Institut für Softwaretechnologie

Universität Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Masterarbeit

Benutzung von Laufzeitdaten für die Wartbarkeitsanalyse von Service- und Microservice-basierten Systemen

Steffen Matthias Schlinger

Studiengang:	Informatik
Prüfer/in:	Prof. Dr. Stefan Wagner
Betreuer/in:	Justus Bogner
Beginn am:	12. Dezember 2018
Beendet am:	12. Juni 2019

Abstract

Mit der Entwicklung von immer mehr neuen IT-Produkten entsteht gleichzeitig eine immer größer werdende Menge an Quellcode. Da während der Entwicklung selten der Fokus auf eine gute Wartbarkeit gelegt wird, sondern auf andere Qualitätsattribute und die schnelle Fertigstellung des Projekts, wird ein Großteil der Entwicklungskosten für die Wartung benötigt [37, 40, 41]. Selbst wenn die Wartbarkeit eine Anforderung während der Entwicklung darstellt, ist sie schwer umzusetzen und noch schwerer zu messen.

Traditionelle Ansätze basieren auf statischen oder dynamischen Analysen, beispielsweise Quellcode-Analysen, um Erkenntnisse über die Wartbarkeit zu gewinnen. Mit dem Aufkommen neuer Architekturen, wie Service- und Microservice-basierten Systemen, lässt sich die Wartbarkeit noch schlechter bis gar nicht mehr bestimmen. Diese Systeme verfolgen einen verteilten, heterogenen und teils dezentralen Ansatz, worauf die bisherigen Methoden nicht anwendbar sind. Um die Wartbarkeit dieser Systeme zu bestimmen reicht außerdem nicht die Sicht einer einzelnen Komponente des Systems aus, sondern es müssen alle beteiligten Komponenten gleichzeitig betrachtet werden, um die benötigten Informationen zu gewinnen. Da sich statische Analysen durch die starke Verteiltheit und hohe Heterogenität nicht anwenden lassen, bedarf es eines neuen Ansatzes, um die Wartbarkeit zu bestimmen.

Diese Arbeit beschäftigt sich deshalb mit der Nutzung von Laufzeitdaten zur Bestimmung der Wartbarkeit von Service- und Microservice-basierten Systemen. Es wird geprüft, ob sich Laufzeitdaten für diesen Anwendungsbereich eignen und wo die Vor- und Nachteile dieses Ansatzes liegen.

Um die Eignung zu prüfen, wird eine Übersicht über vorhandene Wartbarkeitsmetriken sowie Werkzeuge zum Sammeln von Laufzeitdaten erstellt. Es wird weiterhin ein Framework entworfen und implementiert, welches mithilfe der gefundenen Metriken und Werkzeuge Erkenntnisse über die Wartbarkeit eines Systems berechnen kann. Letztendlich wird das Framework anhand eines Beispielsystems demonstriert und die gewonnenen Erkenntnisse werden ausgewertet und diskutiert.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Danksagung

Ich möchte an dieser Stelle Justus Bogner und Prof. Dr. Stefan Wagner vom Institut für Softwaretechnologie an der Universität Stuttgart für die Betreuung und Prüfung dieser Arbeit danken.

1 Einleitung

Diese Arbeit beschäftigt sich mit der Nutzung von Laufzeitdaten zur Bestimmung der Wartbarkeit von Service- und Microservice-basierten Systemen.

1.1 Motivation

Um Aussagen über die Qualität eines Softwareprodukts treffen zu können werden Kriterien herangezogen, welche eine Einordnung der Software vornehmen. In ISO 25010 [36] werden Qualitätskriterien für Softwaresysteme beschrieben, welche festlegen, wie gut sich Merkmale eines Systems eignen, festgelegte Kriterien zu erfüllen. Diese Kriterien umfassen die Bereiche Kompatibilität, Übertragbarkeit, Wartbarkeit, Performanz, Funktionalität, Zuverlässigkeit, Benutzbarkeit und Sicherheit.

Einige dieser Kriterien sind bzw. werden bereits ausführlich untersucht, während andere bisher weniger Beachtung fanden, wie beispielsweise die Wartbarkeit, obwohl die Wartung einen großen Teil des Entwicklungsbudgets einnimmt [37, 40, 41].

Traditionell wird Software als Monolith entwickelt und betrieben, wobei die Software für all ihre Funktionalitäten selbst zuständig ist. Ein Nachteil dieses Ansatzes ist, dass selbst bei einer nur kleinen Änderung der Software das komplette Programm neu erstellt und in Betrieb genommen werden muss. Auch beim Thema Skalierbarkeit gibt es Nachteile: Wenn die Performance der Software nicht mehr ausreicht, müssen mehrere Instanzen des gesamten Monolithen betrieben werden (falls die Software dafür überhaupt geeignet ist), statt nur die überlasteten Komponenten zu replizieren.

Um die Wartbarkeit einer Software zu verbessern wurde deshalb in letzter Zeit verstärkt auf ein neues Pattern gesetzt: Microservices. Microservices beschreiben den Ansatz, eine Software als eine Gruppe von Diensten zu entwickeln und zu betreiben, statt als einen einzelnen Monolithen. Dieser Ansatz eröffnet neue Möglichkeiten bei der Wartung: Neue Funktionalitäten können beispielsweise mithilfe neuer Dienste umgesetzt werden, ohne dass bereits eingesetzte Dienste beeinflusst werden. Auch können einzelne Dienste skaliert werden, um auf die aktuelle Auslastung reagieren zu können.

Durch die in Microservice-Systemen verbreitete Heterogenität und die verteilte und dezentralisierte Architektur ist die Bestimmung der Wartbarkeit mithilfe traditioneller Werkzeuge schwierig bzw. unmöglich, sodass ein neuer Ansatz benötigt wird. Eine Möglichkeit hierfür ist das Sammeln von Daten während der Laufzeit, um aus den gewonnenen Datensätzen Metriken zur Bestimmung der Wartbarkeit des Systems zu berechnen. Laufzeitdaten werden auch heute schon bei der Bestimmung anderer Qualitätsmerkmale von Software genutzt, beispielsweise um Aussagen über die Performanz oder die Verfügbarkeit von Systemen zu treffen [46].

1.2 Zielsetzung

Das Ziel dieser Arbeit ist es zu untersuchen, ob sich Laufzeitdaten dazu eignen, die Wartbarkeit von Microservice-basierten Systemen zu bestimmen. Um dieses Ziel zu erreichen wird ein Framework erstellt, welches Laufzeitdaten aufzeichnen und daraus Schlüsse auf die Wartbarkeit ziehen kann. Dieses Ziel wird auf folgende Unteraufgaben aufgeteilt:

Auflistung und Analyse Service-orientierter Wartbarkeitsmetriken Für die Berechnung der Wartbarkeit müssen Metriken definiert werden, welche Aussagen über dieses Attribut treffen können. Hierfür werden Service-orientierte Wartbarkeitsmetriken gesucht und aufgelistet, wobei als Basis die Arbeit von Bogner et al. [26] dient. Diese enthält bereits eine ausführliche Auflistung, welche aktualisiert und erweitert werden soll. Weiterhin wird die Eignung der Metriken für Microservice-Systeme untersucht, da hier besondere Voraussetzungen gelten. Außerdem wird untersucht, ob sich die Metriken aus Laufzeitdaten berechnen lassen, da sich diese Arbeit auf jene Teilmenge beschränkt.

Erstellung einer Übersicht über vorhandene Werkzeuge zum Sammeln von Laufzeitdaten Hierbei wird besonderer Fokus auf die Bestimmung der Vor- und Nachteile eines jeden Werkzeugs gelegt, um optimale Laufzeitdaten für die Metrikberechnung zu erhalten, da die Qualität der Aussagen über die Wartbarkeit voraussichtlich stark von der Qualität der Eingangsdaten abhängt. Weiterhin wird untersucht, wie invasiv die gefundenen Werkzeuge sind, d.h. wie groß der Integrationsaufwand ist, beispielsweise ob und wie viel Quellcode-Anpassung notwendig ist. Außerdem soll bestimmt werden, ob die Werkzeuge für Service-orientierte Systeme geeignet sind.

Entwurf und Implementierung eines Frameworks zur Berechnung von Metriken aus Laufzeitdaten Es wird ein Framework zur Berechnung der Metriken spezifiziert und implementiert. Hierbei wird besonderer Wert auf die einfache Erweiterbarkeit und Integrationsmöglichkeit weiterer Datenquellen gelegt, um das Framework an die vielfältigen Microservice-Systeme anpassen zu können. Für die Wartbarkeitsbestimmung wird eine Teilmenge der gefundenen, Microservice-geeigneten Metriken, welche sich anhand von Laufzeitdaten berechnen lassen, ausgewählt. Zum Sammeln der Laufzeitdaten werden geeignete Werkzeuge aus der erstellten Übersicht ausgewählt, welche die für die Metriken erforderlichen Daten bereitstellen können.

Demonstration des Frameworks anhand eines Beispiel-Microservice-Systems Es wird ein bestehendes Microservice-System ausgewählt, welches auf seine Wartbarkeit untersucht werden soll. Zur Erzeugung von Laufzeitdaten wird das System von mehreren Testgruppen anhand gegebener Aufgaben genutzt. Das Framework wird daraufhin auf die erzeugten Laufzeitdaten angewendet und die Ergebnisse werden auf ihre Vollständigkeit und ihre Genauigkeit analysiert.

1.3 Aufbau der Arbeit

Nach einer Einführung zu den Grundlagen der Bereiche Wartbarkeit, Service-orientierte Architektur und Microservices werden verwandte Arbeiten vorgestellt.

Daraufhin wird der untersuchte Ansatz, die Wartbarkeit von Microservice-Systemen anhand von Laufzeitdaten zu ermitteln, beschrieben. Es werden das Architekturmodell und das Datenmodell eingeführt, die Suche und Auswahl der Metriken und Werkzeuge beschrieben und die Implementierung ausgeführt.

Anschließend wird das Framework anhand eines Beispiel-Microservice-Systems demonstriert und die Ergebnisse analysiert. Es wird über die Vor- und Nachteile sowie diverse Eigenschaften dieses Ansatzes diskutiert und es werden Verbesserungsmöglichkeiten aufgezeigt.

Zum Schluss werden die Erkenntnisse zusammengefasst und es wird ein Ausblick auf künftige Möglichkeiten gegeben.

2 Grundlagen

Im Folgenden werden Grundlagen vorgestellt, welche dem Verständnis dieser Arbeit dienen.

2.1 Wartbarkeit

In ISO 25010 (siehe Abbildung 2.1) werden Qualitätskriterien für Softwaresysteme beschrieben, welche festlegen, wie gut sich Merkmale eines Systems eignen, festgelegte Kriterien zu erfüllen [36]. Diese Kriterien umfassen neben der Wartbarkeit außerdem die Bereiche Kompatibilität, Übertragbarkeit, Performanz, Funktionalität, Zuverlässigkeit, Benutzbarkeit und Sicherheit. Nach ISO 25010 beschreibt die Wartbarkeit „den Grad an Effektivität und Effizienz, mit welchem ein Produkt oder System durch das vorgesehene Wartungspersonal modifiziert werden kann“ [36].

