte Thema, wie in den Suchen S1, S2 und S3 definiert, beinhalten. Aus den beiden genannten Gründen betrachteten wir für die Suchen S1, S2 und S3 nur die ersten 40 Ergebnisse, die wir für eine weitere Filterung verwenden.

Filtern der Ergebnisse von GoogleScholar

Insgesamt untersuchten wir 120 Ergebnisse über die Suchen S1, S2 und S3 und filterten diese aufgrund der Kriterien K1 und K2. Dabei übernahmen wir schlussendlich 10 Einträge, die in der folgenden Tabelle 3.2 aufgelistet sind.

Autor	Literatur
Bucchiarone et al. [2018]	Erfahrungsbericht einer Bankapplikation, die von einem Monolith auf Microservices migriert wurde und wie dessen Skalierbarkeit dadurch verbesser wurde.

Autor	Literatur
Carvalho et al. [2019]	Beschreiben anhand einer Onlineumfrage mit Spezialisten, welche Kriterien, für die Extraktion der möglichen Microservices aus einem Monolith, von Relevanz sind. Im Ergebnis wurden sieben Kriterien formuliert, die die Entscheidung für die Extraktion des Monoliths zu Microservices erleichtert.
Escobar et al. [2016]	Nutzen einen Prozess um die Abhängigkeiten zwischen den Buisness- und Datenlayer zu visualisieren, die als Grund
Fritzsch et al. [2019b]	Untersuchten zehn verschiedene Unternehmen und deren Migrationsprozess eines Monoliths zu Microservices.
Gouigoux und Tamzalit [2017]	Zeigten die Ergebnisse eines Migrationsprozesses anhand einer Finanzsoftware, dessen Kernlogik auf Microservices umgestellt wurde und schilderten die gewonnenen Erkenntnisse daraus.
Kazanavicius und Mazeika [2019]	Evaluieren verschiedene Techniken und Herausforderungen für die Migration eines Monoliths zu Microservices und beurteilen dessen Vor- und Nachteile.
Knoche und Hasselbring [2019]	Präsentieren das Ergebnis einer Umfrage zu Unternehmen, die Microservices einsetzen wollen und was deren Beweggründe sowie Hindernisse, für deren Einsatz, sind.
Megargel et al. [2020]	Stellen eine mögliche Migrationstechnik vor einen Monolithen auf Microservices zu migrieren. Der Monolith ist dabei eine Software aus dem Bankensektor.
N Bjørndal et al. [2020]	Beschreiben eine Methodik die überprüft, ob eine Migration von einem Monolith zu Microservices Vorteile bringt, in Bezug auf die Performance und Skalierbarkeit der Software.
Sindre Grønstøl Haugeland et al. [2021]	Anhand einer Enterprise Software schlagen die Autoren eine Vorgehensweise vor, wie ein bestehender Monolith auf Microservices migriert werden kann.

Tabelle 3.2: Übersicht über die gefundenen Literatur aus der Literaturrecherche auf GoogleScholar

Diskussion Literaturrecherche

Die hier durchgeführte Literaturrecherche besitzt nicht den Anspruch einer vollständigen und vollumfänglichen Recherche. Unser Ziel ist es, dass wir einen groben Überblick über vorhandene Migrationsstrategien bekommen und nicht jegliche Strategien evaluieren. In diesem Abschnitt möchten wir die Punkte erwähnen, die für eine vollständige Literaturrecherche, aus unserer Sicht, angebracht wären und wo wir Schwachstellen in unserer gewählten Methodik sehen.

Bezüglich der gewählten Suchtermen S1, S2 und S3 könnte man diese noch erweitern um allgemeine Begriffe, die auf eine Architektur zutreffen würden, wie *System*, *Design* oder *Structure*. Damit werden Quellen gefunden, die ihre Software nicht als Monolith bezeichnen, aber dennoch eine Migration durchführen. Darüber hinaus gibt es mehrere Schreibweisen von dem Begriff *Microservices*, z.B. *Micro-Service*, *Microservice* (ohne dem *s* am Ende) oder *Micro Service*. Besonders in früheren Quellen, wo der Begriff *Microservices* noch nicht allgemein definiert war, sind mehrere Schreibweisen dieses Begriffes denkbar.

Eine weitere Schwachstelle, die wir in unserer Methodik sehen, ist die Auswahl der beiden Suchplattformen GoogleScholar und Scopus. Für eine vollständige Suche emp-

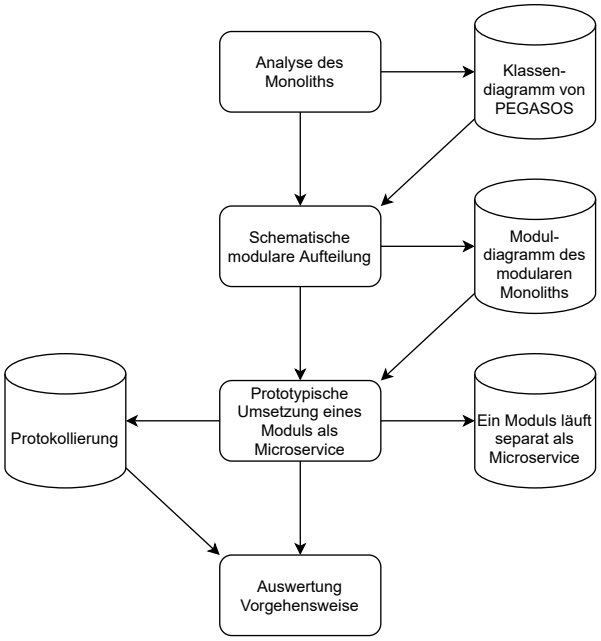

Abbildung 3.2: Methodik der Umsetzung

fehlen wir die Hinzunahme von weiteren Suchmaschinen wie ACM Digital Library[27], IEEE Explore[28] oder ISI Web of Science[29]. Über diese Hinzunahme der Suchmaschinen können weitere Quellen gefunden und mit in die Evaluierung aufgenommen werden.

In vielen Literaturrecherchen kommt die Methodik des sogenannten *Snowballing* zum Einsatz. Diese Technik ist ein Mittel, um weitere relevante Literatur zu einem Thema zu finden, wenn eine vorhandene Menge an Literatur bereits vorliegt. Wohlin [2014] beschreibt eine mögliche Richtlinie und Vorgehensweise für das *Snowballing*. In unserer Literaturrecherche besitzen wir ebenfalls eine Menge an Literatur, durch die Ergebnisse von GoogleScholar und Scopus, sodass ein anschließenden *Snowballing* empfehlenswert wäre, weitere qualifizierte Literatur zu finden.

3.2 Migration

Auf der Grundlage der vorhergehenden Literaturrecherche wird eine Methodik definiert, wie der Monolith PEGASOS auf Microservices migriert werden kann. In der Abbildung 3.2 ist die Vorgehensweise visualisiert, welche in die vier Schritte *Analyse des Monoliths*, *Moduluare Aufteilung*, *Prototypische Umsetzung eines Moduls als Microservice* und *Auswertung Vorgehensweise* aufgeteilt ist.

[27] https://dl.acm.org/
[28] https://ieeexplore.ieee.org
[29] https://login.webofknowledge.com/

3.2.1 Analyse des Monoliths

Aus der Definition von Microservices, siehe Kapitel 2, ist ein einzelner Microservice eine Einheit, die eine bestimmte Aufgabe innerhalb der Software übernimmt. Dadurch das für die Software PEGASOS keine ausführliche Dokumentation der derzeitigen Softwarearchitektur vorliegt, versuchen wir in dem ersten Schritt *Analyse des Monoliths* herauszufinden, wie der Monolith PEGASOS derzeit strukturiert ist. Das resultierende Klassendiagramm dient als Grundlage für den darauffolgenden Schritt *Schematische modulare Aufteilung*, damit potentielle Microservicekandidaten herausgefunden werden können.

Diesen ersten Schritt finden wir bei vielen Studien, dass zuerst die vorhandene Klassenstruktur untersucht und analysiert wird de Lauretis [2019]; Delgado Preti et al. [2021]; Kuryazov et al. [2020]. Eine Möglichkeiten die Klassenstruktur zu beschreiben stellt Mishra et al. [2018] vor, indem der Monolith zuerst Formal durch ein UML Klassendiagramm dargestellt wird und anschließend die einzelnen Klassen zu Komponenten, die am Ende einen Microservice bilden, zusammengefasst werden.

Ebenfalls existieren für diesen ersten Schritt, welcher im Englischen *Decomposition* oder *Extraction* bezeichnet wird, mehrere automatische Ansätze, da besonders für große und komplexe Software eine manuelle Analyse und Auswertung als zu komplex und aufwendig erscheint. Eski und Buzluca [2018] schlagen zum Beispiel anhand einer Komplexitätsmetrik und einer Clusteringmethodik eine Vorgehensweise vor, womit ein bestehender Monolith untersucht werden kann. Allerdings finden wir die automatischen Ansätze zu experimentell und speziell an eine bestimmte Domain oder auf ein bestimmtes Framework gebunden. Daher fokussieren wir uns auf Methodiken, welche wir mit unserer Software PEGASOS voraussichtlich ohne Probleme durchführen können. In unserem Fall von PEGASOS entscheiden wir uns daher für die Erstellung eines UML-Diagramms der Klassenstruktur von PEGASOS, welches Carvalho et al. [2019]; Escobar et al. [2016] vorschlagen. Die Erstellung eines UML-Diagramms von PEGASOS können wir, mithilfe der Verwendung des UML-Tools von JetBrains[30], effektiv durchzuführen.

Carvalho et al. [2019] fanden in einer Umfrage heraus, dass die beiden Metriken Kohäsion und Kopplung sehr nützliche Kriterien sind, wenn die Analyse des Monoliths durchgeführt wird aber es in der praktischen Umsetzung an nützlichen und anwendbaren Tools fehlt. Kohäsion beschreibt inwieweit gleiche Teile des Codes logisch oder funktional zusammengehören, währenddessen Kopplung die funktionale Verbindung zwischen zwei Klassen misst, z.B. durch einen Methodenaufruf. Für die Kopplung nutzen wir die Tools von IntelliJ, indem wir dort die eingehenden und ausgehenden Verbindung von Klassen und Modulen messen können, sowie zyklische Referenzen in der Software auffinden können. Bezüglich der Kohäsion nutzen wir die Erfahrungen der Entwickler von NEXUS/MARABU um zu ermitteln, inwieweit bestimmte Teile des Quellcodes zusammengehören oder lieber getrennt betrachtet werden sollten.

[30] https://www.jetbrains.com/help/idea/class-diagram.html

3.2.2 Schematische modulare Aufteilung

Nachdem wir einen Überblick über die vorhandene Klassenstruktur erstellt haben, versuchen wir in diesem Schritt herauszufinden, welche Klassen wir zu einzelnen Modulen zusammenfassen, aus denen später die Microservices werden [Newman, 2015]. Nach der Art und Weise wonach der Monolith in Module unterteilt werden soll, empfiehlt sich die Methodik des Domain Driven Designs Evans und Fowler [2019]. In Anlehnung an das Domain Driven Design werden die Grenzen der Module definiert und die Funktionalitäten des Moduls gegenüber Anderen abgrenzt [Newman, 2021]. Dieses Vorgehen, dass zuerst der Monolith in einzelne Module, nach dem Domain Driven Design aufgeteilt wird, findet sich in vielen Studien wieder [de Lauretis, 2019; Firmansyah et al., 2019; Kalske et al., 2018; Taibi et al., 2017]. Gonçalves et al. [2021] beschreibt den Vorgang des Aufteilens des Monoliths genauer, indem die einzelnen Module des Monoliths keine zyklischen Referenzen besitzen dürfen und die Kommunikation entweder über ein Interface oder Eventsystem stattfindet. Dies ist notwendig, damit die Schnittstellen und Eventsysteme durch ein Netzwerkkommunikationsprotokoll, z.B. mit REST[31], implementiert werden können.

Unter der Voraussetzung, dass ein Monolith modular aufgebaut und die Logik sauber separiert ist, lässt sich dieser Schritt relativ einfach durchführen, da die Module in diesem Monolith idealerweise bereits die Microservices darstellen. PEGASOS ist allerdings eine hochkomplexe Software, die sich nicht in ein bestimmtes Schema packen lässt und somit auch die modulare Aufteilung eine Herausforderung darstellt. Wir nutzen entsprechend die Analyse aus dem vorherigen Schritt und überprüfen, ob wir in dem Abhängigkeitsgraph des Klassendiagramms bestimmte Muster erkennen. Dieser Schritt wird in enger Zusammenarbeit mit den Entwicklern von PEGASOS durchgeführt, ob die von uns neu definierten Module und ihre Abgrenzungen Sinn ergeben, in Hinblick auf deren Verwendung innerhalb der Software.