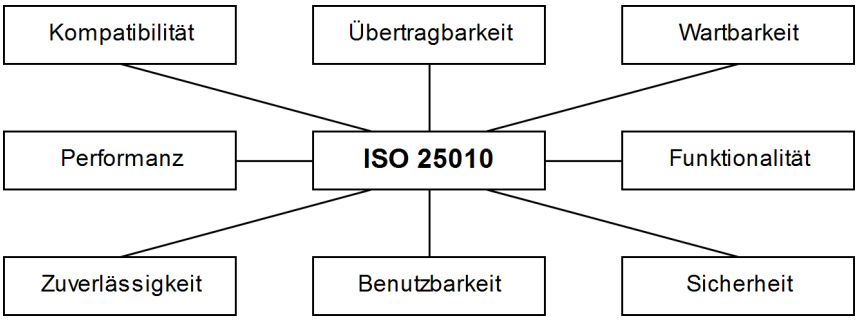

Abbildung 2.1: Übersicht ISO 25010.

Softwaresysteme sind oft über einen langen Zeitraum in Betrieb und müssen währenddessen fortlaufend gewartet werden. Bennett et al. unterteilen die Wartung in vier Unterarten: Die korrektive Wartung, welche die Suche und die Behebung gefundener Fehler beschreibt, die adaptive Wartung zur Anpassung einer Software an geänderte Umgebungsbedingungen, die vollendende Wartung bei neuen Anforderungen an die Software und die präventive Wartung, um zukünftige Fehler zu vermeiden [37]. Die adaptive und die vollendende Wartung werden auch mit dem Begriff Erweiterbarkeit beschrieben. Die Wartbarkeit wird umso wichtiger, umso länger ein Softwaresystem in Betrieb sein wird und umso geringer die Expertise in diesem Sachgebiet ist.

Nach Bennett et al. [37] ist die Erweiterbarkeit für 75% des Wartungsaufwands verantwortlich. Wenn die künftigen geänderten Umgebungsbedingungen und Anforderungen bereits vorhergesehen werden können, kann bereits bei der Entwicklung darauf eingegangen werden und beispielsweise durch Konfigurationsmöglichkeiten auf deren Unterstützung hingearbeitet werden, sodass sich der spätere Wartungsaufwand reduziert.

Die gute Wartbarkeit einer Software kann sich auch positiv auf andere Qualitätsmerkmale einer Software auswirken: Wenn sich eine Software leicht warten lässt, können beispielsweise sicherheitskritische Fehler schneller behoben werden [22]. Auch treten durch einen übersichtlichen und gut strukturierten Quellcode weniger ungewollte Nebeneffekte bei Änderungen an der Software auf, sodass weniger neue Fehler entstehen [22].

2.2 Wartbarkeitsmetriken

Die Wartbarkeit ist ein System-internes Attribut, welches nicht direkt gemessen werden kann [32]. Es gibt keine Einzelmetrik, welche in verschiedenen Systemen erhoben und verglichen werden könnte. Um sie trotzdem beschreiben zu können werden andere Attribute des Systems betrachtet, um daraus Schlüsse über die Wartbarkeit zu ziehen. Diese Schlüsse entstehen durch Metriken, welche einzelne oder mehrere Attribute eines Systems erheben bzw. messen und daraus Erkenntnisse über die Wartbarkeit ziehen.

Die ersten Metriken waren auf einen prozeduralen Programmierstil beschränkt. Beispiele hierfür sind McCabe's Cyclomatic Complexity (CC) [47], welche die Anzahl an unabhängigen Kontrollflüssen durch ein Programm ermittelt, oder die Halstead-Metriken [48], die den Programmumfang und die Programmkomplexität nutzen. Für den Objekt-orientierten Programmierstil wurden diese Metriken überarbeitet. Es entstanden beispielsweise die Metriken von Li und Henry (LH metrics) [49] sowie die Metriken von Chidamber und Kemerer (CK metrics) [50]. Durch immer umfangreichere Systeme wurden diese schließlich in Module unterteilt, wofür neue Metriken, beispielsweise zum Bestimmen des Zusammenhalts auf Modul-Ebene, benötigt wurden. Eine Übersicht über solche Metriken liefert die Arbeit von Sarkar et al. [51]. Als ein Beispiel hierfür ist die Metrik Coupling Between Modules (CBM) von Lindvall et al. zu nennen [52].

Bei der Erhebung der Metriken spielen mehrere Aspekte eine Rolle [30, 31]:

Wichtigkeit Da Metriken nie die vollständige Wartbarkeit eines Systems angeben können, müssen Metriken nach ihrer Wichtigkeit zur Bestimmung der Wartbarkeit eingeordnet werden. Metriken, welche hierzu mehr beitragen sollten höher gewichtet werden als Metriken, welche weniger beitragen.

Vollständigkeit Ist die Metrik vollständig, d.h. fehlen keine Erkenntnisse? Muss die Vollständigkeit manuell überprüft werden oder ist sie immer gegeben?

Genauigkeit Wie exakt ist die erhobene Metrik? Ist eine automatische/manuelle Korrektur nötig?

Interpretation Die berechnete Metrik muss interpretiert werden: In welcher Einheit liegen die Ergebnisse vor? Sind kleinere oder größere Werte besser? Was ist die Bedeutung der Ergebnisse und worüber gibt sie Aufschluss?

Automatisierung Kann die Metrik automatisiert erhoben werden oder sind manuelle Schritte nötig? Gibt es vorhandene Technologien, welche genutzt werden können oder werden Neuentwicklungen benötigt?

Invasivität Wie hoch ist der Aufwand zur Erhebung der benötigten Daten? Muss das zu messende System angepasst werden, und wenn ja wie? Ist eine Quellcode-Anpassung nötig oder reicht eine Anpassung der Konfiguration?

Metriken lassen sich in statische und dynamische Metriken unterteilen [29]. Statische Metriken messen statische Eigenschaften von Systemen, beispielsweise die Anzahl an Diensten oder die festen Abhängigkeiten. Sie sind allerdings ungeeignet, um das dynamische Verhalten eines Systems zu analysieren [28]. Hierfür werden dynamische Metriken eingesetzt: Diese können dynamische Eigenschaften eines Systems erfassen, wie beispielsweise die Anzahl an Aufrufen zwischen zwei Diensten. Sie werden meist durch Ausführungsverfolgungen oder Ausführungsmodelle erhoben [28].

Chhabra et al. haben diese beiden Metriken-Typen verglichen [28]. Sie kommen zu dem Schluss, dass sich statische Metriken leichter sammeln lassen und im Entwicklungsprozess früher erhebbar sind, allerdings auch weniger bzw. ungenauere Erkenntnisse liefern. Weiterhin können dynamische Metriken umfangreichere Daten verwenden, was weitergehende Analysen wie Verhaltensanalysen ermöglicht, wodurch sie Erkenntnisse über mehr Attribute des Systems liefern können als statische Metriken.

Bogner et al. haben eingeordnet, wie sich Attribute eines Systems auf dessen Wartbarkeit auswirken [26]. Die Größe („Size“) eines Microservice-Systems wirkt sich demnach negativ auf die Wartbarkeit aus, da es hierdurch schwieriger zu überblicken ist und Nebenwirkungen von Änderungen nicht so einfach zu erkennen sind. Die Komplexität („Complexity“) wirkt sich ebenfalls negativ auf die Wartbarkeit aus. Sie beschreibt den Aufwand an Interaktionen zwischen einer Anzahl an Elementen, um ein bestimmtes Ziel zu erreichen. Umso komplexer ein System ist, umso schwerer ist es, die Zusammenhänge zu verstehen und damit wird es umso schwerer, das System zu warten. Auch das dritte Attribut, die Kopplung („Coupling“), hat eine negative Auswirkung auf die Wartbarkeit. Es beschreibt die Stärke der Bindung mit anderen Diensten. Umso geringer diese Bindung, umso weniger Wechselwirkungen müssen beachtet werden und umso einfacher wird die Wartung. Der Zusammenhalt („Cohesion“) ist das einzige Attribut, welches eine positive Auswirkung auf die Wartbarkeit hat. Es beschreibt den Grad, zu dem ein Dienst für genau eine Funktionalität zuständig ist. Umso stärker einzelne Funktionalitäten zentralisiert vorliegen, umso einfach können sie geändert werden. Die Zentralisierung („Centralization“) beschreibt, wie stark ein System von zentralen Komponenten abhängt. In Microservice-basierten Systemen kann sie nicht gemessen werden, da sie auf dem Grundsatz der Dezentralisierung basieren, wodurch dieses Attribut hier nicht bestimmt werden kann.

Mithilfe dieser Attribute können Wartbarkeitsmetriken in Kategorien eingeordnet werden, wodurch eine erste Orientierung entsteht, wie Metriken mit Hinblick auf die Wartbarkeit zu interpretieren sind.

2.3 Serviceorientierte Architektur (SOA)

Heutige Unternehmen haben meist eine große Anzahl an verschiedenen und unterschiedlich alten IT-Systemen und Programmen. Es wurde viel Aufwand investiert, diese Systeme zu integrieren, da eine einzelne Software nur in Ausnahmefällen die gesamte benötigte Funktionalität für das Unternehmen abdeckt. Hierdurch entstand eine immer komplexere und unübersichtlichere IT-Landschaft, die sich auch weiterhin fortlaufend weiterentwickelt. Im Zuge der Digitalisierung unterliegen diese IT-Systeme einem ständigen Wandel. Die Veröffentlichungszyklen von Fehlerbehebungen und neuen Funktionalitäten wird zunehmend kürzer, während die Architektur der Systeme immer komplexer wird. Gleichzeitig müssen die Unternehmen in kürzester Zeit auf veränderte Kundenwünsche reagieren, um wettbewerbsfähig zu bleiben [38].

Als Resultat entwickelt sich die Architektur der IT-Systeme eines Unternehmens hin zu einem Ecosystem-basierten Ansatz [38]. Die Funktionalitäten werden in einzelne Module aufgeteilt und verteilt, damit auch Geschäftspartner und Kunden Zugang zu den Unternehmensfunktionalitäten haben. Hierbei kommt es durch die bereits angesprochene vielfältige IT-Landschaft zu Problemen in den Bereich Interoperabilität und Heterogenität [38]. Um diese Probleme zu lösen wurde nach einer neuen Architektur gesucht, welche lose gekoppelt („loosely coupled“), ortstransparent („location transparent“) und Protokoll-unabhängig („protocol independent“) sein sollte [38]. Diese Architektur hat den Vorteil, dass die nutzende Software sich wenig bis keine Gedanken über den expliziten Dienst und seine technischen Details machen muss, sondern die Auswahl eines geeigneten Dienstes der zugrundeliegenden Infrastruktur überlassen kann.

Hieraus entstand die Service-orientierte Architektur oder kurz SOA. Sie beschreibt ein Architekturmuster, welches durch die Trennung von Beschreibung, Implementierung und Bindung ein hohes Level an Flexibilität bei der Weiterentwicklung der IT-Landschaft bietet [45]. Eine Service-orientierte Architektur ist eine leistungsfähige Umgebung auf Unternehmensniveau, welche aus einer Reihe an IT-Diensten besteht, welche die Aufgaben und Prozesse des Unternehmens erfüllen. Diese Dienste lassen sich zu neuen Diensten zusammenstellen, um höhere Aufgaben zu erfüllen.

Ein Dienst in einer SOA kann eigenständig genutzt werden und stellt eine abgeschlossene Funktionalität dar. Er wird in einem Verzeichnis registriert und kann dort über wohldefinierte Schnittstellen gefunden werden. Gebunden wird er dynamisch zur Laufzeit und ist unabhängig von der zugrundeliegenden Plattform, wodurch Interoperabilität sichergestellt wird. Die einzelnen Dienste können bei einer SOA in einer beliebigen Anzahl an IT-Systemen verwendet werden, was sich positiv auf die Wiederverwendbarkeit auswirkt. Als Folge der losen Kopplung können weitere Dienste eingeführt werden, welche neue Funktionen bieten oder vorhandene Funktionen verbessern, ohne bestehende Dienste zu beeinflussen. Durch die dynamische Bindung an Dienste können bestehende Programme die verbesserten Dienste nutzen, ohne dass eine Anpassung der Programme nötig wäre. Weiterhin wird der Dienst durch die Bereitstellung weiterer Instanzen skalierbar, wodurch Probleme mit der Performanz gelöst werden können [45].

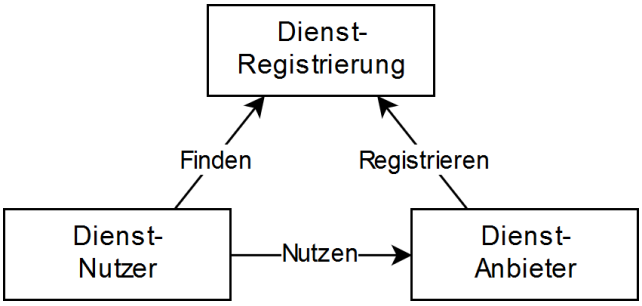

Abbildung 2.2: Übersicht Service-orientierte Architektur, basierend auf [38].

Abbildung 2.2 zeigt den Aufbau einer Service-orientierten Architektur. Die einzelnen Dienste registrieren sich an einer zentralen Stelle, welche hierdurch einen Überblick über die vorhandenen Dienste erhält. Ein anderer Dienst kann nun die Registrierung anhand von Merkmalen nach einem passenden Dienst suchen lassen und den gefundenen Dienst daraufhin direkt nutzen.

2.4 Microservices

Die genaue Definition von Microservices ist umstritten und wird fortlaufend kontrovers diskutiert [33]. Manche Diskussionen setzen Microservices und SOA gleich, andere sehen (verschieden viele) Unterschiede. Bogner et al. haben hierzu eine Arbeit veröffentlicht, welche beide Konzepte Pattern-basiert vergleicht [33]. Sie kamen zu der Erkenntnis, dass 74% der 118 untersuchten SOA-Patterns vollständig mit den Microservice-Charakteristiken vereinbar waren, sodass nach aktuellem Stand tendenziell nicht von einer Gleichsetzung der beiden Konzepte auszugehen ist.

Durch Kombination der Definitionen von S. Newman [34] und J. Thones [35] entsteht folgende Definition: Microservices sind kleine, eigenständige und zusammenarbeitende Dienste, welche für eine bestimmte Funktionalität zuständig sind und unabhängig in Betrieb genommen, skaliert und betrieben werden können. Dabei folgen sie einer Reihe von Charakteristiken, durch welche sie sich von regulären Diensten abheben [34]:

Beschränkter Zusammenhang Jeder Microservice fokussiert sich auf genau eine bestimmte Funktionalität, wodurch der Zusammenhalt innerhalb eines Microservices sehr hoch ist und die Abhängigkeiten nach außen sehr gering sind.

Dezentralisierung Jeder Microservice ist autonom. Die System-Architektur folgt dem Prinzip der Choreographie statt der Orchestrierung, daher gibt es keine zentrale Vermittlungsstelle. Die Auswahl der konkret zu nutzenden Microservice-Instanzen erfolgt dynamisch.

Leichtgewichtige Kommunikation Die Kommunikation zwischen den Microservices erfolgt mithilfe von leichtgewichtigen Netzwerkaufrufen, wodurch eine lose Kopplung gegeben ist. Es wird weitestgehend auf eine Standardisierung der Kommunikation verzichtet.

Technische Heterogenität Jeder Microservice nutzt die Technologie oder Plattform, welche für die Funktionalität am besten geeignet ist. Um die Kompatibilität zwischen den Microservices zu gewährleisten sollten die APIs der Microservices Plattform-unabhängig sein.

Durch diese Charakteristiken entstehen bei der Nutzung von Microservices eine Reihe von Vorteilen [34]:

Ausfallsicherheit Bei Ausfall eines Microservices, welcher nicht für andere Microservices essenziell ist, kann das restliche System ohne die Funktionalität dieses Microservices weiterlaufen. Hierdurch entsteht eine erhöhte Ausfallsicherheit („Resilience“). Sobald der Dienst repariert wurde, kann er wieder in den laufenden Betrieb eingebunden werden. Ein monolithisches System würde beim Ausfall einer Komponente meist komplett ausfallen.

Skalierbarkeit Durch die Eigenständigkeit und die dynamische Bindung an einzelne Microservice-Instanzen können die Microservices skaliert werden. Wenn die Performanz nicht mehr ausreicht, können (bei entsprechender Eignung) weitere Instanzen desselben Microservices gestartet werden, um die Last abzufangen.

Technologie-Unabhängigkeit Da die Microservices lose gekoppelt sind und Plattform-unabhängig miteinander kommunizieren, können die einzelnen Dienste mithilfe der Technologien umgesetzt werden, die am besten für die zu erfüllende Aufgabe geeignet sind. Hierdurch entsteht eine Umgebung mit einer Vielzahl verschiedener Technologien, welche jederzeit austauschbar sind, wenn sich die Anforderungen an den Dienst ändern oder eine Technologie veraltet ist.

Übersichtlichkeit Durch ihre relativ kleine Größe bleiben die einzelnen Microservices übersichtlich und lassen sich so leicht verändern. Da hierdurch Nebenwirkungen der Änderungen leichter zu entdecken sind, werden ungewollte Fehler verringert. Da die Abhängigkeiten zwischen den Microservices nur auf den APIs basieren ist die Architektur sehr stabil und dies hat weiterhin den Vorteil, dass keine versteckten Abhängigkeiten entstehen können.

Teamunterstützung Microservices bringen ebenfalls Vorteile bei der Entwicklung innerhalb von Teams: Durch die Verteilung der Funktionalitäten auf einzelne Dienste können sich die Teams auf verschiedene Dienste aufteilen und so Konflikte bei Änderungen vermeiden.

Einfache Integration Auch die Inbetriebnahme wird erleichtert: Sobald ein neuer Microservice für die Inbetriebnahme bereit ist, kann er dem laufenden Gesamtsystem hinzugefügt werden, ohne dass das System angehalten werden muss. Durch die dynamische Auswahl von Microservice-Instanzen können andere Dienste die veränderten Microservices ohne eigene Anpassung nutzen. Systeme zur fortlaufenden Integration („Continuous Delivery“) werden auch unterstützt.

Wiederverwendbarkeit Einzelne Dienste können iterativ zu höheren Diensten zusammengestellt werden, um komplexere Aufgaben zu erfüllen. Hierdurch steigt die Wiederverwendbarkeit, da bestehende Microservices in neuen Projekten weiterverwendet werden können, ohne dass dieselbe Funktionalität erneut implementiert werden muss. Wenn Fehler in einem Microservice entdeckt werden, müssen diese lediglich in einem Dienst behoben werden, statt dass alle Systeme angepasst werden müssten, welche diesen Dienst nutzen. Hierbei ist allerdings zu beachten, dass durch die mehrfache Verwendung eines Microservices die Abhängigkeit von diesem Dienst steigt, sodass je nach Anwendungsbereich eine Duplizierung sinnvoll sein kann.

Dem gegenüber steht eine Reihe von Nachteilen bei der Nutzung von Microservices [34]:

Monitoring Die verteilte, dezentrale Architektur von Microservice-Systemen erschwert die Überwachung und die Suche nach Fehlerquellen, da mehrere Dienste gleichzeitig betrachtet werden müssen.

Ausfallsicherheit Durch die Microservice-Architektur sinkt zwar die Wahrscheinlichkeit für einen Komplettausfall des Systems, allerdings steigt durch die Vielzahl an einzelnen Diensten die Wahrscheinlichkeit für einen Teilausfall.

Zusätzliche Last Es entsteht zusätzlicher Netzwerkverkehr und zusätzliche Rechenlast, da viele Dienste miteinander kommunizieren müssen.

Komplexität Wie in allen verteilten Systemen muss aufgrund des CAP-Theorems [27] auf eines der Eigenschaften Konsistenz, Verfügbarkeit und Ausfallsicherheit verzichtet werden.

Umfangreiche Technologie-Landschaft Durch die Vielzahl an eingesetzten Technologien müssen die Entwicklungswerkzeuge umfangreich sein und diese Technologien unterstützen. Außerdem müssen Entwickler verfügbar sein, welche sich mit den eingesetzten Technologien auskennen.

In Bezug auf die Wartbarkeit unterscheiden sich Service- und Microservice-basierte Systeme an einigen Stellen [26]:

Oft sind Attribute des Systems verschieden zu interpretieren. Microservice-Systeme basieren zum Beispiel normalerweise auf einer größeren Anzahl an Diensten als reguläre Service-basierte Systeme. Da Microservice-Systeme eine stärkere Modularisierung befürworten, ist hier eine größere Anzahl an Diensten angemessen als bei Service-basierten Systemen. Ein Problem bei der Bestimmung der Wartbarkeit ist die Festlegung eines akzeptablen Bereichs für die Attribute des Systems. Um dieses Problem zu umgehen wird oft auf Durchschnittswerte zurückgegriffen und es werden dynamische statt statische Attribute betrachtet.

Durch die noch höhere Anzahl verschiedener Technologien in Microservice-Systemen im Vergleich zu Service-basierten Systemen wird die Bestimmung der Wartbarkeit komplizierter. Es müssen mehr Technologien adaptiert werden, um an die benötigten Daten zu kommen. Beispielsweise muss für statische Quellcodeanalysen für jede Sprache ein eigener Adapter entwickelt werden, welcher die Wartbarkeit des Quellcodes bestimmen kann.