Bei einem großen Monolith sollte allerdings nicht gleich die gesamte Software auf Microservices umgestellt werden, sondern immer nur ein Modul nach dem anderen oder nur die Module, die sich für Microservices eignen [Carrasco et al., 2018; Velepucha und Flores, 2021]. Besonders wenn die praktischen Erfahrungen im Team, wie es in der Firma NEXUS/MARABU der Fall ist, nicht vorhanden sind, sollte die Migration hin zu Microservices in kleinen Schritten vollzogen werden [Kalske et al., 2018]. Dies hat den Vorteil, dass für ein einzelnes Modul entsprechende Erfahrungen gesammelt und anschließend für die weitere Migration genutzt werden können. Ebenfalls ist der investierte Aufwand geringer, als wenn der gesamte Monolith in einem Schritt migriert wird, wobei im Falle einer fehlerhaften Migration die Kosten geringer sind. Entsprechend entscheiden wir uns dafür, dass wir in dieser Arbeit nur ein einzelnes Modul auf Microservices migrieren, damit wir die gesammelten Erfahrungen für eine weitere zukünftige Migration verwenden können.

Nachdem die Komponente feststeht, welche wir als einen Microservice implementieren, konzipieren wir einen genauen Architekturplan, worin zyklische Referenzen, Abhängigkeiten von und zu anderen Modulen aufgelöst und die Schnittstellen definiert sind. Dieser Architekturplan bildet die programmiertechnische Grundlage der Umsetzung. Zusätzlich zu dem Architekturplan ist an dieser Stelle zu entscheiden, mit welcher

[31] https://restfulapi.net/

Technologie wir den Microservice umsetzen und welches Kommunikationsprotokoll wir für die Übertragung der Daten verwenden. Ein großer Vorteil von Microservices ist die individuelle Verwendung der Technologie für jeden Microservice [Newman, 2015], damit für die jeweilige Aufgabe, die der Microservice übernimmt, eine passende Technologie ausgewählt werden kann. In unserem Fall entscheiden wir uns allerdings dafür, dass wir bei der derzeitigen verwendeten Technologie von PEGASOS bleiben, damit wir die eigentlichen Probleme bei einer Migration zu Microservices herausfinden ohne das wir zu sehr auf Probleme stoßen, die durch fehlende Erfahrungen mit einer bestimmten Technologie entstehen.

3.2.3 Prototypische Umsetzung eines Moduls als Microservice

Aufgrund der verschiedene Domain, Komplexität, Erfahrungen im Team und Technologiestack ist der Monolith PEGASOS ein einzigartiger Fall, für den keine Anleitung für die Migration zu Microservices existiert. Allgemein existiert nicht die beste Anleitung, um zu Microservices zu migrieren, nur erfolgreiche Migrationen und Praktiken, die sich für bestimmte Produkte und Situationen geeignet zeigten [Kazanavicius und Mazeika, 2019]. Daher verwenden wir einen iterativen Prozess, wodurch wir auf aufkommende Probleme schnell reagieren können und die anschließende Lösung und gesammelten Erfahrungen für den weiteren Prozess nutzen können.

Carrasco et al. [2018] erwähnen eine Problematik, dass durch die Komplexität von Microservices ein Team aus unerfahrenen Entwickler keine gute Idee sind, da die Unerfahrenheit zu höheren Kosten und einer längeren Entwicklungszeit führt. Die Firma NEXUS/MARABU besitzt keine Entwickler, die eine Praxiserfahrung, insbesondere in der Domaine von PEGASOS, besitzen. Für das nicht Vorhandensein von erfahrenden Entwickler schlagen Carrasco et al. [2018] vor, dass die Migration zu Microservices von einen kleinen Team vorgenommen werden soll und wenn dieses Team genug Erfahrungen gesammelt hat, dieses vergrößert wird. Dieses Vorgehen übernehmen wir, sodass wir einerseits in einem kleinen Team von zwei Personen beginnen und einem iterativen Prozess folgen.

Wir wollen mit der Umsetzung herausfinden, welche Probleme bei der Migration auftreten und was für die zukünftige Migration von PEGASOS zu Microservices wichtig ist. Dafür setzen wir während der Umsetzung eine Protokollierung um, die während des Prozesses als Grundlage dient, um die Vorgehensweise anzupassen und eine spätere Auswertung durchführen zu können. Das geführte Protokoll wird in dem Abschnitt 3.2.5 genauer beschrieben.

3.2.4 Iterativer Prozess der Migration

Der iterative Prozess ist folgendermaßen strukturiert und in der Abbildung 3.3 dargestellt. An erster Stelle steht eine Ausarbeitung der konkreten Umsetzung an, worin die Softwarearchitektur und der Migrationsprozess beschrieben ist. Dazu gehört, welche Klasse zuerst migriert wird, welche Pattern und Softwarepraktiken zum Einsatz kommen und wie die konkrete Softwarearchitektur auf Klassenebene aussieht. Nach der anfänglichen Planung beginnt der eigentliche iterative Prozess, der die vorher definierte Vorgehensweise umsetzt. Der iterative Prozess ist Tagesbasiert, sodass jeder Arbeitstag, an dem an der Migration gearbeitet wird, als eine einzelne Iteration angesehen wird.

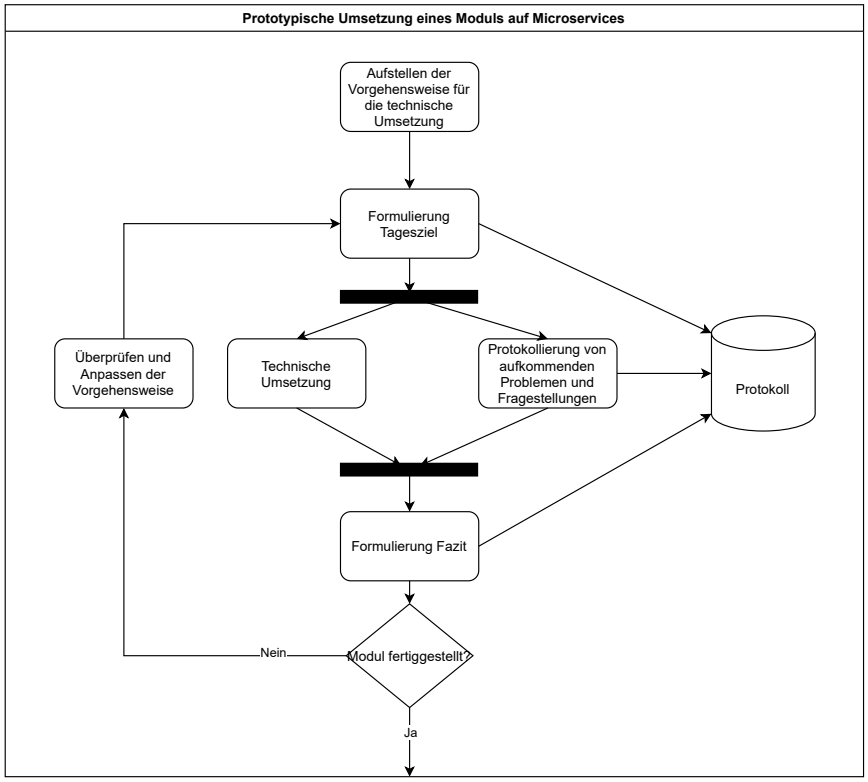

Abbildung 3.3: Detaillierte Vorgehensweise für die prototypische Umsetzung

Zu Beginn jeder Iteration legen wir das Tagesziel fest, welches beinhaltet, was wir an dem Arbeitstag konkret erreichen wollen. Dieses Tagesziel dient später der Validierung um herauszufinden, inwieweit sich unsere geplante Vorgehensweise umsetzen lies und wo eventuelle Probleme auftraten. Nach der Festlegung des Tagesziels beginnt die eigentliche Umsetzung der Migration zum Microservice auf technischer Basis. Parallel zu dieser Umsetzung werden Probleme und aufkommende Fragestellungen in dem Protokoll vermerkt, die während der Umsetzung auftreten.

Am Ende des Arbeitstages wird als Zusammenfassung aus der technischen Umsetzung, dem formulierten Tagesziel und den aufkommenden Problemen ein Fazit gezogen. Dieses Fazit beinhaltet eine Beschreibung des Resultats, was am Ende des Arbeitstages erreicht wurde und einen Ausblick, ob die geplante Vorgehensweise weitergeführt werden kann oder ob diese angepasst werden muss. Das Fazit wird anschließend ebenfalls im Protokoll vermerkt.

Nach der Formulierung des Fazits schauen wir, ob wir das Modul fertiggestellt haben oder ob wir noch offene Aufgaben der Vorgehensweise haben. Für den Fall, dass das Modul fertiggestellt ist, betrachten wir den iterativen Prozess als abgeschlossen. Für den anderen Fall überprüfen wir die Vorgehensweise, auf der Grundlage der Protokolle und passen eventuell die Vorgehensweise an. Anschließend beginnt der iterative Prozess an einem weiteren Arbeitstag mit einer neuen Iteration, welcher ebenfalls mit der Formulierung des Tagesziel beginnt.

3.2.5 Protokollierung

Damit durch die schrittweise Migration auf Microservice Erkenntnisse gezogen werden können und der Prozess an die aufkommenden Probleme angepasst werden kann, wird die Migration mitprotokolliert. Die Protokollierung findet für jeden Arbeitstag, an dem an der Migration gearbeitet wird, statt. Das Protokoll ist in die folgender Felder unterteilt, ein Beispiel ist in der Tabelle 3.2.5 zu finden.

In dem Feld *Datum* ist der jeweilige Tag notiert, an dem ein Schritt des iterativen Prozesses durchgeführt wurde. Dieses Datum dient vorrangig der chronologischen Zuordnung der durchgeführten Arbeitsschritte.

Zu Beginn eines jeden Arbeitstages wird in dem Feld *Tagesziel*, dass jeweilige geplante Vorgehen beschrieben. In unserem Beispiel wurde an dem Arbeitstag geplant, dass die Logik der Klasse *PhysicalArchiveServiceBean* in den Microservice verschoben wird.

Das Feld *Resultat* dient der Überprüfung, inwieweit das Tagesziel erreicht und was davon konkret umgesetzt wurde. Die Beschreibung ist lediglich eine Bestandsaufnahme, was an dem Ende des Arbeitstages erreicht wurde. In unserem Fall wurde nur eine einzelne Methode der Klasse *PhysicalArchiveServiceBean* in den Microservice verschoben. Die vorgenommene Planung, dass alle Methoden der Klassen migriert werden, wurde somit nicht erreicht.

Während der Umsetzung dient das Feld *Aufkommende Probleme* für problematische Umstände, die während der Entwicklung auftreten. Dieses Feld wird parallel zu der praktischen Umsetzung ausgefüllt, wenn etwaige Probleme auftreten. In unserem Fall ist während des Prozesses aufgefallen, dass die Klasse zu komplex ist und nicht

an einem Tag komplett migriert werden kann. Darüber hinaus sind Probleme mit den Datenobjekten, die in den Datenbanken gespeichert werden, aufgetreten.

Gegenüber den aufkommenden Problemen werden in dem Feld *Aufkommende Fragestellungen* Fragen notiert, die während der Migration auftreten. Inhaltlich behandeln diese Fragen neue Themengebiete, Diskussionspunkte oder eine Unklarheit, die es in einer weiteren Iteration zu berücksichtigen gilt. In unserem Fall wurden die Fragen bezüglich verschiedene Thematiken, hierbei Logging oder den Datenbankzugriffen, aufgeschrieben, die in der Vorbereitung nicht berücksichtigt wurden sind.

Der letzte Punkt *Fazit* in dem Protokoll ist ein Fazit von der einen Iteration. In unserem Fall ist das Fazit die Erkenntnis, dass im Detail viele Probleme auftreten, die einzeln behandelt werden müssen und selbst die Migration einer einzelnen Klasse komplex sein kann. Daher ist hierbei die weitere Überlegung, in kleinen Schritten zu migrieren, in unserem Fall Methode für Methode und die alte Klasse parallel laufen zu lassen für die Logiken, die noch nicht migriert wurden sind.

3.2.6 Auswertung der Vorgehensweise

Nach der technischen Umsetzung nutzen wir die Protokolle für eine Auswertung, in der die gewonnenen Erkenntnisse dargestellt werden, mittels einer Interpretation der Protokolle. Dazu vergleichen wir zuerst das Tagesziel mit dem Resultat und überprüfen inwieweit das Ziel erreicht wurde. Anschließend untersuchen wir die Probleme und Fragestellungen, die in den Protokollen beschrieben sind und gruppieren sie in zusammenhängende Bereiche. Dabei fassen wir Probleme, Fragestellungen und die Fazits zusammen, die entweder der gleichen Ursache zugrundeliegend oder die einer Thematik zuzuordnen sind. Abschließend werten wir die zusammengefassten Informationen aus den Protokollen aus, indem wir zu jedem Bereich die gewonnene Erkenntnis formulieren.

3.3 Zusammenfassung

In dem ersten Abschnitt dieses Kapitel zeigten wir die Vorgehensweise unserer Literaturrecherche, aus der die Vorgehensweise für die Umsetzung definiert wurde. Über die Suchplattformen GoogleScholar und Scopus und nach der Filterung mit bestimmten Kriterien verblieben 27 wissenschaftliche Studien. Diese Studien wurden in dem zweiten Abschnitt dieses Kapitel untersucht, um aus denen eine Vorgehensweise für unsere Migration zu definieren. Die definierte Vorgehensweise für die Umsetzung besteht aus den vier Abschnitten Analyse des Monoliths, Schematische modulare Aufteilung, Prototypische Umsetzung und Auswertung der Vorgehensweise.

Datum	04.08.2021
Tagesziel	Auf Grundlage der theoretischen Ausarbeitung werden die ersten Logiken in den Microservice verschoben. Dabei steht die Klasse PhysicalArchiveServiceBean im Vordergrund, da die Logik in dieser Klasse überschaubar ist und für eine erste Iteration als geeignet erscheint.
Resultat	Eine Methode der Klasse PhysicalArchiveServiceBean wurde in den Microservice verschoben. In dem Monolith existiert ein Proxy, welcher die Aufrufe entweder auf den Microservice umleitet oder noch die alte Logik im Monolith aufruft, je nachdem welche Methode aufgerufen wird.
Aufkommende Probleme	Bei der Umsetzung viel auf, dass für einen Anfang die komplette Klasse zu viel ist, die auf einmal in den Microservice zu verschieben. Entsprechend wurde zu einer einfacheren Lösung gegriffen und nur eine einzelne Methode der Klasse in den Microservice verschoben. Alle anderen Funktionalitäten bleiben noch im Monolith. Einige Methoden sind komplexer als andere, um einen Anfang zu haben, wurde eine Methode genommen, die keine Datenbankzugriffe und nur sehr wenige Zugriffe auf andere Systeme des Monoliths besitzt. Bei vielen anderen Methoden sind viele Kleinigkeiten und Details, die nicht zu einer einfachen Lösung geführt haben. Dazu gehören die Datenbankzugriffe, Objekte die diese Datenbankobjekte darstellen – die besitzen mitunter über 1.000 Codezeilen. Ebenfalls werden Klassen benutzt, die sowohl Daten, aber auch Logiken beinhalten.
Aufkommende Fragestellungen	Vor allem die Themen wie Logging, Session und die Datenbankzugriffe führten bei der Umsetzung zu Unklarheiten, weil nicht eindeutig ist, wo diese Funktionalitäten hingehören. Tendenziell wurde gesagt, dass Logging, Session und die Datenbankzugriffe nicht in den Microservice kommen, sondern diese Funktionalität über eine REST-Schnittstelle im Monolith zur Verfügung gestellt werden. Dies sorgt wiederum für einen zusätzlichen Aufruf vom Microservice zurück an den Monolithen.
Fazit	Am Ende der Umsetzung wurde statt der vorgenommenen kompletten Klasse nur eine einzelne Methode in den Microservice verschoben. In dem Zuge konnte gleich der Prozess ausprobiert werden, wie eine solche Umsetzung parallel zur Entwicklung stattfinden kann. Indem iterativ einzelne Methoden in den Microservice verschoben werden, bis am Ende alle Methoden im Microservice sind und die alte Logik im Monolith gelöscht werden kann oder eventuell noch zur Überprüfung der Daten genutzt werden kann.

Tabelle 3.3: Beispielhaftes Protokoll, welches während der Migration geführt wurde.

4. Durchführung

In diesem Kapitel beschreiben wir die Durchführung der vorher definierten Methodik, wie wir vorgegangen sind und welche Ergebnisse wir produzierten. Dabei ist dieses Kapitel in zwei Abschnitte aufgeteilt. Der erste Abschnitt beschäftigt sich mit der Analyse des Monoliths sowie deren schematische Aufteilung in Module und der zweite Abschnitt stellt den iterativen Prozess und deren technische Umsetzung vor.

4.1 Resultat der Analyse

Die Analyse des Monoliths führten wir mit dem UML-Tool von IntelliJ durch und versuchten dadurch, einen Überblick über die derzeitige Klassenstruktur zu erhalten. Das Ergebnis des automatisch generierten Klassendiagramms war allerdings nicht brauchbar, da der Monolith über 5.000 Javaklassen besitzt und durch diese hohe Anzahl eine manuelle Überprüfung nicht praktikabel war. Daher versuchten wir den Monolith über die Ordnerstruktur näher zu untersuchen, da die Ordner, dem Anschein nach, in einzelne Funktionalitäten aufgeteilt sind. Bei der Ordnerstruktur fiel uns auf, dass diese einzelnen Aufgabenbereichen zugeordnet sind, z.B. das Archivsystem, die FHIR Integration oder das Mailingsystem, die potentielle Microservice Kandidaten darstellen. Allerdings war diese Struktur auch noch zu komplex, als dass wir eine genaue Aussage treffen konnten, mit welchem Modul wir anfangen, um es zu migrieren. Nach einer Rücksprache mit dem verantwortlichen Entwickler evaluierten wir die verschiedenen Aufgabenbereiche und entschieden uns letztlich, dass wir mit der Migration der Aktenverwaltung beginnen, da diese von dessen Aufgabe her von den anderen Modulen abgekapselt ist und eine gewisse Eigenständigkeit aufweist.

Entsprechend betrachteten wir nur noch die Klassen in dem Ordner der Aktenverwaltung, die ca. 150 Klassen besitzt. Während wir die Abhängigkeiten untersuchten stellten wir fest, dass die Klassen der Aktenverwaltung eine hohe Kopplung zu anderen Klassen des Monoliths besitzen, die nicht Teil der Aktenverwaltung sind. Dazu gehören einerseits Utilityklassen, die bestimmte Funktionalitäten anbieten, Datenobjekte, die global verwendet werden oder Singletons, die bestimmte Referenzen auf andere Systeme bereitstellen, beispielsweise den Zugriff auf die Datenbank. Die

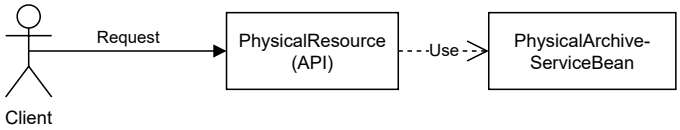

Abbildung 4.1: Der Requests des Clients wird zuerst in der PhysicalResource Klasse verarbeitet und anschließend an den PhysicalArchiveServiceBean weitergeleitet, worin die Logik zur Verarbeitung implementiert ist.

Referenzen bestanden dabei sowohl von der Aktenverwaltung zu anderen Klassen des Monoliths, als auch dass andere Klassen außerhalb der Aktenverwaltung auf die Klassen der Aktenverwaltung zugreifen, somit zyklische Referenzen. Wegen der starken Kopplung der Aktenverwaltung war eine Migration keine leichte Aufgabe, da zuerst die Referenzen aufgelöst werden mussten.

Durch die starke Kopplung der Aktenverwaltung wäre es eine zu komplexe Aufgabe gewesen, dass wir versuchen, die gesamte Aktenverwaltung in einem Schritt zu migrieren. Somit benötigten wir einen einfacheren Einstiegspunkt und suchten daher nach der API, an der die Requests der Clients geschickt werden. Von dort aus versuchten wir anschließend der Logik zu folgen, um ein besseres Bild von der Funktionalität zu bekommen. Dieser Ansatz zeigte eine deutlich bessere Vorstellung der Struktur. Die Requests werden zuerst im Clientlayer von PEGASOS verarbeitet, wovon sie an entsprechende Beans[32] weitergeleitet werden. Ein Aufruf, der den Ablageort einer Akte von einem Patienten ändern soll, ist in der Abbildung 4.1 dargestellt. Zuerst wird der Requests eines Clients in der PhysicalResource Klasse verarbeitet und anschließend der PhysicalArchiveServiceBean aufgerufen, welcher die Logik zur Verarbeitung der Aufgabe enthält. Nach einer Absprache mit den verantwortlichen Entwicklern entschieden wir uns dafür, dass wir den PhysicalArchiveServiceBean in einen Microservice verschieben.

4.2 Modulare Aufteilung

Damit wir den ausgewählten PhysicalArchiveServiceBean in den Microservice migrieren konnten, untersuchten wir die Kopplung des PhysicalArchiveServiceBean zu dem Monolith PEGASOS. In dem Kapitel 3 stellten wir heraus, dass das zu extrahierende Modul keine zyklischen Referenzen zu dem Monolith PEGASOS besitzen soll und die Referenzen lediglich von dem Modul zu dem Monolith zeigen sollen. Unter diesen Voraussetzungen ist eine Migration zu einem Microservice einfacherer. In unserem Fall untersuchten wir den PhysicalArchiveServiceBean und dessen Referenzen auf das System PEGASOS und stellten folgende Beobachtung fest. Der Bean besaß einerseits Referenzen zu anderen Systemen wie den Zugriff auf die Datenbank, das Sessionhandling oder das Cachesystem, aber es waren auch Referenzen vorhanden von anderen Systemen zum PhysicalArchiveServiceBean, wie in Abbildung 4.2 dargestellt.

[32] https://www.geeksforgeeks.org/enterprise-java-beans-ejb/

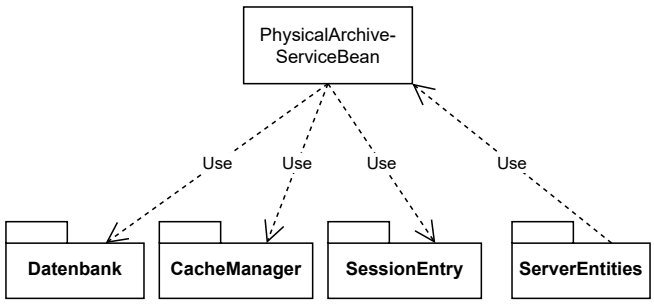

Abbildung 4.2: Exemplarische Auswahl der Abhängigkeiten des PhysicalArchiveServiceBean zu und von anderen Systemen in PEGASOS.

Bevor wir die Migration durchführen konnten, mussten wir den PhysicalArchiveServiceBean als unabhängiges Modul von dem Monolith trennen. Dafür lösten wir die Referenzen von und zu dem PhysicalArchiveServiceBean folgendermaßen auf. In Abbildung 4.3 ist die Klassenstruktur zwischen den ServerEntities und dem PhysicalArchiveServiceBean vor und nach dem Refactoring vereinfacht dargestellt. Die ServerEntities greifen direkt auf den PhysicalArchiveServiceBean, welches wir durch das Pattern Interface-Segregation[33] verhindern, indem wir die konkrete Implementierung durch ein Interface abkapselten. Dies ermöglichte uns später in der Umsetzung den Einsatz des Proxy-Patterns[34], um die Logik an den Microservice zu delegieren. Die zweite Art von Referenz, die der PhysicalArchiveServiceBean auf eine konkrete Klasse des Monoliths PEGASOS hatte, lösten wir ebenfalls durch das Pattern Interface-Segregation auf. In Abbildung 4.4 ist dieser Vorgang dargestellt, dass der PhysicalArchiveServiceBean nicht mehr direkt auf den CacheManager zugreift und seine Implementierung über ein Interface verborgen bleibt. Gegenüber den hier dargestellten Abhängigkeiten waren noch weitere Klassen durch Referenzen verbunden, die allerdings jeweils durch einen der beiden dargestellten Fälle ebenfalls aufgelöst werden konnten.

Der nächste Schritt war, dass wir den PhysicalArchiveServiceBean in den Microservice verschieben und im Monolith einen Proxy implementieren, der die Aufrufe an den Microservice weiterleitet. In der Abbildung 4.5 ist die neue Architektur mit dem Microservice abgebildet. Ein Request eines Clients wird von der PhysicalResource von PEGASOS empfangen, welcher die Anfrage verarbeitet und an das Interface IPhysicalArchiveServiceBean weiterleitet. Die Implementierung dieses Interfaces ist nicht mehr der PhysicalArchiveServiceBean, sondern ein Proxy, welcher einen neuen Request erstellt und die Anfrage an die API des Microservices weiterleitet. Innerhalb des Microservices sieht es ähnlich wie in dem Monolith PEGASOS aus, allerdings ist dort in dem PhysicalArchiveServiceBean die Logik für die Verarbeitung vorhanden.

[33]https://www.baeldung.com/solid-principles
[34]https://refactoring.guru/design-patterns/proxy

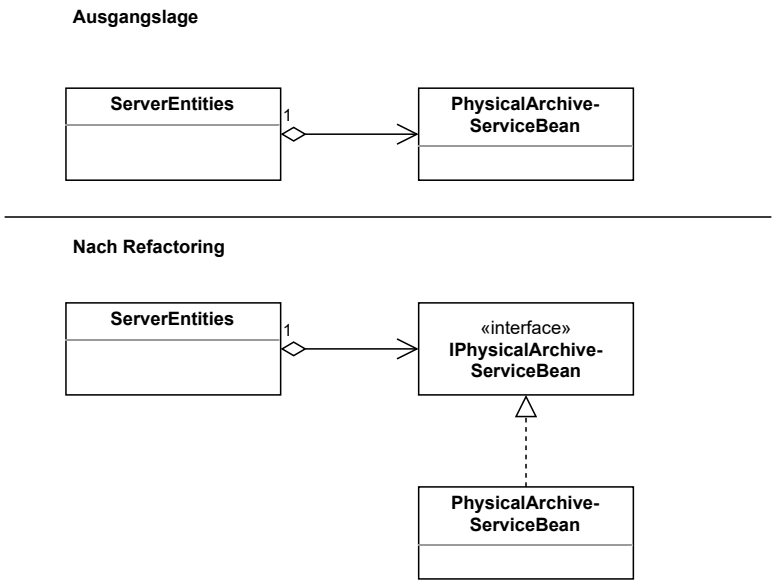

Abbildung 4.3: Auflösen der direkten Abhängigkeiten zwischen den ServerEntities und dem PhysicalArchiveServiceBean.