Weitere Schwierigkeiten kommen durch die stärkere Verteilung des Gesamtsystems hinzu. Microservice-Systeme sind normalerweise vollständig dezentralisiert, wodurch keine zentrale Stelle zur Bestimmung der Wartbarkeit vorhanden ist, wie es oft in Service-basierten Systemen möglich ist. Die benötigten Daten müssen deshalb an verschiedenen Stellen gesammelt und miteinander verknüpft werden. Ein Problem hierbei ist, dass sich die Architektur von Microservice-Systemen laufend ändert, während Service-basierte Systeme meistens eine relativ statische Architektur haben. Die Architektur des Microservice-Systems muss daher laufend bestimmt werden, damit die Daten zur Bestimmung der Wartbarkeit an den richtigen Stellen erhoben werden und verknüpft werden können.

3 Verwandte Arbeiten

Zur Bestimmung der Wartbarkeit müssen geeignete Metriken definiert werden. Mit Aufkommen von verteilten und Service-basierten Systemen mussten die vorhandenen Metriken neu validiert werden. Um Erkenntnisse über Teile des Systems zu erlangen, konnten vorhandene Metriken teilweise weiter genutzt werden, für die Auswertung des Gesamtsystems hingegen benötigte man neue Metriken. Eine Übersicht über Wartbarkeitsmetriken für Service-basierte Systeme haben Bogner et al. erstellt [26]. Diese Übersicht wird als Basis genutzt, um in dieser Arbeit eine Übersicht von Microservice-geeigneten Metriken zu erstellen.

Für die Analyse der Wartbarkeit von Microservice-Systemen muss häufig die Architektur bekannt sein. Diese kann entweder vorgegeben oder extrahiert werden. Eine Übersicht über nicht auf Microservices ausgelegte Ansätze geben L. O'Brien et al. [53]. Diese Übersicht zeigt verschiedene Herangehensweisen, Techniken und Werkzeuge auf. Architecture Reconstruction and MINing (ARMIN) beispielsweise nutzt den Quellcode, um die Architektur des Systems zu extrahieren [54]. Dabei werden die einzelnen Elemente des Systems ausgelesen und auf verschiedene Weisen analysiert und dargestellt.

Im Bereich der Extraktion der Architektur von Microservice-Systemen gibt es bisher fünf veröffentlichte Ansätze:

G. Granchelli et al. haben mit MicroART einen Ansatz vorgestellt, welcher die statische Architektur eines Microservice-basierten Systems extrahiert und dabei auf Methoden der modellgetriebenen Softwareentwicklung (MDE) zurückgreift [20]. Der Ansatz ist in zwei Schritte unterteilt: Während der erste Schritt automatisiert ein physisches Architekturmodell erstellt, dient der zweite Schritt der Erstellung eines logischen Architekturmodells. Da im zweiten Schritt manuelle Schritte nötig sind, lässt sich der Ansatz nicht vollständig automatisieren. Weiterhin muss der Quellcode des Systems als GitHub-Repository vorliegen und das System selbst als Docker-Container laufen.

Der zweite Ansatz, MiSAR, von N. Alshuqayran et al., präsentiert auf Grundlage einer empirischen Studie ein Metamodell für die Architektur von Microservice-basierten Systemen und Zuweisungsregeln zwischen dem Microservice-System und dem Metamodell, um durch MDE-Methoden die Architektur zu extrahieren [55]. Die Erstellung dieser Artefakte erfolgt dabei vollständig manuell.

Der dritte Ansatz, das Microservice Architecture Analysis Tool (MAAT), stammt von Engel et al. und stellt ein vollständiges Framework dar, mit welchem eine Übersicht über ein Microservice-System erstellt werden kann [17]. Es enthält ein eigenes Werkzeug zum Sammeln von Laufzeitdaten mithilfe der OpenTracing API, weiterhin ein Werkzeug zur automatisierten Evaluation der Architektur sowie deren grafischen Aufarbeitung. Bei der Evaluation wird allerdings nur die statische Architektur erfasst und keine dynamischen Aspekte.

M. Cardarelli et al. haben einen Ansatz namens MicroQuality entwickelt, um die Qualität von Microservice-Architekturen zu evaluieren [56]. Dieser Ansatz kann die Architektur wahlweise automatisiert oder manuell extrahieren und in ein Technologie-unabhängiges Format umwandeln, welches auf Qualitätsmerkmale untersucht werden kann. Bei der automatischen Extraktion wird auf vorhandene Werkzeuge, wie beispielsweise MicroART, zurückgegriffen.

Der Ansatz von Mayer et al. kann vollautomatisch und fortlaufend die Architektur extrahieren, indem er statische sowie dynamische Daten sammelt und diese zusammenfügt [19]. Hierbei ist er allerdings auf REST-Interfaces limitiert, was den Anwendungsbereich einschränkt. Die gesammelten Daten werden via REST-APIs zur Verfügung gestellt. Zur Erzeugung der API-Beschreibungen der Dienste wird Swagger [60] genutzt. Die OpenTracing API wird nicht unterstützt und die Nutzung des Werkzeugs ist auf das Spring Framework [42] limitiert.

Die Architektur von Microservices unterliegt einem ständigen Wandel, weshalb diese ständig neu erkannt werden muss. Ahmad et al. haben sich mit der Erkennung von Änderungsmustern in Service-basierten Architekturen beschäftigt [57]. Um dies zu erreichen, werden Architecture Change Logs genutzt, welche sequentielle Änderungen beschreiben. Diese ermöglichen es, Architekturänderungen vorherzusagen und so zur Aktualität des Architekturmodells beizutragen.

Aufbauend auf diesen und weiteren Ansätzen wurden Anwendungsmöglichkeiten gefunden. In der Arbeit von Cito et al. wird das neue Paradigma Feedback-Driven-Development präsentiert [58]. Es beschreibt, wie fortlaufend gewonnene Erkenntnisse über Laufzeitelemente an die Entwickler zurückgegeben werden können, indem die Erkenntnisse direkt mit den entsprechenden Codefragmenten in der Entwicklungsumgebung verbunden werden.

Jin et al. haben einen funktionalitäts-orientierten Ansatz vorgestellt, mit welchem automatisiert Kandidaten zur Umwandlung in Microservices in monolithischen Systemen gefunden werden können [4]. Hierfür werden Ausführungsverfolgungen genutzt, um einen Überblick über zusammengehörige Codefragmente zu bekommen.

Zur grafischen Visualisierung von Microservice-Systemen haben Mayer et al. ein Web-basiertes Dashboard entwickelt, welches zur fortlaufenden Überwachung des Systems sowie zum Management genutzt werden kann [59]. Das Hauptaugenmerk liegt auf der Dokumentation des Systems über einen längeren Zeitraum hinweg.

Keine der vorgestellten Arbeiten setzt gleichzeitig auf eine vollständige Automatisierung, umfangreiche Ergebnisse und eine breite Kompatibilität. Diese Arbeit versucht, all diese Attribute in einem Framework zu vereinen. Um eine vollständige Automatisierung zu erreichen, werden lediglich voll-automatisierbare Werkzeuge zum Sammeln der Laufzeitdaten unterstützt. Durch erweiterbare Metriken können die gewünschten Eigenschaften aus den Laufzeitdaten extrahiert und berechnet werden, wodurch umfangreiche Erkenntnisse gewonnen werden können. Da jede Komponente des Frameworks modular aufgebaut und auf Erweiterbarkeit ausgelegt ist, können durch neue Komponenten, wie neue Integratoren für Datenquellen oder neue Exporteure für die Ausgabe der berechneten Metriken, weitere Technologien unterstützt werden.

Zum Sammeln von Laufzeitdaten wird auf vorhandenen Werkzeugen aufgebaut. Nach bester Kenntnis gibt es noch keine Übersicht über Werkzeuge zum Sammeln von Laufzeitdaten in Microservice-Systemen, sodass für diese Arbeit eine Übersicht erstellt wurde. Einzelne Ansätze, wie die Laufzeitdaten gesammelt werden können, existieren bereits. Ein Konzept ist Dapper von

Google [15]. Es beschreibt eine Tracing-Plattform in verteilten Systemen, welche für einen hohen Datendurchsatz ausgelegt ist und Einblicke in die Ausführungsabläufe der Systeme bietet. Einige Werkzeuge, wie beispielsweise Zipkin, basieren auf diesem Konzept.

4 Ansatz zur Berechnung von Service-basierten Wartbarkeitsmetriken aus Laufzeitdaten

Der vorgestellte Ansatz nutzt Laufzeitdaten, welche Plattform- und Technologie-unabhängig sind sowie die dynamische Architektur von Microservice-Systemen unterstützen, um die Wartbarkeit von Microservice-Systemen zu bestimmen. Bei Beschränkung auf Laufzeitdaten sind die Methoden zur Bestimmung der Wartbarkeit von Microservice-Systemen (exklusive Zentralisierungsmetriken) auch auf Service-basierte Systeme anwendbar [26], daher kann das Framework auch auf Service-basierte Systeme angewendet werden, lediglich die Interpretation der Ergebnisse muss entsprechend angepasst werden.

Der Entwurf und die Implementierung des Frameworks wurden in fünf Teilschritte unterteilt: Es wurden ein Architekturmodell und ein Datenmodell entworfen, welche die Anforderungen an das System in Bezug auf die Erweiterbarkeit erfüllen. Daraufhin wurden Microservice-geeignete Metriken und Werkzeuge gesucht, bewertet und geeignete ausgewählt. Anschließend wurde das gesamte Framework implementiert.

4.1 Architektur

Beim Entwurf der Architektur des Frameworks wurde der Fokus auf eine einfache Erweiterbarkeit gelegt. Das Framework sollte möglichst anpassungsfähig sein, um mit den verschiedenst strukturierten Microservice-Systemen kompatibel zu sein.

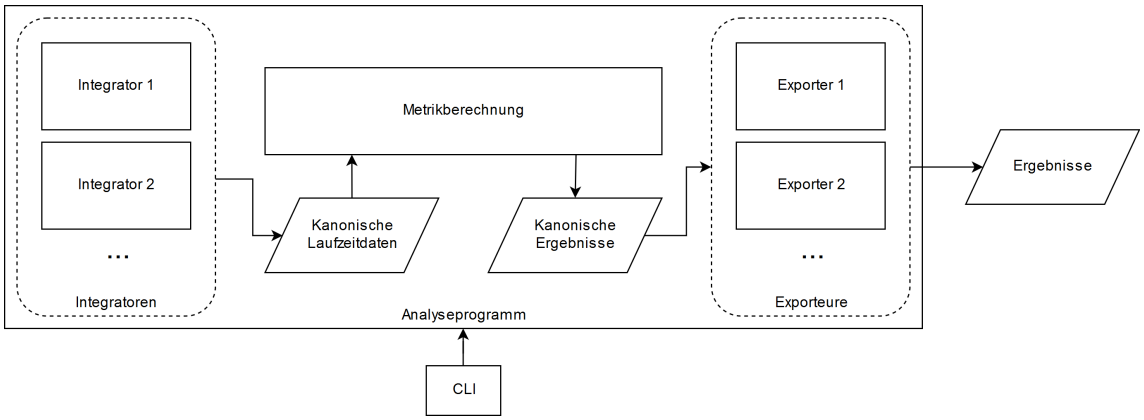

Abbildung 4.1: Architektur Analyseprogramm.

Abbildung 4.1 zeigt den Entwurf des Analyseprogramms. Es besteht aus einer beliebigen Anzahl an Integratoren, welche sequentiell aufgerufen werden, die Laufzeitdaten aus Datenquellen einlesen und in ein kanonisches Format umwandeln. Die Integratoren arbeiten dabei an demselben kanonischen Datenmodell: Dem ersten Integrator wird ein leeres kanonisches Datenmodell zur Verfügung gestellt, welches von den darauffolgenden Integratoren nach Belieben verändert und erweitert werden kann. Hierdurch werden die Laufzeitdaten der verschiedenen Datenquellen zu einem einzelnen Datensatz zusammengesetzt, wodurch sich die Stärken der einzelnen Datenquellen ergänzen und die Schwächen abgemildert werden. Es könnte sich beispielsweise ein Integrator (z.B. eine Netzwerkverkehr-Auswertungskomponente) auf Laufzeitdaten zur Größe des Systems spezialisieren und ein anderer (z.B. eine Tracingkomponente) auf den Zusammenhang zwischen den Diensten. Durch die Kombination der beiden Integratoren würden sich die beiden Spezialisierungen ergänzen.

Es gibt keine Persistierung der Laufzeitdaten innerhalb des Analyseprogramms, diese findet bereits in den Werkzeugen zum Sammeln der Laufzeitdaten statt, sodass das zusätzliche Speichern im Analyseprogramm eine doppelte Speicherung zur Folge hätte. Das Speichern der kanonischen Laufzeitdaten ist ebenfalls nicht sinnvoll, da je nach implementiertem Integrator ausführliche Datentransformationen stattfinden, sodass sich die kanonischen Laufzeitdaten selbst bei kleinen Änderungen in den gesammelten Laufzeitdaten stark verändern können.

Die Laufzeitdaten im kanonischen Format werden an die Metrikberechnung weitergegeben. Diese besteht aus einer Reihe alleinstehender Metrikmodule, welche sich durch weitere Metriken ergänzen lassen. Jedes dieser Module berechnet die Metrik, welche implementiert wurde und erzeugt Ergebnisse in einem kanonischen Format.

Die kanonischen Ergebnisse werden an die Exporteure weitergeleitet, welche die Ergebnisse in das Zielformat umwandeln und ausgeben. Beispiele hierfür wären eine XML-formatierte Ausgabe als Maschinenlesbares Format oder Markdown-formatiert als Menschenlesbares Format. Auch hier ist eine einfache Erweiterbarkeit für neue Dateiformate durch weitere Exporteure gegeben.

4.2 Datenmodell

Es werden zwei kanonische Datenformate benötigt: Da die Laufzeitdaten in verschiedenen Formaten vorliegen, die Metrikberechnung allerdings das Format der Daten kennen muss, ist eine Konvertierung in ein kanonisches Format nötig. Die Ergebnisse der Metrikberechnung müssen ebenfalls in einem kanonischen Format vorliegen, da die Exporteure das Eingabeformat kennen müssen, welches sie in das Zielformat konvertieren sollen.

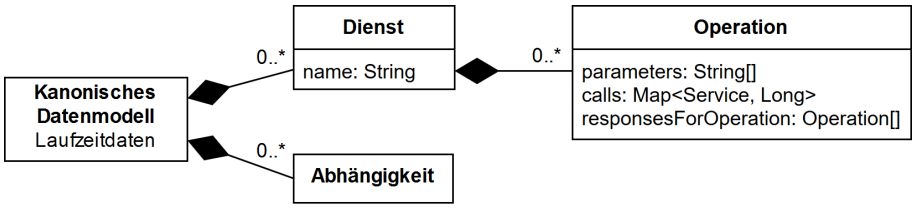

Abbildung 4.2: Datenmodell Laufzeitdaten.

Das kanonische Datenformat für die Laufzeitdaten, zu sehen in Abbildung 4.2, basiert auf einem gerichteten Graphen. Die Knoten repräsentieren die einzelnen Dienste, während die Kanten eine Abhängigkeit eines Dienstes von einem anderen abbilden. Ein Dienst besteht aus einem Namen und einer beliebigen Anzahl an Operationen. Jede Operation besteht aus einer beliebigen Anzahl an Parametern, Aufrufen und möglichen Folgeoperationen. Die Parameter dienen der Identifikation der Operation. Je nach Schnittstellentyp kann ein Parameter verschiedene Formen haben, sodass hier eine Konvertierung in ein einheitliches Format nötig ist. Wenn ein Parameter mehrere Eigenschaften hat, sollten diese in einen einzelnen String gepackt werden, welcher die Operation eindeutig identifiziert. Die Aufrufe repräsentieren, welcher Dienst diese Operation wie oft aufgerufen hat, während die möglichen Folgeoperationen darstellen, welche anderen Operationen desselben oder eines anderen Dienstes als Reaktion auf den Aufruf dieser Operation aufgerufen wurden.

Dieses Datenmodell bietet der Metrikberechnung alle nötigen Daten, um verschiedenste Metriken auf effiziente Weise zu berechnen und weist gleichzeitig eine sehr geringe bis keine Datenredundanz auf.

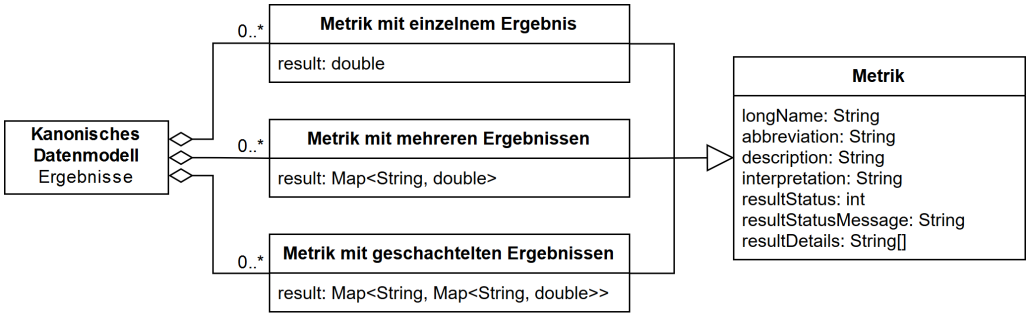

Abbildung 4.3: Datenmodell Ergebnisse der Metrikberechnung.

Das kanonische Datenformat für die Ergebnisse der Metrikberechnung (siehe Abbildung 4.3) basiert auf einer Liste an Metriken. Eine Metrik enthält folgende Elemente:

- Name der Metrik
- Abkürzung des Namens
- Beschreibung der Metrik
- Interpretation des Ergebnisses, welche den möglichen Wertebereich des Ergebnisses aufzeigt sowie Hinweise darauf gibt, wie die Werte zu interpretieren sind
- Ergebnisstatus, welcher angibt, ob ein Fehler bei der Berechnung aufgetreten ist (0) oder ob die Berechnung der Metrik erfolgreich war und ein einzelnes Ergebnis (1), mehrere Ergebnisse (2) oder geschachtelte Ergebnisse (3) erzeugt wurden
- Ergebnisstatusnachricht, welche im Falle eines Fehlers die Fehlernachricht enthält, ansonsten leer ist

- Ergebnis, eines der folgenden Varianten:
  - Einzelner Wert
  - Mehrere Ergebnisse, jedes Ergebnis mit Zuordnung von Bezeichner zu Wert
  - Geschachtelte Ergebnisse, jedes Ergebnis mit Zuordnung von Bezeichner zu einer Liste mit Zuordnung von Bezeichner zu Wert
- Ergebnisdetails, welche erweiterte Informationen zum Ergebnis der Metrik liefern können (beispielsweise die gefundenen Zyklen bei der Metrik Cyclic Service Dependencies (CSD))

Es sind mehrere Arten von Ergebnissen nötig, da manche Metriken ein einzelnes Ergebnis produzieren, andere Ergebnisse pro Dienst und wieder andere pro Dienst zu jedem anderen Dienst. Mit diesen drei Ergebnistypen ist es möglich, alle drei Ergebnisarten darzustellen.

4.3 Metrikauswahl

Um geeignete Metriken für das Framework auswählen zu können muss ein Überblick über die vorhandenen Metriken erstellt werden. Als Basis hierfür dient die Arbeit von Bogner et al., welche eine Übersicht über vorhandene Service-basierte Wartbarkeitsmetriken erstellt haben [26]. Diese Übersicht wurde in einem ersten Schritt auf den aktuellen Stand gebracht. Dabei wurden dieselben Suchkriterien wie bei der ursprünglichen Suche angewendet, wodurch in der Arbeit von Jin et al. [4] fünf weitere Metriken gefunden wurden.

Die vollständige, 58 Metriken umfassende Liste (siehe Tabellen 4.1 und 4.2) wurde daraufhin auf ihre Eignung zum Einsatz in Microservice-Systemen überprüft, wobei sich ergab, dass sich alle Metriken außer den Zentralisierungsmetriken für den Einsatz in Microservice-Systemen eignen, wie auch schon Bogner et al. in ihrer Arbeit festgestellt haben [26].

Da sich diese Arbeit auf die Nutzung von Laufzeitdaten zur Bestimmung der Wartbarkeit beschränkt, wurden alle Metriken auf die Eignung hierfür untersucht und die Ungeeigneten aussortiert. Die verbleibenden Metriken stellen alle geeignete Kandidaten für den Einsatz in diesem Anwendungsbereich dar. In einem letzten Schritt wurden Metriken aussortiert, welche inhaltliche Duplikate einer anderen Metrik darstellen, da hierdurch kein Mehrwert gewonnen würde. Weiterhin wurden die Metriken im Bereich Aggregation aussortiert, da sich diese ohne semantisches Vorwissen nicht zuverlässig automatisiert berechnen lassen.