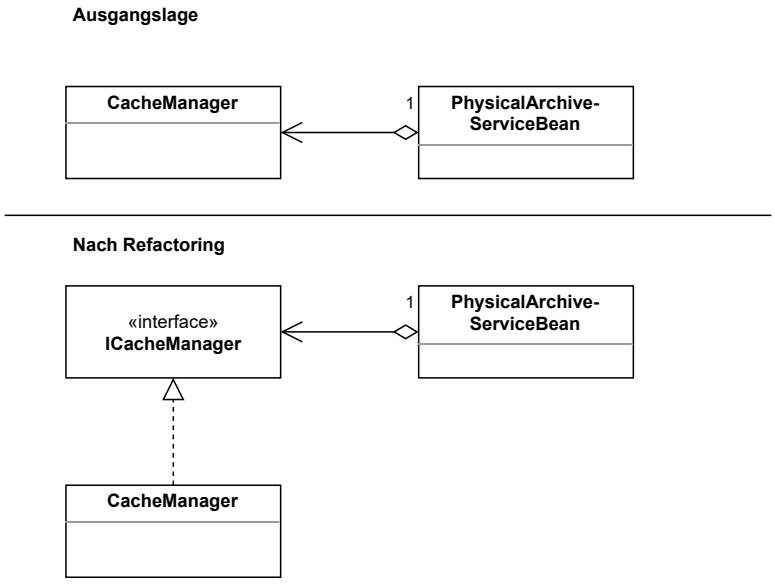

Abbildung 4.4: Auflösen der direkten Referenz des PhysicalArchiveServiceBean auf den CacheManager.

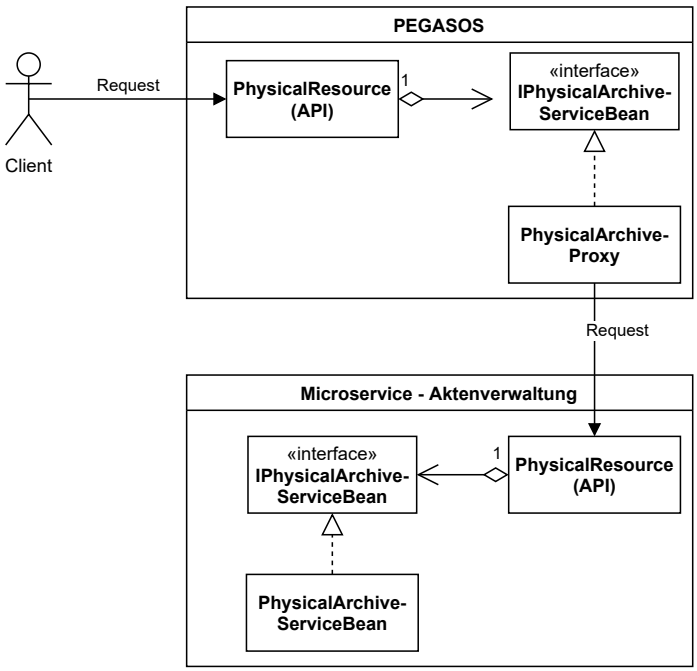

Abbildung 4.5: Die Kommunikation des Monoliths PEGASOS mit dem Microservice der Aktenverwaltung.

Damit war die benötigte Logik, die der Monolith PEGASOS vom Microservice benötigt umgesetzt, allerdings benötigte der Microservice Logiken, die in dem Monolith PEGASOS vorhanden waren, z.B. die Datenbankzugriffe. Bei der Auflösung der Referenzen, verglichen mit Abbildung 4.4, von dem Microservice zu anderen System führten wir ein Interface als Abstraktionsschicht ein, welches die eigentliche Logik versteckt. Dieses Interface, in unserem Beispiel IDatabase siehe Abbildung 4.6, war sowohl in dem Monolith als auch in dem Microservice vorhanden. Wenn der PhysicalArchiveServiceBean in dem Microservice eine Funktionalität der Datenbank benötigte, wird die Funktionalität über einen implementierten Proxy als Request an den Monolith PEGASOS weitergeleitet. Innerhalb des Monoliths PEGASOS ist ebenfalls das Interface IDatabase vorhanden, wohinter sich die tatsächliche Logik auf die Datenbank befand.

Zusammenfassung Analyse des Monoliths

Das Resultat der Analyse zeigte eine mögliche Architektur für den PhysicalArchive-ServiceBean, welcher in einen Microservice verschoben wurde und deren Abhängigkeit zu dem noch vorhandenen Monolith PEGASOS. Diesen Architekturplan versuchten wir in der prototypischen Umsetzung anschließend umzusetzen. Im Bezug auf die vorgestellte Architektur wiesen wir auf einige Umstände hin, die es zu beachten gilt.

Der Request von einem Client erfolgte direkt an den Monolith, wurde dort verarbeitet und an den Microservice weitergeleitet, dabei wäre es doch scheinbar geeigneter gewesen, wenn der Client direkt den Microservice aufrufen würde, da PEGASOS

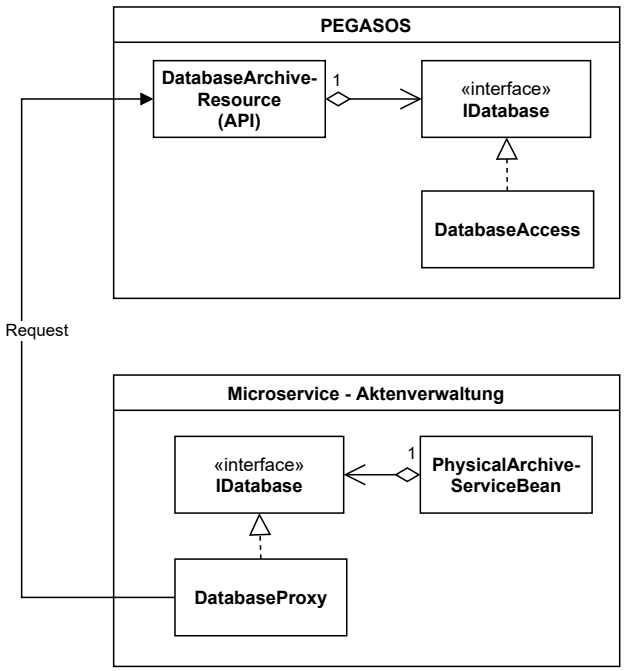

Abbildung 4.6: Die Kommunikation des Microservices Aktenverwaltung, wenn dieser Logiken des Monoliths PEGASOS benötigt.

nichts anderes unternimmt, als den Aufruf an den Microservice weiterzuleiten, vgl. mit Abbildung 4.5. Der Hintergedanke bei dieser Implementierung war, dass bei einer kompletten Migration hin zu Microservices letztlich nur noch Weiterleitungslogik in dem Monolith vorhanden ist und dieser die Funktion übernimmt, die Anfragen der Clients an den jeweiligen Zuständigen Microservice weiterzuleiten oder überhaupt zu ermitteln, welche Microservices in dem Netzwerk zur Verfügung stehen. Dies ermöglichte ebenfalls, dass die Clients nicht verändert werden mussten, mit den Microservices kompatibel bleiben und es kann zu einem späteren Augenblick überlegt werden, ob die Clients die Microservices selbst kontaktieren oder über einen anderen Mechanismus.

Eine ähnlicher Umstand ergab sich mit den Requests, die der Microservice an den Monolith schickte, beispielhaft für den Datenbankzugriff. Auch hierbei war der Vorteil, dass der Monolith als Proxy dienen würde, wenn später die Datenbankzugriffe auf einen eigenen Microservice ausgelagert werden. Allerdings wäre es auch hier zu einem späteren Zeitpunkt möglich gewesen, dass der Microservice der Aktenverwaltung selbst auf den Microservice der Datenbank zugreift, da hierbei nur der Proxy des Microservices durch eine andere Implementierung relativ einfach ausgetauscht werden musste.

Ungeachtet der Umstände, dass geklärt werden musste, wie die Microservices von dem System gefunden werden, nutzten wir den hier vorgestellten Architekturplan als Grundlage für die prototypische Umsetzung. Bezüglich der eingesetzten Technologie, mit der wir den neuen Microservice schrieben, hielten wir es für ratsam bei

dem derzeitigen verwendeten Technologiestack zu bleiben, damit wir uns auf den Migrationsprozess fokussieren konnten und uns nicht auf technologischen Probleme konzentrieren mussten. Dahingehend verwendeten wir für den Microservice die Sprache Java, als Kommunikationsprotokoll REST und der Microservice wurde mittels eines WildFly Servers im Netzwerk zur Verfügung gestellt.

4.3 Ergebnisse der Umsetzung

In diesem Abschnitt beschreiben wir die Ergebnisse und Erfahrungen, die während der Ausführung aufkamen und die wir letztlich in den Protokollen notiert haben. Während dieses gesamten iterativen Prozesses wurden Protokolle angefertigt von denen 24 in diesem Abschnitt ausgewertet werden. Protokoll, die die Analyse des Monoliths betreffen, sind im vorherigen Teil ausgewertet und werden hier nicht weiter berücksichtigt. Diese Umsetzung und Anfertigung der Protokolle geschah in einem Zeitraum von fünf Wochen.

4.3.1 Modulare Abkapselung des PhysicalArchiveServiceBean

Das Auflösen der Referenzen von und zu dem PhysicalArchiveServiceBean nahm insgesamt zehn Arbeitstag in Anspruch und dauerte damit fast die Hälfte der gesamten Migrationszeit. Dabei konnte der geplante Architekturentwurf allerdings komplett in seiner Form umgesetzt werden, sodass die Klasse für eine Migration zu einem Microservice bereitsteht. Während der Umsetzung trafen wir allerdings auf Probleme, die für unser weiteres Vorgehen von Relevanz sind.

Das erste Problem, welches während des Refactorings auftrat, ist die Verwendung von Singletons, die den PhysicalArchiveServiceBean anderen Klassen zur Verfügung stellte. Einerseits, bedingt durch unseren Mangel an technischer Kompetenz bezüglich der Techniken und Frameworks EJB und CDI, mussten wir uns in technische Besonderheiten einarbeiten und die verwendete Funktionalität von PEGASOS und deren Technologie verstehen. Auf der anderen Seite wurde spezieller Code zur Instanziierung verwendet, den wir erst nach langem Debuggen herausfanden und dann durch unsere Abstraktion ersetzen konnten. Viele dieser Probleme sind historisch gewachsen, da wir mitunter Klassen ändern mussten, die schon seit längeren Jahren nicht mehr angepasst wurden sind.

Dass wir Klassen änderten, die innerhalb der Firma schon länger nicht mehr verändert wurden sind, bringt uns zu einer weiteren Problematik. Die Software PEGASOS besitzt Tests, die die Funktionalität der Software überprüft, allerdings nicht auf die Funktionalität der Klassen, die wir gerade änderten. Außerdem existierten keine Tests, die die Funktionalität überprüfen würde, die wir gerade anpassten. Das sorgte bei uns für ein unwohles Gefühl, da wir nicht garantieren konnten, dass wir durch unseren Refactoringvorgang die Integrität der Software beschädigt haben. Eine Annahme, die wir nach dieser Feststellung trafen ist, dass ein solch gravierender Refactoringaufwand durch Tests zwingend überprüft werden sollte, da ansonsten nicht mehr die Integrität der Software sichergestellt ist. Wir entschieden daher, dass wir bei der tatsächlichen Migration für die Proxies und neu erstellten Schnittstellen granulare Tests erstellten, damit wir für diese eine gewisse Stabilität bezüglich aufkommender Fehler garantieren konnten. Außerdem konnten die Unit-Tests schneller ausgeführt

werden, als den ganzen Monolith zu bauen, zur Verfügung zu stellen und anschließend manuell zu testen.

4.3.2 Erstellung des Microservices

Nachdem wir den Monolith gerefactored haben und die Struktur für eine Migration gegeben ist, überprüften und recherchierten wir mehrere technische Pattern, wie wir in unserem Fall am besten die Implementierung aufbauen. Obwohl Vorschläge, siehe Kapitel 2.1.3, zur Umsetzung vorhanden waren, wie z.B. das Strangler-Fig-Application oder Branch-By-Abstraction[35] fiel es uns schwer, aus der Analyse heraus zu sagen, welche Methodik, für unsere Migration, geeignet war. Wir entschieden uns daher, in kleinen iterativen Schritten zu gehen und ggf. nochmal die technische Implementierung zu ändern, wenn wir auf zu große Probleme stoßen.

Das Einrichten des neuen Services als zusätzlichen Wildfly-Server lief problemlos und das finden des jeweiligen Servers lösten wir vorläufig durch das Hinterlegen der internen IP von dem Microservice in dem Monolith PEGASOS. Nachdem der Microservice eingerichtet war fingen wir damit an, die Klasse PhysicalArchiveServiceBean in den Microservice zu verschieben und stellten relativ schnell fest, dass die Komplexität der gesamten Klasse zu hoch war, als dass wir sie mit einmal migrieren konnten. Die vielen Abhängigkeiten der Klasse zu dem Monolith PEGASOS und das Programmieren der gesamten API-REST-Schnittstellen benötigte viel Zeit, in der Zeit konnten wir die Software weder starten noch testen.

Letztlich entschieden wir uns für den Einsatz der Strangler-Fig-Application Technik, in Kombination mit Branch-By-Abstraction und das wir die Klasse Methodenweise iterativ migrieren. In der Abbildung 4.7 ist unsere konkrete Implementierung dargestellt. Der verwendete Proxy in dem Monolith besaß zwei Klassen, von der eine die originale Logik des PhysicalArchiveServiceBean beinhaltete und eine zweite Klasse, die für die Verarbeitung der Logik einen Request an den Microservice schickte. Dadurch konnten wir eine einzelne Methode migrieren, der Proxy verwendet für die schon migrierten Methode den Client, welcher den Request an den Microservice schickte. Für alle anderen Methoden, die noch nicht migriert wurden sind, nutzten wir die noch vorhandene Logik des PhysicalArchiveServiceBean. Nach Abschluss der Migration konnten wir den PhysicalArchiveServiceBean dann aus dem Monolith entfernen, da er nicht mehr benötigt wurde - alle Methoden dieser Klasse wurden durch den PhysicalArchiveClient und dem Microservice abgedeckt. Durch das Strangler-Fig-Pattern konnten wir schneller die Software deployen und testen, sowie auf mögliche Probleme schneller reagieren.

4.3.3 Migration des PhysicalArchiveServiceBean

Die unscheinbar hohe Komplexität der Migration hatte uns gezeigt, dass wir in kleinen überschaubaren Schritten vorgehen sollten. Daher untersuchten wir die vorhandenen Methoden des PhysicalArchiveServiceBean und suchten eine Methode, die klein, überschaubar und nicht viele Abhängigkeiten auf benötigte Fremdlogik enthielt. Eine Methode, die wir für praktikabel fanden, ist die, die eine Akte auf Korrektheit bezüglich den Ablageorten überprüft. Die Implementierung der kleinen Methode in

[35] https://martinfowler.com/bliki/BranchByAbstraction.html

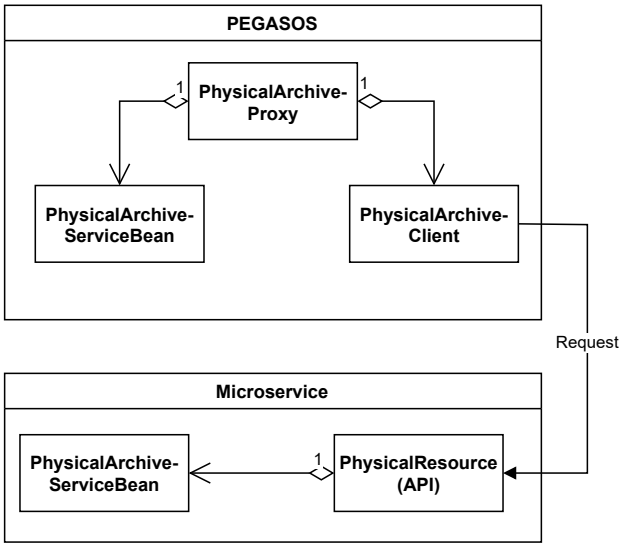

Abbildung 4.7: Den Einsatz des StranglerFigPattern während der Migration.

den Microservice zeigte deutlich schneller Erfolge und die Software PEGASOS war nach diesem Schritt immer noch einsatzbereit und ausführbar. Die Methode besaß allerdings drei Abhängigkeiten, die zu drei neuen Fragestellungen geführt haben.

Die erste Fragestellung war das Thema Logging. In dem Monolith wurde der Error mitgeschrieben, wenn die Akte fehlerhaft war. Mit der Implementierung im Microservice stellte sich hierbei die Frage, muss die Nachricht des Fehlers wieder zurück in den Monolith oder behandelt der Microservice das Logging für sich selbst. Wir entschieden uns in unserer Implementierung dafür, dass wir die Nachricht wieder an den Monolith zurückschickten und dort eine Schnittstelle für das Logging einrichteten, damit diese die Nachricht empfangen kann. Allerdings erkannten wir demgegenüber, dass es viele Gründe gab, warum wir das Logging nur innerhalb des Microservices betrachten sollten, weil dieser auch für die Funktionalität zuständig war.

Die zweite Fragestellung betraf eine ähnliche Problematik und zwar die Sessionid des jeweiligen Nutzers. Über die Sessionid war gespeichert, dass er sich bereits authentifiziert hat oder welche Id er besitzt, die zum Abfragen für bestimmte Funktionalitäten verwendet wurde. Ähnlich wie der zuvor genannten Loggingproblematik erkannten wir, dass dies eine Thematik ist, welches nicht nur diesen Microservices sondern generell für alle Microservices von Bedeutung ist. Wir entschieden uns hier dafür, dass das Handling der Session in dem Monolith PEGASOS bleibt und als Parameter mit an den Microservice übergeben wird.

Die dritte Fragestellung, die hierbei aufkam, ist die Verwendung von Utilityklassen. Die Methode verwendete eine Funktionalität, dass der Anzeigetext einer bestimmten Akte entweder Standardmäßig oder mittels der Sessionid des Nutzers ermittelt wird. Unser erster Ansatz, die Funktionalität zur Ermittlung des Anzeigetextes in den Microservice zu kopieren schlug fehl bzw. fanden wir im Nachhinein unbrauchbar, weil dafür noch weitere Funktionalität angepasst werden müsste. Der Anzeigetext war abhängig von der Sprache, womit der Nutzer angemeldet ist, welche an der Sessionid

gespeichert ist. Obwohl es nur eine kleine Funktionalität war sorgt dieser Umstand dazu, dass immer mehr Funktionalität im Microservice gelangt, weil Abhängigkeiten benötigt werden. Nach unserer vorher genannten Devise migrieren wir in kleinen Schritten und implementierten eine einfachere Lösung, indem wir die Funktionalität der Utilityklasse im Monolith behielten.

An dieser Stelle stellten wir fest, dass die Migration durch die neuen Fragestellungen doch deutlich komplexer werden kann als angedacht, da die migrierte Methode nur knapp 20 Zeilen lang war und lediglich eine Überprüfung der Akte darstellt. Mittlerweile sind fast vier Arbeitswochen vergangen worin wir lediglich die Migration einer einzelnen Methode erreicht haben. Daher kalkulierten wir damit, dass wir die komplette Migration nicht mehr erreichen würden, bevor wir auf Probleme stoßen, die einen neuen Ansatz benötigen. Um die Probleme so früh wie möglich zu erkennen, migrierten wir als nächstes eine Methode, die eine komplexere Logik und andere Abhängigkeiten zum Monolithen benötigte. Die zu nächste zu migrierende Methode war dafür zuständig, dass der Ablageort einer Akte geändert wird.

Bei der Migration dieser Methode stellten wir einen Lerneffekt fest. Das Erstellen der REST-Schnittstellen, das Anlegen der notwendigen Datencontainer sowie das Erstellen der Tests geht deutlich schneller als bei der vorherigen Methode, obwohl wir hierbei deutlich mehr Abhängigkeiten und komplexere Logik vorhanden haben. Nach knapp einer weiteren Woche war die Migration fertiggestellt und wir sind auf zwei primäre neue Problematiken gestoßen.

Das erste Problem war der Einsatz des Caches. Im Monolith wurden die Akten aus dem Cache geholt und nach der Aktualisierung des Ablageortes wurde auch entsprechend der Cache informiert, dass diese Daten aktualisiert wurden sind. Für uns stellte sich dabei die Frage, wer dafür konkret zuständig ist und wo die Logik dafür liegen sollte. Aus unserer Sicht wäre der Microservice dafür zuständig, da dieser auch die Daten dafür verarbeitet, allerdings ist die Logik des Caches sehr stark an die Logik des Monoliths PEGASOS gebunden, sodass die Extraktion der benötigten Funktionalität ein großes Refactoring vonnöten macht. Daher blieb die Funktionalität des Caches im Monolith und die Aktualisierung des Caches wurde über eine weitere REST-Schnittstelle zur Verfügung gestellt, die der Microservice nutzt.

Die Datenbankzugriffe sind unser zweites Problem, welches durch die Migration der Methode aufkam. Bei einer Migration zu Microservices besitzt jeder Service eine eigene Datenbank, die er für seine Funktionalität benötigt. Wir waren jetzt ebenfalls an dem Punkt, dass wir die monolithische Datenbank aufteilen hätten müssen und die notwendigen Daten in den Microservice migrieren und neu strukturieren müssten. Aus einer kurzen Analyse der Datenbank von PEGASOS wurde schnell deutlich, dass die Aufspaltung einen sehr hohen Aufwand zur Folge hätte und wir die Datenbank im Monolith beließen. Jedoch sollten wir für eine vollumfängliche Migration dieses Thema zuerst angehen, da wir in diesem Bereich den höchsten Bedarf und vermutlich auch den größten Aufwand sehen.

Während der Migration dieser beiden Methoden stellten wir weitere Punkte fest, die bei einer Migration berücksichtigt werden müssen. Der erste Punkt ist das Thema Sicherheit. PEGASOS wird im Gesundheitssektor eingesetzt und besitzt hochsensible persönliche Daten, die entsprechend gegenüber Angreifern gesichert werden müssen.

Der Einsatz von Microservices bringt dabei ein neues Risiko mit sich, da aus einem einzelnen Request von dem Client an den Monolith sehr viele Requests werden, die alle potentiell durch Angreifer abgefangen und entschlüsselt werden können. Außerdem ist nicht jeder Nutzer, der an dem System eingeloggt ist, berechtigt alle Daten und Informationen zu sehen. Diese Rechteüberprüfung muss auch über die Microservices aktiv sein, sodass keine Daten abgefragt werden können, für die ein Nutzer gar keine Rechte besitzt.

Als zweites ist das Thema Performance von PEGASOS von Bedeutung. Aus internen Berichten ist häufiger der Server von Performanceproblemen betroffen, da zum Laden der Daten viel Zeit benötigt wird. Durch Microservices könnte sich das Problem hierbei verschärfen, da aus dem einen genutzten Requests sehr viele Requests werden, die allein von der Anzahl her mehr Bandbreite im Netzwerk belegen, als der einzelne Requests zum Monolith. Anderseits können die Microservices individuell skaliert werden, sodass genau die Services mehr Rechenleistung und Instanzen bekommen, die stark belastet werden. Dadurch könnte sich die Performance von PEGASOS deutlich verbessern.

Ein letzter wichtiger Punkt, der uns während der Migration auffiel, ist, dass das nicht Vorhandensein von Tests unsere Arbeit deutlich erschwert bzw. uns in einer Unsicherheit zurücklässt. Durch das Refactoring im Monolith und dem Verschieben von Klassen und Methoden könnten sich infolge dessen neue Fehler eingeschlichen haben. Tests könnten hierbei eine Absicherung schaffen, sodass dadurch sichergestellt ist, dass die Migration nicht zu einem Fehler in der Software führt.

An dieser Stelle stellten wir fest, dass immer mehr Probleme und Fragen auftreten, die eine hohe Komplexität und Aufwand aufweisen und dessen Lösung eine intensive Analyse benötigt. Insbesondere das Thema der Datenbank, das Handling der Session eines Nutzers oder die Verwendung von Utilityklassen erfordert eine genaue Analyse, damit daraus eine konkrete Handlungsanweisung für eine weitere Migration formuliert werden kann.

4.4 Zusammenfassung

In diesem Kapitel beschrieben wir die durchgeführten Schritte der Vorgehensweise und deren Erfahrungen die wir daraus gewonnen haben. Zuerst wurde versucht, durch eine Analyse des Monoliths, ein geeignetes Modul für die Migration herauszufinden. Die Analyse gestaltete sich als komplizierter als gedacht, sodass letztlich, durch eine Absprache mit den Entwicklern, die Klasse PhysicalArchiveServiceBean in einen Microservice migriert wird. Der zweite Schritt, dass die Klasse im Monolith von der restlichen Logik abgekapselt wird konnte dahingehend erfolgreich durchgeführt werden. Anschließend wurde diese Klasse in einen neu erstellten Microservice migriert. Während dieses Prozesses wurde mehrere Probleme deutlich, die zu einer Abänderung der Vorgehensweise führten, wodurch am Ende nicht die gesamte Klasse sondern nur drei Methoden dieser migriert wurde.

5. Auswertung und Diskussion

Die gewonnenen Erkenntnisse aus der Umsetzung fassen wir in diesem Kapitel zusammen und unterteilen dieses Kapitel dabei in drei Abschnitte. In dem ersten Abschnitt werden die gelernten Erfahrungen und Probleme aus der Umsetzung zusammengefasst und in allgemeingültigen Lessons Learned formuliert. Im zweiten Abschnitt geben wir einen Ausblick, auf die weitere Migration zu Microservices von PEGASOS. Der dritte Abschnitt beschäftigt sich mit der ausgewählten Methodik, inwieweit diese für den gewollten Zweck geeignet war und welche Verbesserungspotentiale sie aufweist.

5.1 Erfahrungen und Ergebnisse der Durchführung

Während der Umsetzung wurden durch die Protokolle Probleme und aufkommende Fragestellungen aufgeschrieben, die wir in diesem Abschnitt zu fünf Lessons Learned zusammengefasst haben.