Die entstandene Metrikliste wurde daraufhin untersucht, ob sie alle Aspekte der Wartbarkeit und Möglichkeiten ihrer Berechnung abdeckt. Dabei wurden mehrere Lücken entdeckt:

- Es gibt noch keine systemweiten Metriken zu den Abhängigkeiten zwischen den Diensten
- Die absolute Anzahl an Aufrufen pro Dienst und pro Operation wird noch nicht mit einbezogen
- Keine Erkennung von Zyklen in den Abhängigkeiten der Dienste, welche mehrere Dienste umfassen

Name	Abkürzung	Bereich	Kategorie	Autoren	Quelle	MS-kompatibel	Laufzeitdaten-geeignet	Ausgewählt
Number of Services Involved in the Compound Service	NSIC	Dienst	Komplexität	Rud et al.	[8]	ja	ja	ja
Services Interdependence in the System	SIY	System	Kopplung	Rud et al.	[8]	ja	ja	ja
Absolute Importance of the Service	AIS	Dienst	Kopplung	Rud et al.	[8]	ja	ja	ja
Absolute Dependence of the Service	ADS	Dienst	Kopplung	Rud et al.	[8]	ja	ja	ja
Absolute Criticality of the Service	ACS	Dienst	Kopplung	Rud et al.	[8]	ja	ja	ja
Weighted Intra-Service Coupling between Elements	WISCE	Element	Kopplung	Perepletchikov et al.	[3]	ja	nein	nein
Weighted Extra-Service Incoming Coupling of an Element	WESICE	Element	Kopplung	Perepletchikov et al.	[3]	ja	nein	nein
Weighted Extra-Service Outgoing Coupling of an Element	WESOCE	Element	Kopplung	Perepletchikov et al.	[3]	ja	nein	nein
Extra-Service Incoming Coupling of Service Interface	ESICSI	Dienst	Kopplung	Perepletchikov et al.	[3]	ja	nein	nein
Element to Extra Service Interface Outgoing Coupling	EESIOC	Element	Kopplung	Perepletchikov et al.	[3]	ja	nein	nein
Service Interface to Intra Element Coupling	SIIEC	Dienst	Kopplung	Perepletchikov et al.	[3]	ja	nein	nein
System Partitioning Factor	SPARF	System	Kopplung	Perepletchikov et al.	[3]	ja	nein	nein
System Purity Factor	SPURF	System	Kopplung	Perepletchikov et al.	[3]	ja	nein	nein
Response for Operation	RFO	Operation	Komplexität	Perepletchikov et al.	[3]	ja	ja	ja
Total Response for Service	TRS	Dienst	Komplexität	Perepletchikov et al.	[3]	ja	ja	ja
Service Interface Data Cohesion	SIDC	Dienst	Zusammenhalt	Perepletchikov et al.	[7]	ja	ja	ja
Service Interface Usage Cohesion	SIUC	Dienst	Zusammenhalt	Perepletchikov et al.	[7]	ja	ja	ja
Service Sequential Usage Cohesion	SSUC	Dienst	Zusammenhalt	Perepletchikov et al.	[7]	ja	nein	nein
Strict Service Implementation Cohesion	SSIC	Dienst	Zusammenhalt	Perepletchikov et al.	[7]	ja	nein	nein
Loose Service Implementation Cohesion	LSIC	Dienst	Zusammenhalt	Perepletchikov et al.	[7]	ja	nein	nein
Total Interface Cohesion of a Service	TICS	Dienst	Zusammenhalt	Perepletchikov et al.	[7]	ja	nein	nein
Service Granularity	SG	Dienst	Komplexität	Qingqing et al.	[1]	ja	ja	nein
Relative Coupling of Service	RCS	Dienst	Kopplung	Qingqing et al.	[1]	ja	ja	ja
Relative Importance of Service	RIS	Dienst	Kopplung	Qingqing et al.	[1]	ja	ja	ja
Service Coupling Factor	SCF	System	Kopplung	Qingqing et al.	[1]	ja	ja	ja
Service Coupling Factor	SCF	System	Kopplung	Hofmeister et al.	[5]	ja	ja	nein
System's Service Coupling	SSC	System	Kopplung	Hofmeister et al.	[5]	ja	ja	nein
Extent of Aggregation	EOA	System	Komplexität	Hofmeister et al.	[5]	ja	ja	nein
System's CentraliZation	SCZ	System	Zentralisierung	Hofmeister et al.	[5]	nein	ja	nein

Tabelle 4.1: Wartbarkeitsmetriken für Service-basierte Systeme, basierend auf [26] (1/2).

Name	Abkürzung	Bereich	Kategorie	Autoren	Quelle	MS-kompatibel	Laufzeitdaten-geeignet	Ausgewählt
Density of Aggregation	DOA	System	Komplexität	Hofmeister et al.	[5]	ja	ja	nein
Aggregator CentraliZation	ACZ	System	Zentralisierung	Hofmeister et al.	[5]	nein	ja	nein
Weighted Service Interface Count	WSIC	Dienst	Größe	Hirzalla et al.	[2]	ja	ja	ja
Stateless Services	SS	System	Komplexität	Hirzalla et al.	[2]	ja	nein	nein
Service Support for Transactions	SST	System	Komplexität	Hirzalla et al.	[2]	ja	nein	nein
Service Realization Pattern	SRP	System	Komplexität	Hirzalla et al.	[2]	ja	nein	nein
Number of Services	NOS	System	Größe	Hirzalla et al.	[2]	ja	ja	nein
Service Composition Pattern	SCP	System	Komplexität	Hirzalla et al.	[2]	ja	ja	ja
Service Access Method	SAM	System	Komplexität	Hirzalla et al.	[2]	ja	nein	nein
Dynamic vs. Static Service Selection	DSSS	System	Komplexität	Hirzalla et al.	[2]	ja	nein	nein
Number of Versions per Service	NOVS	System	Komplexität	Hirzalla et al.	[2]	ja	nein	nein
Average Number of Directly Connected Services	ADCS	System	Kopplung	Shim et al.	[6]	ja	nein	nein
Inverse of Average Number of Used Message	IAUM	System	Zusammenhalt	Shim et al.	[6]	ja	ja	ja
Number of Operations	NO	System	Größe	Shim et al.	[6]	ja	ja	nein
Number of Services	NS	System	Größe	Shim et al.	[6]	ja	ja	ja
AVG # of Operations to AVG # of Messages	AOMR	System	Größe	Shim et al.	[6]	ja	ja	nein
Coarse-Grained Parameter Ratio	CPR	System	Größe	Shim et al.	[6]	ja	nein	nein
CoHesion at Message level	CHM	System	Zusammenhalt	Jin et al.	[4]	ja	nein	nein
CoHesion at Domain level	CHD	System	Zusammenhalt	Jin et al.	[4]	ja	nein	nein
InterFace Number	IFN	System	Größe	Jin et al.	[4]	ja	ja	nein
OPeration Number	OPN	System	Größe	Jin et al.	[4]	ja	ja	nein
InteRaction Number	IRN	Dienst	Größe	Jin et al.	[4]	ja	ja	nein
Mean Absolute Importance/Dependence in the System	MAIDS	System	Kopplung	Schlinger	—	ja	ja	ja
Mean Absolute Coupling in the System	MACS	System	Kopplung	Schlinger	—	ja	ja	ja
Dynamic Relative Dependence of Service	DRDS	Dienst	Kopplung	Schlinger	—	ja	ja	ja
Dynamic Relative Importance of Service	DRIS	Dienst	Kopplung	Schlinger	—	ja	ja	ja
Dynamic Relative Dependence of Service in the System	DRDSS	Dienst	Kopplung	Schlinger	—	ja	ja	ja
Dynamic Relative Importance of Service in the System	DRISS	Dienst	Kopplung	Schlinger	—	ja	ja	ja
Cyclic Service Dependencies	CSD	System	Komplexität	Schlinger	—	ja	ja	ja

Tabelle 4.2: Wartbarkeitsmetriken für Service-basierte Systeme, basierend auf [26] (2/2).

Um diese Lücken zu schließen wurden sieben neue Metriken eingeführt. Im Folgenden bezeichnet S die Menge an Diensten, $s,t \in S$ einzelne Dienste, * einen beliebigen Dienst, D eine Menge von Abhängigkeiten, *IS(s)* ist die Anzahl an Diensten, die von s abhängen, *CS(s)* ist die Anzahl an Diensten, von denen s abhängt, *dependencies(S)* bezeichnet die Menge an Abhängigkeiten der Dienste in S, *cycles(D)* bezeichnet die Menge an Zyklen in D und *C(s, t)* bezeichnet die Anzahl an Aufrufen von s nach t innerhalb des Auswertungszeitraums.

Mean Absolute Importance/Dependence in the System (MAIDS) Diese Metrik ist eine systemweite Variante von AIS/ADS [8]. Sie beschreibt die durchschnittliche Anzahl an Diensten, von denen ein Dienst abhängt bzw. die von einem Dienst abhängen (systemweit sind die beiden Werte gleich). Der Wertebereich des Ergebnisses beträgt [0, INF].

$$MAIDS = \frac{\sum_{s \in S} CS(s)}{|S|} = \frac{\sum_{s \in S} IS(s)}{|S|} \tag{4.1}$$

Mean Absolute Coupling in the System (MACS) Stellt eine systemweite Variante der Metrik ACS [8] dar. Gibt Aufschluss über den Grad der Kopplung systemweit, indem sie die durchschnittliche Anzahl an Abhängigkeitsrelationen zwischen zwei Diensten bestimmt (d.h. wie viele Dienste von einem Dienst abhängig sind bzw. von wie vielen anderen Diensten ein Dienst abhängig ist). Der Wertebereich des Ergebnisses beträgt [0, INF].

$$MACS = \frac{\sum_{s \in S} \Big(CS(s) + IS(s) \Big)}{|S|} = 2 * MAIDS \tag{4.2}$$

Dynamic Relative Dependence of Service (DRDS) Dies ist eine Variante der Metrik RCS [1] und gibt Aufschluss über die dynamische Kopplung zwischen den Diensten. Sie beschreibt anhand der Anzahl an Aufrufen, wie stark ein Dienst s von einem anderen Dienst t prozentual abhängig ist. Der Wertebereich des Ergebnisses beträgt [0, 1].

$$DRDS(s,t) = \frac{C(s,t)}{C(s,*)} \tag{4.3}$$

Dynamic Relative Importance of Service (DRIS) Das Gegenstück zu DRDS und eine Variante der Metrik RIS [1]. Beschreibt anhand der Anzahl an Aufrufen, wie stark ein anderer Dienst t von einem Dienst s prozentual abhängig ist. Der Wertebereich des Ergebnisses beträgt [0, 1].

$$DRIS(s,t) = \frac{C(t,s)}{C(*,s)} \tag{4.4}$$

Dynamic Relative Dependence of Service in the System (DRDSS) Systemweite Variante der Metrik DRDS. Sie beschreibt anhand der Anzahl an Aufrufen, wie stark ein Dienst s von einem anderen Dienst t prozentual im Vergleich zu allen Aufrufen systemweit abhängig ist. Der Wertebereich des Ergebnisses beträgt [0, 1].

$$DRDSS(s,t) = \frac{C(s,t)}{C(*,*)} \tag{4.5}$$

Dynamic Relative Importance of Service in the System (DRISS) Das Gegenstück zu DRDSS und eine systemweite Variante der Metrik DRIS. Beschreibt anhand der Anzahl an Aufrufen, wie stark ein anderer Dienst t von einem Dienst s prozentual im Vergleich zu allen Aufrufen systemweit abhängig ist. Der Wertebereich des Ergebnisses beträgt [0, 1].

$$DRISS(s,t) = \frac{C(t,s)}{C(*,*)} \tag{4.6}$$

Cyclic Service Dependencies (CSD) Systemweite Metrik, welche in den Abhängigkeiten der Dienste die Anzahl an Zyklen bestimmt (Zyklen in den Abhängigkeiten verschlechtern die Wartbarkeit, weshalb sie vermieden werden sollten). Während die Metrik SIY [8] nur Zyklen ohne dazwischen liegende Dienste erkennt, sucht diese Metrik nach Zyklen mit einer beliebigen Anzahl an dazwischenliegenden Diensten. Der Wertebereich des Ergebnisses beträgt [0, INF].

$$CSD = |cycles(dependencies(S))| \tag{4.7}$$

Die daraus entstandene Liste an Metriken (siehe Tabelle 4.3) stellt die zu implementierenden Metriken dar, welche die Wartbarkeit des Zielsystems bestimmen sollen. Sie umfasst 23 Metriken, 13 aus der Kategorie Kopplung, 5 aus der Komplexität, 3 aus dem Zusammenhalt und 2 aus der Größe. 8 sind systemweite Metriken, 14 auf Dienstebene und eine auf Operationsebene.