E1 Aufteilung des Monoliths ist die komplexeste Aufgabe

E2 Die Migration in kleinen Schritten mit einem agilen Vorgehen durchführen

E3 Das Entwicklungsteam sollte Praxiserfahrungen über Microservices mitbringen

E4 Vor der Migration müssen Tests zur Überprüfung vorhanden sein

E5 Abhängigkeiten der Module untereinander sollten vorher im Monolith untersucht werden

Darin formulieren wir das Problem, auf das wir gestoßen sind und wie wir dieses in einer weiteren Migration besser lösen würden. Ebenfalls vergleichen wir unsere gewonnenen Erkenntnisse mit vergleichbaren Studien und überprüfen, inwieweit wir Übereinstimmung zu unseren Lessons Learned ermitteln können. Die Literatur, die wir für den Vergleich nutzen, entstammen aus der durchgeführten Literaturrecherche, siehe Kapitel 3.1.

Aufteilung des Monoliths ist die komplexeste Aufgabe

Aus unserer Sicht ist die Analyse des Monoliths und deren schematische Aufteilung einer der wichtigsten und gleichzeitig schwierigsten Aufgabe. Zu wissen, welche Klassen und Komponenten wir zu einem System zusammenfassen und welche wir voneinander trennen, ist einerseits schwierig und bringt immer die Gefahr mit, dass während oder auch erst nach der Implementierung auffällt, dass die Abgrenzungen der Microservices nicht gut gewählt sind. Diese Aufgabe sollte daher nicht aus einer rein theoretischen Seite betrachtet werden, sondern immer unter der Berücksichtigung der jeweiligen Software sowie deren Entwickler. Ebenfalls ist ein automatisierter Ansatz empfehlenswert, allerdings muss dafür ein passendes Tool für die Software zur Verfügung stehen. Daher schlagen wir für eine zukünftige Migration vor, dass zuerst ein grober Überblick über die Softwarestruktur eingeholt wird und die Entwickler dazu befragt werden. Die Entwickler wissen aus Erfahrungen, welche Teile der Software zusammengehören und welche getrennt voneinander behandelt werden müssen.

Ähnliche Erfahrungen erwähnen auch Kalske et al. [2018] die sagen, dass die Entwickler am Besten eine Vorstellung für die Aufspaltung des Monoliths in Module haben und das die technische Aufspaltung des Monoliths sehr viel Zeit in Anspruch nehmen kann. Ebenso fanden Taibi et al. [2017] in einer Umfrage heraus, dass unter den Teilnehmer die Aufspaltung des Monoliths in einzelne Module die größte Herausforderung, während des Migrationsprozesses ist. In unserer Durchführung hatten wir Probleme, mittels eines Tools die Analyse und Migration zu unterstützen, welches sich mit der Umfrage von Velepucha und Flores [2021] deckt, worin die Auswahl von automatisierten Tools problematisch war.

Die Migration in kleinen Schritten mit einem agilen Vorgehen durchführen

Während der Migration zu Microservices können viele Probleme auftreten, die vorher einem noch nicht bewusst waren. Insbesondere dann, wenn die technische Expertise im Entwicklungsteam im Bereich Microservices und deren Migration von einem Monolith nur geringfügig vorhanden ist. Ein agiles Vorgehen eignet sich hierbei, damit nach und nach die Expertise aufgebaut werden kann und auf aufkommende Fehler und Probleme schneller reagiert werden kann. Ebenso muss die Software an Kunden auslieferbar bleiben. Daher sollte in kleinen Schritten vorgegangen werden, sodass nach dem Auftreten eines Problems schnell zu einem vorherigen Stand zurückgegangen werden kann, der funktionsfähig ist. Ebenfalls können somit im Produktiveinsatz die ersten Funktionalitäten, die über Microservices zur Verfügung stehen, bereitsgestellt und die ersten Erfahrungen damit gesammelt werden. Je komplexer die Domaine ist desto kleiner sollten die Schritte sein, mit denen migriert wird.

Ein iteratives Vorgehen in einem kleinen Team, insbesondere wenn kaum Erfahrungen vorhanden sind erwähnen Carrasco et al. [2018] in ihrer durchgeführten Literaturrecherche. Dessen Resultat zeigt, dass ein unerfahrenes Team zu höheren Kosten und einer längeren Entwicklungszeit führt, insbesondere im Umgang mit verteilten Systemen. Ebenfalls erwähnt Kalske et al. [2018], dass in kleinen Schritten die Software auf Microservices migriert werden soll, allein wegen der Komplexität der Architektur. Unser Lessons Learned, dass ein iteratives Vorgehen gewählt werden soll, löst dabei einerseits das Problem der wenigen Erfahrungen im Team, da sie

über die Zeit erlernt wird und anderseits dass der Komplexität, durch die Zerlegung des Vorgangs in einzelne iterative Schritte. Fritzsch et al. [2019a] zeigen durch ihre Umfragen, dass die Mehrheit der Unternehmen Scrum[36] als Projektmanagement nutzten und damit ein iteratives und agiles Vorgehen.

Das Entwicklungsteam sollte Praxiserfahrungen über Microservices mitbringen

Die Migration wurde von einem Team durchgeführt, welches noch keine Erfahrung mit einer vergleichbaren Migration zu Microservices hatte. Fehlende Praxiserfahrung erschwerten daher den Prozess, da wir noch kein Wissen über bestimmte Besonderheiten von Microservices hatten und erst während der Umsetzung Probleme feststellten, woran wir in der Vorbereitung nicht gedacht hatten. Wir empfehlen daher die Migration durchzuführen, wenn das Team bereits Erfahrung mit Microservices hat, Erfahrungen z.B. durch Schulungen oder Prototyping sammelt oder sich einen Experten als Fachkompetenz hinzuholt.

Während wir keine konkrete vergleichbare Studie gefunden haben, worin ein unerfahrenes Team von einem Monolith auf Microservices migriert oder dieses als Problem ansah, erwähnen Velepucha und Flores [2021] in diesem Zusammenhang, dass die Entwickler eine gewisse Lernbereitschaft mitbringen müssen, da durch die Umstellung auf Microservices auch die Arbeit damit anders wird. Auch Fritzsch et al. [2019a] bringt in dem Zusammenhang an, dass die technischen Herausforderungen bei der Migration weniger die Herausforderung ist mit der Begründung, dass ein Experte für dieses Gebiet eingestellt werden kann, der die nötige technische Kompetenz mitbringt. Als letztes nennen Knoche und Hasselbring [2019], dass die Gründe gegen Microservices, das nicht Vorhandensein von Erfahrungen sind. Dies deckt sich mit unserer Einschätzung, dass vor der Migration ein gewisses technisches Wissen in dem Team vorhanden sein sollte.

Vor der Migration müssen Tests zur Überprüfung vorhanden sein

Als wir in unserem Fall die Funktionalität aus dem Monolith in den Microservice verschoben haben konnte wir keinen objektiven Nachweis, beispielsweise durch eine Testabdeckung, vorweisen, um die Integrität der Software nachzuweisen. Wir konnten zwar einen erfolgreichen manuellen Test durchführen, allerdings sind manuelle Tests für eine große Software nicht mehr anwendbar. Daher sollten mindestens für die Funktionalitäten, die in den Microservice verschoben werden, Tests bereitstehen, mit denen nach Abschluss der Migration die Integrität nachgewiesen werden kann.

Kalske et al. [2018] benennen das Vorhandensein von Tests. einer Continuous Integration und Continuous Devliery Pipeline als essentiell, weil ansonsten die Funktionalitäten und die vielen Services nicht mehr überprüft werden können. Ebenso Fritzsch et al. [2019a] wechselten Firmen von einer manuellen Testmethodik zu einer automatisierten mit einem positiven Einfluss auf die Anzahl der Fehler im System. Demgegenüber zeigen Knoche und Hasselbring [2019] auf, dass viele Firmen auch nach Einsatz von Microservices bei manuellen Tests bleiben. Unbeantwortet bleibt hierbei die Fragen, inwieweit dies einen negativen oder positiven Einfluss auf die Software hat. Durch unser gemachtes Lessons Learned schließen wir uns der Auffassung an, dass automatisierte Tests wichtig für die Integrität der Software sind.

[36] https://www.scrum.org/

Abhängigkeiten der Module untereinander sollten vorher im Monolith untersucht werden

Funktionalitäten, die in einen Microservice verschoben werden, besitzen keine direkte Kopplung auf andere Systemkomponenten des Monoliths und können daher nicht auf diese direkt zugreifen. Klassische Kopplungen sind dabei das Logging von Events, das Handling einer Session von Nutzern oder die Verwendung der Datenbank, die allesamt für alle Microservices relevant sein könnten. Daher empfehlen wir, dass bevor die Migration durchgeführt wird, die Abhängigkeiten des zu migrierenden Moduls untersucht werden und wie diese Abhängigkeiten in dem Kontext der Microservices behandelt werden. Unter Umständen führt das zu einer Problematik, die gesondert betrachtet werden muss.

Kalske et al. [2018] erwähnt in seiner Arbeit die Problematik des Loggings, welche eine zusätzliche Herausforderung für die Migration darstellt. Ebenso hatten laut Fritzsch et al. [2019a] vier Systeme Probleme die Abhängigkeiten zu fremden Bibliotheken oder dem eigenen Monolith aufzulösen und in der Microservicearchitektur einzubetten. Auf diese Problematik zielt unser Lessons Learned ab, dass zuerst die Abhängigkeiten untersucht werden, bevor bestimmte Probleme mit fremden oder eigenen Systemen eventuell auftreten.

5.2 Ausblick auf die weitere Migration von PEGASOS

Auf der Grundlage unserer Lessons Learned formulieren wir in diesem Abschnitt eine Handlungsempfehlung für die Software PEGASOS.

Zuerst sollte der Fokus bei der Migration auf eine erneute Analyse des Monoliths und deren modulare Aufteilung liegen. Bei der Analyse sollten automatische Tools evaluiert werden, inwieweit diese einsetzbar sind, um Metriken zu ermitteln, die für die modulare Aufteilung verwendet werden können. Diese Metriken sind Grundlage für eine Diskussion mit den Entwicklern, damit diese ihre Erfahrungen mit der Software in die Analyse mit einfließen lassen können. Anschließend sollte überlegt werden, in welches Modul welche Funktionalität gehört und wie die Abhängigkeiten der Module untereinander sind. Der Abhängigkeitsgraph sollte dabei keine zyklischen Referenzen aufweisen. Bevor überhaupt ein Microservice eingesetzt werden kann, muss die modulare Aufteilung des Monoliths umgesetzt werden. Die Aufteilung sollte in kleinen Schritten durchgeführt und jeder mit Tests validiert werden, damit durch diese Neustrukturierung der Architektur keine neuen Fehler eingebaut werden.

Nachdem die Modularisierung des Monoliths abgeschlossen ist stellen wir an dieser Stelle eine Hypothese, über die Verwendung von Microservices für PEGASOS, auf. Wir behaupten, dass eine Modularisierung der Software bereits zu einer Verbesserung der Probleme mit PEGASOS führen würde und der Mehraufwand für die Implementierung von Microservices sich nicht rentieren wird. Die Behauptung können wir an dieser Stelle nicht validieren, allerdings ist die Modularisierung von PEGASOS eine Voraussetzung, bevor überhaupt zu Microservices migriert wird. Daher empfehlen wir, dass zuerst der Monolith PEGASOS modular aufgebaut wird und im Produktiveinsatz eingesetzt wird. Nach einer Evaluierungsperiode ließe sich feststellen, inwieweit der modular aufgebaute Monolith bereits zu einer Verbesserung geführt hat. Erst

nach der Evaluierung sollte diskutiert werden, inwieweit sich Microservices für die Software als nützlich erweisen. Ähnlichkeiten zu unser genannten Hypothese finden wir auch bei [Carrasco et al., 2018], welcher erwähnt, dass die Vor- und Nachteile von Microservices und deren Alternativen untersucht werden sollten. Unter Umständen ist der organisatorische Mehraufwand höher, als der Nutzen aus den Microservices.

Sollte die Software PEGASOS auf Microservices migriert werden schauen wir zuerst auf das Entwicklungsteam und deren Kompetenz in Bezug auf Microservices. Je geringer die Kompetenz desto eher sollte entweder jemand mit der nötigen Kompetenz hinzugeholt oder das Team in der Thematik zuerst geschult werden. Bei der Durchführung werden dann zuerst die Module migriert, die am wenigsten mit dem Monolith gekoppelt sind und die nicht systemrelevante oder kritische Funktionalitäten beinhalten. Aufkommende Probleme sind dadurch weitaus unkritischer und es können weitere Erfahrungen mit PEGASOS und Microservices erlangt werden, die für die weitere Migration von Vorteil sind.

5.3 Auswertung der Methodik der Migration

Die Methodik, siehe Kapital 3, definiert sich in den drei Hauptschritten, dass zuerst der Monolith analysiert wird, anschließend wird ein Architekturplan über die modulare Aufteilung erstellt, welcher im letzten Schritt, bei der Umsetzung, angewendet wird. In den folgenden Abschnitten formulieren wir Probleme, aber auch positive Aspekte, die wir mit unserer angewandten Methodik feststellten.

Analyse des Monoliths

Der erste Schritt der gewählten Methodik zur Migration von PEGASOS ist die Analyse des Monoliths und deren derzeitigen Klassenstruktur. Die eigentliche Planung, dass wir ein komplettes Klassendiagramm von PEGASOS erstellen, fanden wir im Nachhinein als einen ungeeigneten Schritt, da wir wegen der Größe und Komplexität der Software schlicht keine Erkenntnisse für den darauffolgenden Schritt der modularen Aufteilung ableiten konnten. Außerdem fehlte eine konkrete Methodik, wie wir aus dem Klassendiagramm von PEGASOS eine geeignete Analyse durchführen, die später bei der modularen Aufteilung des Monoliths hilft.

Während der Durchführung fanden wir es geeigneter, speziell auf die Architektur von PEGASOS zu schauen, damit wir daraus eine geeignete Methodik für die Analyse definieren können. In unserem Fall nutzten wir letztlich die Erfahrungen der Entwickler, damit wir ein geeignetes Modul zur Migration finden können, in unserem Fall die Aktenverwaltung. Doch auch diese Aktenverwaltung war zu komplex, sodass wir nur eine einzelne Klasse migrieren, anstelle eines kompletten Moduls, wie wir dies ursprünglich geplant hatten.

Für die Formulierung einer neuen Methodik würden wir Vorwissen über die aktuelle Software stärker mit in die Methodik einfließen lassen. Daher schlagen wir vor, dass zuerst mit den Entwicklern geredet wird, damit wir deren Erfahrungen bei der Arbeit mit der Software einfließen lassen und eine dafür angepasste Analyse definieren können.

Schematische modulare Aufteilung

Da der vorherige Schritt, die Analyse des Monoliths, nicht in der geplanten Form ausgeführt wurde, konnte auch dieser Schritt nicht planungsmäßig umgesetzt werden. Die Anpassungen an diesem Schritt hielten sich allerdings in Grenzen, sodass nur eine einzelne Klasse, anstelle eines geplanten kompletten Moduls von dem Monolith getrennt und in den Microservice verschoben wurde. Das Resultat, dass wir einen Architekturplan für die konkrete Umsetzung erstellt haben, stellte sich bei den darauffolgenden als nützlich heraus, da wir eine bessere Vorstellung von der letztlich aussehenden Architektur hatten. Allerdings muss hier berücksichtigt werden, dass es sich nur um eine einzelne Klasse und nicht um ein komplette Modul gehandelt hat, welches eine eventuell weitaus größere Komplexität und Probleme mit sich gebracht hätte.

Prototypische Umsetzung

Der letzte Schritt der Methodik, die technische Umsetzung der Migration, stellte sich durch den iterativen Prozess als deutlich machbarer heraus, im Gegensatz zu den vorherigen Schritten der Methodik. Dadurch konnten wir während der Arbeit mit PEGASOS auf Probleme reagieren und anschließend unsere Vorgehensweise anpassen, ohne das wir den vorher definierten Ablauf ändern mussten. Ebenfalls ist das Erstellen der Protokolle, während des iterativen Prozess eine gute Möglichkeit, Probleme und Fragestellungen zu formulieren, die in einer anschließenden Lesson Learned ausgewertet werden können.

Ähnlich zu dem vorherigen Schritt mussten wir diesen Vorgang anpassen, da wir kein komplettes Modul zur Migration Verfügung hatten, sondern lediglich eine einzelne Klasse, die auf einen Microservice migriert wird. Der eigentliche Ablauf der Umsetzung wurde davon allerdings nicht weiter gestört und musste entsprechend nicht angepasst werden.

5.4 Threats of Validity

In diesem Abschnitt gehen wir auf Schwachstellen und Problematik bezüglich unserer Durchführung sowie der Auswertung ein und wie wir diese entweder nicht vermeiden konnten oder in zukünftigen Arbeit vermeiden würden.

Die Durchführung bezog sich schlussendlich nur auf eine einzelne Klasse, die auf einen Microservice migriert wurde, weswegen nur an einem sehr kleinen Teil der Software die Migration vollzogen wurde. Dadurch sind die daraus gewonnenen Erkenntnisse mit einer gewissen Vorsicht zu betrachten, weil das Gesamtsystem an sich nur sehr wenig verändert wurde, wodurch eventuell viele Aspekte nicht betrachtet wurden sind. Der Grund von dem geringen Umfang lag an dem Umstand, dass wir als Außenstehende die Migration durchführten und somit keine Erfahrungen mit der vorhanden Codebasis besaßen, was die Migration an sich deutlich erschwerte. Wir halten es für eine zukünftige Studie über Microservices als sinnvoller, wenn die technische Migration an sich von erfahrenen Entwickler durchgeführt wird, die bereits als Entwickler in dem Unternehmen oder an der Software arbeiten.

Ebenfalls unter dem Umstand, dass nur eine einzelne Klasse migriert wurde, konnten einige wichtige Aspekte nicht umgesetzt und validiert werden. Davon betroffen ist die Aufspaltung der monolithischen Datenbank von PEGASOS, das Monitoring oder die Netzwerksicherheit der Microservices. Probleme in diesen Bereichen können allerdings erst dann untersucht werden, wenn mehrere Module als Microservice umgesetzt wurden sind und in einem Netzwerk aktiv sind. Dadurch sind unsere Ergebnisse nur auf den Kontext einer einzelnen Klasse bezogen und nicht auf das Gesamtsystem von mehreren Microservices. In unserem Beispiel konnten wir das nicht untersuchen, weil der Aufwand als Außenstehender, die Software in einem großen Umfang auf Microservices zu migrieren, zu hoch ist, als dass wir dies umsetzen konnten. Auch für diesen Punkt halten es wir für ratsamer die Migration nicht eigenständig durchzuführen, sondern diese von erfahrenen Entwickler der jeweiligen Software umzusetzen lassen.

Die Entwicklung von Microservices geht meistens mit einer Veränderung der organisatorischen Struktur her. Wir führten die Migration eigenständig durch, womit wir die vorzufindende Struktur der Firma NEXUS/MARABU nicht mit einbezogen und somit konnten wir bezüglich den organisatorischen Herausforderungen, die mit einer Migration einhergehen, nicht beurteilen oder herausfinden. Unsere Ergebnisse könnten sich verändern, wenn die technische Umsetzung innerhalb der Struktur und des Projektmanagements von PEGASOS stattgefunden hätte.

Zusammengefasst lässt sich sagen, dass unsere Umsetzung die Schwachstelle besitzt, dass wir eigenständig die Migration durchführten und diese nicht durch die Entwickler von NEXUS/MARABU geschah. Ebenfalls wurde die Organisationsstruktur sowie die Produktivumgebungen nicht in die Betrachtung mit einbezogen, wodurch Herausforderungen bezüglich der Teamstrukturen, der Kommunikation innerhalb und zwischen den Teams sowie der Projektleiter nicht berücksichtigt wurden sind. Wir empfehlen für zukünftige Forschungen in diesem Gebiet, dass die Migration von den Entwickler des Systems durchgeführt wird und die Organisation der Firma sowie das Projektmanagement mit einbezogen werden, damit ebenfalls organisatorische Herausforderungen untersucht werden können. Dadurch sehen wir eine Steigerung in der Aussagekraft der Ergebnisse sowie eine größere Bandbreite, von zu ermittelnden Problemstellen, die mit einer Migration auf Microservices einhergehen.

5.5 Zusammenfassung

In diesem Kapitel nutzten wir die Ergebnisse der Durchführung, um mit diesen fünf Lessons Learned zu formulieren. Diese fünf Lessons Learned verglichen wir mit der gefundenen Literatur aus der Literaturrecherche und verglichen die Ergebnisse auf Gemeinsamkeiten und Übereinstimmungen. Darüber hinaus gaben wir einen Ausblick auf die weitere Migration von PEGASOS und formulierten eine Handlungsempfehlung für den zukünftigen Weg auf Microservices. Abschließend formulierten wir mögliche Schwachstellen unserer Methodik und formulierten Threats of Validity, die wir bezüglich unseren Ergebnissen und Vorgehensweise sehen.

6. Fazit

In diesem letzten Kapitel fassen wir unser Vorgehen sowie unsere Methodik zusammen, beschreiben die durchgeführte Umsetzung und fassen die Ergebnisse dieser Arbeit zusammen. Darüber hinaus erfolgt die Beschreibung der Schwachstellen und kritischen Aspekte unserer Arbeit. Zum Abschluss geben wir einen Ausblick auf zukünftige Forschungsarbeiten.

6.1 Zusammenfassung

In dieser Arbeit führten wir einen Reengineeringsprozess von einem Monolith auf Microservices durch, anhand des Fallbeispieles der Software PEGASOS. Ziel dieses Prozesses war es, dass wir Probleme und Erfahrungen, die mit einer Migration auf Microservices aufkommen, herausfinden. Dafür formulierten wir, auf der Grundlage einer durchgeführten Literaturrecherche, zuerst eine Methodik, mit der wir den Monolith PEGASOS auf Microservices migrierten. Die Methodik war die Grundlage für die technische Umsetzung sowie für die Sicherstellung, dass wir verwertbare Ergebnisse, aus der durchgeführten Migration, bekommen.

Mit der Umsetzung konnten wir fünf primäre Erkenntnisse ermitteln, die bei einer Migration von einem Monolith zu Microservices von Bedeutung sind. Die erste Erkenntnis ist, dass die Analyse des Monoliths und deren modulare Aufteilung einer der wichtigsten und gleichzeitig schwierigsten Aufgabe bei einer Migration ist. Zweitens, die Migration sollte in kleinen Schritten und agil geplant werden, damit auf aufkommende Probleme reagiert wird und die Software bei der Migration in einem ausführbaren Zustand bleibt. Als dritte Erkenntnis zeigten wir, dass eine hohe Praxiserfahrung über Microservices in dem Entwicklungsteam vorhanden sein sollte. Unsere vierte Erkenntnis beschreibt die Notwendigkeit einer Testabdeckung der Software, damit die Integrität sichergestellt bleibt. In der fünften und letzten Erkenntnis dieser Arbeit wiesen wir nach, dass die Kopplungen innerhalb des Monoliths genauer untersucht werden müssen, die eine Relevanz für viele oder alle Microservices haben. Darunter fallen z.B. Datenbankzugriffe, Logging oder das Thema der Netzwerksicherheit.

Die gewonnenen Erkenntnisse wurden mit Umfragen von Firmen verglichen, die ebenfalls einen Monolith auf Microservices migrierten. Hieraus ergab sich, dass die Erkenntnisse E1, E2 und E4 ebenfalls in vergleichbaren Studien gewonnen wurde, die wir dadurch mit unserer Arbeit bestätigen konnten. Die Erkenntnisse E3 und E5 ergaben weniger Übereinstimmung mit der Literatur und wir konnten nur vereinzelt Zusammenhänge erkennen.

Bezüglich unseren Erkenntnissen empfahlen wir der Firma NEXUS/MARABU für die Migration des System PEGASOS auf Microservices, sich auf die modulare Abkapselungen der einzelnen Domains im Monolith zu konzentrieren. Erst nach Abschluss des Refactorings des Monoliths auf die modularen Domains, empfehlen wir auf Microservices zu migrieren, da es sich an diesem Punkt besser evaluieren lässt, inwieweit Microservices eine Lösung für die derzeitigen Probleme von PEGASOS ist. Außerdem ist ein modularer Monolith die technische Voraussetzung für die Migration auf Microservices.

Die Schwachstellen unserer Arbeit lagen unter anderem darin, dass wir die Migration eigenständig und ohne technische Unterstützung der Entwickler, des Systems PEGASOS, durchführten. Einerseits konnten wir damit nicht die organisatorischen Probleme und Herausforderungen untersuchen und anderseits mussten wir uns zuerst mit dem System auseinandersetzten, wodurch wir das vorgenommene Ziel der Umsetzung nicht erreichten. Unsere Erkenntnisse könnten sich bei einer erneuten Untersuchung der Migration verändern, wenn eine höhere technische Kompetenz des Fallbeispiels vorhanden ist und die Projekt- und Teamstruktur des jeweiligen Unternehmens mit einbezogen wird. Dahingehend sehen wir eine höhere Qualität der Ergebnisse, wenn die Entwickler sowie das Projektmanagement mit in die wissenschaftliche Untersuchung einbezogen werden.

6.2 Zukünftige Arbeiten

Wir haben in unserer Arbeit die Migration an einem Modul in dem System PEGASOS durchgeführt. Weiterführende Forschung könnte in dem Themengebiet der monolithischen Datenbank und dessen Aufspaltung erfolgen, welches wir in unserem Fallbeispiel nicht untersucht haben. Im Hinblick auf die Datenbankthematik sehen wir ein Potential, dass dieses, im Zusammenhang mit unseren Ergebnisse, zu neuen Erkenntnissen führen kann. Darüber hinaus kann es auch sinnvoll sein, weitere Problematiken zu behandeln, die nicht auf einen einzelnen Microservice bezogen sind, sondern viele Microservices im System betreffen. Dazu gehört die Thematik des Monitorings, Loggings oder der Netzwerksicherheit, bezüglich der migrierten Microservices.