4.4 Werkzeugauswahl

Das Framework soll möglichst viele Datenquellen unterstützen, weshalb ein modularer Ansatz mithilfe von Werkzeugen und Integratoren gewählt wurde, welche Laufzeitdaten sammeln und in das kanonische Laufzeitdatenformat konvertieren. Für das Sammeln der Laufzeitdaten wurde eine Übersicht an Werkzeugen erstellt (siehe Tabelle 4.4). Hierfür wurden dieselben Suchkriterien wie bei der Suche nach Metriken angewendet, da diese Arbeiten neben den Metriken oftmals Hinweise auf Werkzeuge enthalten, welche geeignete Laufzeitdaten sammeln können.

Durch die Suche wurden 13 Werkzeuge gefunden, welche sich in 6 explizit Microservice-geeignet, 6 implizit Microservice-geeignet und ein nicht Microservice-geeignet einteilen lassen.

Name	Abkürzung	Bereich	Kategorie	Autoren	Quelle
Number of Services Involved in the Compound Service	NSIC	Dienst	Komplexität	Rud et al.	[8]
Services Interdependence in the System	SIY	System	Kopplung	Rud et al.	[8]
Absolute Importance of the Service	AIS	Dienst	Kopplung	Rud et al.	[8]
Absolute Dependence of the Service	ADS	Dienst	Kopplung	Rud et al.	[8]
Absolute Criticality of the Service	ACS	Dienst	Kopplung	Rud et al.	[8]
Response for Operation	RFO	Operation	Komplexität	Perepletchikov et al.	[3]
Total Response for Service	TRS	Dienst	Komplexität	Perepletchikov et al.	[3]
Service Interface Data Cohesion	SIDC	Dienst	Zusammenhalt	Perepletchikov et al.	[7]
Service Interface Usage Cohesion	SIUC	Dienst	Zusammenhalt	Perepletchikov et al.	[7]
Relative Coupling of Service	RCS	Dienst	Kopplung	Qingqing et al.	[1]
Relative Importance of Service	RIS	Dienst	Kopplung	Qingqing et al.	[1]
Service Coupling Factor	SCF	System	Kopplung	Qingqing et al.	[1]
Weighted Service Interface Count	WSIC	Dienst	Größe	Hirzalla et al.	[2]
Service Composition Pattern	SCP	System	Komplexität	Hirzalla et al.	[2]
Inverse of Average Number of Used Message	IAUM	System	Zusammenhalt	Shim et al.	[6]
Number of Services	NS	System	Größe	Shim et al.	[6]
Mean Absolute Importance/Dependence in the System	MAIDS	System	Kopplung	Schlinger	—
Mean Absolute Coupling in the System	MACS	System	Kopplung	Schlinger	—
Dynamic Relative Dependence of Service	DRDS	Dienst	Kopplung	Schlinger	—
Dynamic Relative Importance of Service	DRIS	Dienst	Kopplung	Schlinger	—
Dynamic Relative Dependence of Service in the System	DRDSS	Dienst	Kopplung	Schlinger	—
Dynamic Relative Importance of Service in the System	DRISS	Dienst	Kopplung	Schlinger	—
Cyclic Service Dependencies	CSD	System	Komplexität	Schlinger	—

Tabelle 4.3: Ausgewählte Wartbarkeitsmetriken für das Framework.

Name	Integration	Autoren	Quelle	MS-kompatibel	Ausgewählt
Zipkin	Quellcode Anpassung nötig, Bibliotheken für viele Sprachen verfügbar	Zipkin Team	[11]	ja	ja
Jaeger	Quellcode Anpassung nötig, Bibliotheken für viele Sprachen verfügbar	Jaeger Team	[12]	ja	nein
Google Dapper	Quellcode Anpassung nötig, keine Implementierung verfügbar	Sigelman et al.	[15]	ja	nein
MicroART	Keine Quellcode-Anpassung, allerdings manuelle Schritte nötig	Granchelli et al.	[20]	ja	nein
Microservice Architecture Analysis Tool (MAAT)	Quellcode-Anpassung nötig, Bibliotheken für viele Sprachen verfügbar	Engel et al.	[17]	ja	nein
Data Collection Library	Quellcode-Anpassung nötig, Bibliothek verfügbar	Mayer et al.	[19]	ja	nein
Dynatrace	Client-Programm, keine Quellcode-Anpassung nötig	Dynatrace LLC.	[9]	ja	nein
New Relic	Client-Programm, keine Quellcode-Anpassung nötig	New Relic, Inc.	[10]	ja	nein
Pivot Tracing	HDFS mit Baggage, keine Quellcode-Anpassung nötig	Mace et al.	[13]	ja	nein
Prometheus	Quellcode Anpassung nötig, Bibliotheken für viele Sprachen verfügbar	Prometheus Team	[16]	ja	nein
Apache HTrace	Quellcode Anpassung nötig, einzelne Bibliotheken verfügbar	HTrace Team	[14]	ja	nein
Kieker	Mehrere Möglichkeiten (Quellcode Anpassung, Middleware, ...)	Bielefeld et al.	[18]	ja	nein
Tcpdump	Passive Netzwerküberwachung, keine Quellcode Anpassung nötig	Tcpdump Team	[21]	nein	nein

Tabelle 4.4: Übersicht Werkzeuge zum Sammeln von Laufzeitdaten.

Explizit für Microservices entworfen wurden die beiden nicht-kommerziellen Werkzeuge Zipkin [11] und Jaeger [12]. Sie sind Implementierungen der OpenTracing API, welche Hersteller-unabhängige APIs zur Ansteuerung von verteilten Tracingsystemen bereitstellt. Das Sammeln sowie die Auswertung von Laufzeitdaten werden unterstützt. Sie benötigen eine zentrale Serverkomponente inklusive Datenspeicher und unterstützen Erweiterungen, wie beispielsweise externe Datenspeicher.

Die Grundlage für Zipkin bietet Google Dapper [15]. Es ist eine Konzeptarbeit, welche ein System zum Tracing in verteilten Systemen beschreibt. Besonderer Wert wurde dabei auf die Skalierbarkeit in großen Systemen gelegt.

Granchelli et al. haben ein semi-automatisches Werkzeug zur Extraktion der Architektur von Microservice-Systemen namens MicroART entworfen [20]. Es stellt die statische Architektur eines Systems dar und ist nur auf Systeme anwendbar, welche in Docker laufen. Weiterhin muss der Quellcode in einem GitHub-Repository liegen, um statische Informationen extrahieren zu können. Es ist nicht-kommerziell und Open Source.

Das nicht-kommerzielle Microservice Architecture Analysis Tool (MAAT) ist ein vollständiges Framework, welches eine Übersicht über ein Microservice-System erstellen kann [17]. Es kann allerdings auch nur zum Sammeln von Laufzeitdaten genutzt werden. Zum Sammeln der Laufzeitdaten wird auch hier die OpenTracing API genutzt. Aus den gesammelten Daten wird die Architektur extrahiert und weitere Informationen werden grafisch aufbereitet dargestellt.

Auch die Data Collection Library von Mayer et al. gehört zu einem größeren Framework, kann aber auch eigenständig genutzt werden [19]. Sie sammelt statische und dynamische Daten und stellt diese über REST-APIs (Representational State Transfer) bereit. Es wird Swagger [60] zur Erzeugung der API-Beschreibungen der Dienste genutzt. Die OpenTracing API wird nicht unterstützt. Außerdem ist die Nutzung des Werkzeugs auf das Spring Framework [42] limitiert.

Zu den implizit Microservice-unterstützenden Werkzeugen zählen Dynatrace [9] und New Relic [10]. Die beiden sind kommerzielle Werkzeuge, welche als Software as a Service (SaaS) bereitgestellt werden. Sie stellen Laufzeit- und Infrastrukturdaten über ein System bereit und enthalten außerdem Tracingsysteme zur Verfolgung von Aufrufen über mehrere Dienste hinweg.

Das Werkzeug Pivot Tracing von Mace et al. ist ein nicht-kommerzielles Werkzeug zum Verfolgen von Aufrufen in verteilten Systemen [13]. Es ermöglicht das Sammeln von Laufzeitdaten, allerdings beschränkt auf Hadoop Distributed File System (HDFS) Umgebungen. Umfangreiche API-Schnittstellen zur Ansteuerung sind vorhanden. Außerdem ermöglicht es während dem Betrieb die laufende Erstellung neuer Auswertungen von einem zentralen Punkt aus. Ein ähnliches Werkzeug ist das Open Source Werkzeug Prometheus, allerdings ohne die Einschränkung auf HDFS Umgebungen [16]. Es ermöglicht weiterhin die Integration von Daten aus Systemen von Drittanbietern und die Visualisierung der gesammelten Daten.

Apache HTrace ist ein nicht-kommerzielles Tracing-Werkzeug mit Fokus auf verteilte Systeme [14]. Es sammelt Laufzeitdaten mithilfe einer zentralen Serverkomponente und eines Datenspeichers. Der Funktionsumfang ist allerdings geringer als bei anderen Werkzeugen.

Auch Kieker ist ein nicht-kommerzielles Werkzeug zum Tracing in verteilten Systemen [18]. Zum Sammeln von Laufzeitdaten bietet es eine Vielzahl an Integrationsmöglichkeiten mit verschiedener Invasivität. Es enthält vordefinierte Auswertungsmöglichkeiten sowie die Möglichkeit zur Erweiterung um eigene Auswertungen.

Das einzige nicht Microservice-geeignete Werkzeug ist Tcpdump [21]. Es schneidet den Netzwerkverkehr mit und bietet Analysemöglichkeiten für die gesammelten Laufzeitdaten. Eine native Unterstützung für Microservices ist nicht vorhanden, durch eine ausführliche Analyse der gesammelten Daten kann aber zumindest eine Grundmenge an Informationen bereitgestellt werden.