Zukünftige Forschung könnten an unseren Ergebnissen und Lessons Learned anknüpfen, indem unsere Erfahrungen genutzt werden, um ein- und mehrere Module eines Monoliths auf Microservices zu migrieren. Dahingehend kann überprüft werden, inwieweit unsere Erkenntnisse auf andere Fallbeispiele übertragen werden können oder eine Grundlage für eine größer angelegte Migration sind.

Eine Frage, die noch weiterer empirischer Untersuchung bedarf, ist außerdem das Herausfinden von organisatorischen Herausforderungen, die mit einer Migration einhergehen. Wir führten die Migration ohne Einbindung des Entwicklungsteam oder des

Projektmanagements durch, sodass organisatorische Hürden nicht aufgetreten sind. Damit diese Frage eindeutig beantwortet werden kann, bedarf es einer Untersuchung, die sich auf die organisatorischen Veränderungen der Firma bezieht, wenn diese eine Migration durchführen.

Die Migration von einem Monolith ist ein hochkomplexes Thema, welches gleichzeitig wenig durch wissenschaftliche Studien untersucht wurde und viele offene Fragen besitzt. Unsere Arbeit trägt einen Beitrag für einen Erfahrungsbericht über Probleme, Herausforderungen und Lösungen, die wir aus unseren Erkenntnissen ableiten konnten, bei. Wir hoffen, dass diese Arbeit eine Unterstützung für weitere Forschungsarbeiten ist, damit das Thema der Migration näher untersucht werden kann.

Anhang

1. Protokolle

Literaturverzeichnis

João Franscisco Almeida und António Rito Silva. Monolith migration complexity tuning through the application of microservices patterns. In Anton Jansen, Ivano Malavolta, Henry Muccini, Ipek Ozkaya, und Olaf Zimmermann, editors, *Software Architecture*, volume 12292 of *Springer eBook Collection*, pages 39–54. Springer International Publishing and Imprint Springer, 2020. (zitiert auf Seite 18)

Christian Bird, Nachi Nagappan, Brendan Murphy, Harald Gall, und Premkumar Devanbu. Don't touch my code! examining the effects of ownership on software quality. In *Proceedings of the the eighth joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*. ACM, 2011. (zitiert auf Seite 4)

Daniel Brahneborg und Wasif Afzal. A lightweight architecture analysis of a monolithic messaging gateway. In *2020 IEEE International Conference on Software Architecture companion*, pages 25–32. IEEE, 2020. (zitiert auf Seite 18)

Antonio Bucchiarone, Nicola Dragoni, Schahram Dustdar, Stephan T. Larsen, und Manuel Mazzara. From monolithic to microservices: An experience report from the banking domain. *IEEE Software*, 35(3):50–55, 2018. (zitiert auf Seite 19)

Andrés Carrasco, Brent van Bladel, und Serge Demeyer. Migrating towards microservices: Migration and architecture smells. In *Proceedings of the 2nd International Workshop on Refactoring*, IWoR 2018, pages 1–6. Association for Computing Machinery, 2018. (zitiert auf Seite 7, 18, 23, 24, 42, and 45)

Luiz Carvalho, Alessandro Garcia, Wesley K. G. Assuncao, Rafael de Mello, und Maria Julia de Lima. Analysis of the criteria adopted in industry to extract microservices. In *2019 IEEE/ACM Joint 7th International Workshop on Conducting Empirical Studies in Industry (CESI 2019) and 6th International Workshop on Software Engineering Research and Industrial Practice (SER & IP 2019)*, pages 22–29. IEEE, 2019. (zitiert auf Seite 20 and 22)

Lorenzo de Lauretis. From monolithic architecture to microservices architecture. In *2019 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, pages 93–96. IEEE, 2019. (zitiert auf Seite 18, 22, and 23)

João Paulo Delgado Preti, Adriano Neres Araújo Souza, Evandro César Freiberger, und Tiago de Almeida Lacerda. Monolithic to microservices migration strategy in public safety secretariat of mato grosso. In *2021 International Conference*

on Electrical, Communication, and Computer Engineering (ICECCE), pages 1–5, 2021. (zitiert auf Seite 18 and 22)

Paolo Di Francesco, Patricia Lago, und Ivano Malavolta. Architecting with microservices: A systematic mapping study. *Journal of Systems and Software*, 150:77–97, 2019. (zitiert auf Seite 15)

Nicola Dragoni, Saverio Giallorenzo, Alberto Lluch Lafuente, Manuel Mazzara, Fabrizio Montesi, Ruslan Mustafin, und Larisa Safina. Microservices: Yesterday, today, and tomorrow. In Manuel Mazzara und Bertrand Meyer, editors, *Present and Ulterior Software Engineering*, pages 195–216. Springer International Publishing, 2017. (zitiert auf Seite 4 and 6)

Nicola Dragoni, Ivan Lanese, Stephan Thordal Larsen, Manuel Mazzara, Ruslan Mustafin, und Larisa Safina. Microservices: How to make your application scale. In Alexander K. Petrenko und Andrei Voronkov, editors, *Perspectives of System Informatics*, pages 95–104. Springer International Publishing, 2018. (zitiert auf Seite 6)

Daniel Escobar, Diana Cardenas, Rolando Amarillo, Eddie Castro, Kelly Garces, Carlos Parra, und Rubby Casallas. Towards the understanding and evolution of monolithic applications as microservices. In Claudio Cubillos und Hernán Astudillo, editors, *Proceedings of the 2016 XLII Latin American Computing Conference (CLEI)*, pages 1–11. IEEE, 2016. (zitiert auf Seite 20 and 22)

Sinan Eski und Feza Buzluca. An automatic extraction approach. In Ademar Aguiar, editor, *Proceedings of the 19th International Conference on Agile Software Development Companion*, ACM Other conferences, pages 1–6. ACM, 2018. (zitiert auf Seite 18 and 22)

Eric Evans und Martin Fowler. *Domain-driven design: Tackling complexity in the heart of software*. Addison-Wesley, Upper Saddle River, NJ, 2019. (zitiert auf Seite 4 and 23)

Tomy Firmansyah, Abiyyu Ganeswangga, und Ahmad Nurul Fajar. Transforming the monolith e-procurement application. *International Journal of Innovative Technology and Exploring Engineering*, 9(1):4780–4784, 2019. (zitiert auf Seite 18 and 23)

Jonas Fritzsch, Justus Bogner, Stefan Wagner, und Alfred Zimmermann. Microservices migration in industry: Intentions, strategies, and challenges. In *2019 IEEE International Conference on Software Maintenance and Evolution*, pages 481–490. IEEE, 2019a. (zitiert auf Seite 18, 43, and 44)

Jonas Fritzsch, Justus Bogner, Alfred Zimmermann, und Stefan Wagner. From monolith to microservices: A classification of refactoring approaches. In Jean-Michel Bruel, Manuel Mazzara, und Bertrand Meyer, editors, *Software Engineering Aspects of Continuous Development and New Paradigms of Software Production and Deployment*, volume 11350 of *SpringerLink Bücher*, pages 128–141. Springer International Publishing, 2019b. (zitiert auf Seite 15 and 20)

Nuno Gonçalves, Diogo Faustino, António Rito Silva, und Manuel Portela. Monolith modularization towards microservices: Refactoring and performance trade-offs. In *2021 IEEE 18th International Conference on Software Architecture Companion (ICSA-C)*, pages 1–8, 2021. (zitiert auf Seite 18 and 23)

Jean-Philippe Gouigoux und Dalila 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)*, pages 62–65, 2017. (zitiert auf Seite 20)

Miika Kalske, Niko Mäkitalo, und Tommi Mikkonen. Challenges when moving from monolith to microservice architecture. In Irene Garrigós und Manuel Wimmer, editors, *Current trends in web engineering*, volume 10544 of *Lecture Notes in Computer Science*, pages 32–47. Springer, 2018. (zitiert auf Seite 18, 23, 42, 43, and 44)

Justas Kazanavicius und Dalius Mazeika. Migrating legacy software to microservices architecture. In Dalius Navakauskas, editor, *2019 Open Conference of Electrical, Electronic and Information Sciences (eStream)*, pages 1–5. IEEE, 2019. (zitiert auf Seite 18, 20, and 24)

Holger Knoche und Wilhelm Hasselbring. Drivers and barriers for microservice adoption – a survey among professionals in germany: 1:1–35 pages / enterprise modelling and information systems architectures (emisaj), vol 14 (2019). 2019. (zitiert auf Seite 20 and 43)

Dilshodbek Kuryazov, Dilshod Jabborov, und Bekmurod Khujamuratov. Towards decomposing monolithic applications into microservices. In Abzetdin Adamov, editor, *14th IEEE International Conference Application of Information and Communication Technologies - AICT2020*, pages 1–4. IEEE, 2020. (zitiert auf Seite 18 and 22)

Alan Megargel, Venky Shankararaman, und David K. Walker. Migrating from monoliths to cloud-based microservices: A banking industry example. In Muthu Ramachandran und Zaigham Mahmood, editors, *Software Engineering in the Era of Cloud Computing*, pages 85–108. Springer International Publishing, 2020. (zitiert auf Seite 20)

Mayank Mishra, Shruti Kunde, und Manoj Nambiar. Cracking the monolith. In *ECSA '18: Proceedings of the 12th European Conference on Software Architecture: Companion Proceedings*, pages 1–4. Association for Computing Machinery, 2018. (zitiert auf Seite 19 and 22)

N Bjørndal, A Bucchiarone, M Mazzara, N Dragoni, und Schahram Dustdar. Migration from monolith to microservices : Benchmarking a case study, 2020. (zitiert auf Seite 20)

Sam Newman. *Vom Monolithen zu Microservices: Patterns, um bestehende Systeme Schritt für Schritt umzugestalten*. O'Reilly, Heidelberg, 1. auflage edition, 2021. (zitiert auf Seite 4, 6, 7, and 23)

Samuel Newman. *Building microservices: Designing fine-grained systems*. O'Reilly, Sebastopol, CA, first edition edition, 2015. (zitiert auf Seite 4, 6, 23, and 24)

OASIS SOA-RM Technical Committee. Reference model for service oriented architecture 1.0,committee specification 1. 2006. (zitiert auf Seite 4)

Nuno Santos und Antonio Rito Silva. A complexity metric for microservices architecture migration. In *IEEE 17th International Conference on Software Architecture*, pages 169–178. IEEE, 2020. (zitiert auf Seite 19)

Sindre Grønstøl Haugeland, Phu H Nguyen, Hui Song, und Franck Chauvel. Migrating monoliths to microservices-based customizable multi-tenant cloud-native apps. 2021. (zitiert auf Seite 20)

Jacopo Soldani, Damian Andrew Tamburri, und Willem-Jan van den Heuvel. The pains and gains of microservices: A systematic grey literature review. *Journal of Systems and Software*, 146:215–232, 2018. (zitiert auf Seite 6 and 15)

Davide Taibi, Valentina Lenarduzzi, und Claus Pahl. Processes, motivations, and issues for migrating to microservices architectures: An empirical investigation. *IEEE Cloud Computing*, 4(5):22–32, 2017. (zitiert auf Seite 23 and 42)

Davide Taibi, Valentina Lenarduzzi, und Claus Pahl. Architectural patterns for microservices: A systematic mapping study. In Víctor Méndez Muñoz, Donald Ferguson, Markus Helfert, und Claus Pahl, editors, *CLOSER 2018*, pages 221–232. SCITEPRESS - Science and Technology Publications Lda, 2018. (zitiert auf Seite 19)

Johannes Thones. Microservices. *IEEE Software*, 32(1):116, 2015. (zitiert auf Seite 3)

Victor Velepucha und Pamela Flores. Monoliths to microservices - migration problems and challenges: A sms. In Carlos Iñiguez Jarrín und Gabriela Lorena Suntaxi Oña, editors, *2021 Second International Conference on Information Systems and Software Technologies*, pages 135–142. IEEE, 2021. (zitiert auf Seite 19, 23, 42, and 43)

Claes Wohlin. Guidelines for snowballing in systematic literature studies and a replication in software engineering. In Martin Shepperd, editor, *Proceedings of the 18th International Conference on Evaluation and Assessment in Software Engineering*, ACM Digital Library, pages 1–10. ACM, 2014. (zitiert auf Seite 21)

Pujianto Yugopuspito, Frans Panduwinata, und Sutrisno Sutrisno. Microservices architecture: Case on the migration of reservation-based parking system. In *2017 17th IEEE International Conference on Communication Technology (ICCT 2017)*, pages 1827–1831. IEEE, 2017. (zitiert auf Seite 19)

Jie Zhang, editor. *Addressing global challenges through automation and computing: 2017 23rd International Conference on Automation & Computing : University of Huddersfield, Huddersfield, UK, 7-8 September 2017*, Piscataway, NJ, 2017. IEEE. (zitiert auf Seite 6)

Olaf Zimmermann. Microservices tenets. *Computer Science - Research and Development*, 32(3-4):301–310, 2017. (zitiert auf Seite 4)

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe. Weiterhin wurde die Arbeit bisher weder in einem anderen Prüfungsverfahren vorgelegt noch anderweitig veröffentlicht. Textpassagen, die wörtlich oder dem Sinn nach auf anderen Quellen beruhen, sind als solche kenntlich gemacht.

Magdeburg, 24. Januar 2022