Klassische Auswertungsmöglichkeiten wie Logdateien oder Datenbanken sind für Microservice-Systeme nicht geeignet, da sie Aufrufe nur aus der Sicht eines einzelnen Dienstes darstellen und die verschiedenen Sichten der einzelnen Dienste nicht trivial miteinander verknüpft werden können. Des Weiteren wären massive Quellcode-Anpassungen nötig, um zumindest ausreichende Informationen zu speichern, welche nötig sind, um die Metriken berechnen zu können.

Die Wahl des Werkzeugs hat eine erhebliche Auswirkung auf die Qualität der Laufzeitdaten und damit auf die Genauigkeit des Endergebnisses. Für das Framework wurde zur Demonstration ein Werkzeug ausgewählt, welches alle für die ausgewählten Metriken benötigten Daten bereitstellen kann. Bei der Auswahl wurde darauf geachtet, dass das Werkzeug das Tracing von Aufrufen über mehrere Dienste hinweg erlaubt sowie die OpenTracing API unterstützt, damit das verwendete Tracingsystem problemlos ausgetauscht werden kann. Das Werkzeug sollte skalierbar sein, um auch umfangreiche Systeme, welche im Microservice-Bereich oftmals üblich sind, unterstützen zu können. Außerdem sollte das Werkzeug frei verfügbar sein, um Lizenzprobleme zu vermeiden. Da nur wenige Analysefunktionen innerhalb des Werkzeugs benötigt werden, wurde auf einem geringen Overhead geachtet, um die Last durch das Werkzeug gering zu halten.

Nach Auswertung anhand dieser Kriterien wurde Zipkin als geeignetster Kandidat ausgewählt, da es alle gestellten Anforderungen erfüllt, leichtgewichtig ist und alle für die Metriken benötigten Daten bereitstellen kann.

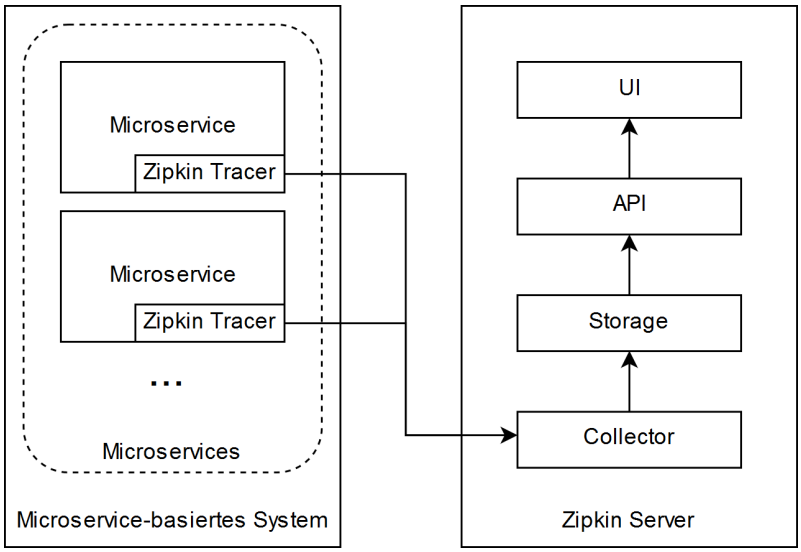

Abbildung 4.4: Architektur Zipkin.

Abbildung 4.4 zeigt den Aufbau von Zipkin. Die Microservices werden um eine Tracingkomponente erweitert, welche Daten über Aufrufe an den zentralen Zipkin-Server schicken. Hierbei wird die OpenTracing API genutzt, es stehen für viele Sprachen fertige Bibliotheken bereit. Ausgehenden Aufrufen werden extra Headerfelder hinzugefügt, um die Zusammengehörigkeit eines Aufrufs

über mehrere Microservices verfolgen zu können. Der Transport zum Zipkin-Server kann über verschiedene Transporte wie HTTP, RabbitMQ [43] oder Apache Kafka [44] erfolgen. Die Zipkin-Integration erzeugt hierbei nur einen geringen Overhead und arbeitet weitestgehend asynchron, um die Laufzeit nicht zu verlängern [11].

Der Zipkin-Server besteht aus einer Collector-Komponente, welche die Daten von den Microservices entgegennimmt, einem Storage mit optionaler externer Datenbank, einer API, um auf die gespeicherten Daten zuzugreifen und einer UI zur grafischen Aufbereitung der Daten.

Zipkin unterscheidet zwischen Spans und Traces. Ein Span umfasst die Daten, die bei einem (Operations-)Aufruf entstehen, wie die ID des Spans und des zugehörigen Traces, Zeitstempel und das Ziel des Aufrufs. Ein Trace hingegen ist eine Menge an zusammengehörigen Spans, welche als Baumstruktur dargestellt werden und den Weg eines Aufrufs durch das System darstellen.

4.5 Implementierung

Das Analyseprogramm wurde vollständig in Java implementiert. Als Abhängigkeitsverwaltung wird Apache Maven [39] eingesetzt.

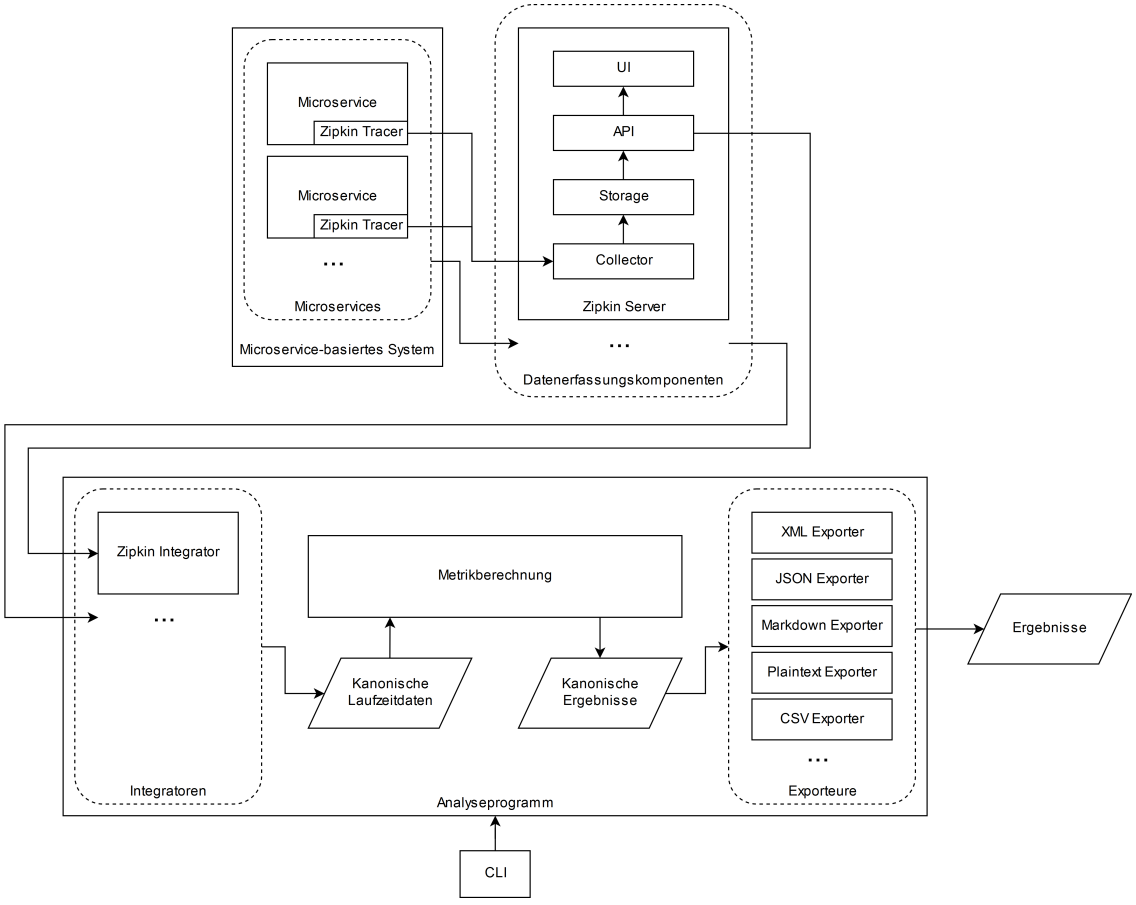

Abbildung 4.5: Vollständige Architektur des Frameworks.

Bei der Umsetzung der Architektur wurde darauf geachtet, die Klassen nahe am entworfenen Design zu halten, um eine möglichst einfache Erweiterbarkeit und Wartbarkeit zu gewährleisten.

Abbildung 4.5 zeigt die vollständige Architektur des Frameworks. Jeder Integrator, jede Metrik und jeder Exporteur wurde als eigene Klasse implementiert, um ein geeignetes Modularisierungslevel zu erreichen.

In dieser ersten Version des Programms wurde Zipkin als Integrator implementiert, 23 Metriken zur Bestimmung der Wartbarkeit und fünf Exporteure: Als maschinenlesbare Formate XML (Extensible Markup Language), JSON (JavaScript Object Notation) und CSV (Comma-Separated Values), als menschenlesbare Formate Markdown und Plaintext. Diese fünf Formate decken alle potenziellen Anwendungsgebiete des Frameworks ab.

Weitere Details zur Implementierung liegen dem Projekt als technische Dokumentation bei.

5 Demonstration des Frameworks anhand eines Beispielsystems

Im Folgenden wird ein Beispielsystem für die Demonstration des Frameworks gesucht und analysiert, der Aufbau der Demonstration beschrieben und die Ergebnisse ausgewertet.

5.1 Beschreibung des Beispielsystems

Zur Demonstration des Frameworks wird es auf gesammelte Laufzeitdaten aus einem Beispielsystem angewendet. Bei der Auswahl des Beispielsystems wurde darauf geachtet, dass das System eine mittlere Größe hat, sodass das Framework komplexe Laufzeitdaten erhält, aber gleichzeitig die Ergebnisse mit vertretbarem Aufwand kontrolliert werden können. Außerdem sollte es eine Oberfläche bereitstellen, über welche mehrere Benutzer gleichzeitig Laufzeitdaten zu jeder Funktionalität des Systems erzeugen können.

Auf Basis dieser Anforderungen wurde das System „ramanujan“ [24] ausgewählt. Es ist ein auf Microservices basierendes Microblogging-System mit Web Frontend, ähnlich wie Twitter. Es nutzt Node.js [25] und ist vollständig in JavaScript geschrieben. Die Architektur basiert auf dem Seneca Framework [23], einem Microservice-Toolkit, welches ein dezentrales Kommunikationssystem bereitstellt, über welches die einzelnen Microservices miteinander kommunizieren können. Die Kommunikation ist Publish-Subscribe-basiert, d.h. die einzelnen Microservices geben an, für welche Art von Nachrichten sie sich interessieren und gesendete Nachrichten werden darauf basierend an die interessierten Empfänger geroutet. Der Vorteil an diesem Ansatz ist, dass keine Suche nach passenden Diensten notwendig ist, wodurch das System sehr anpassungsfähig wird.

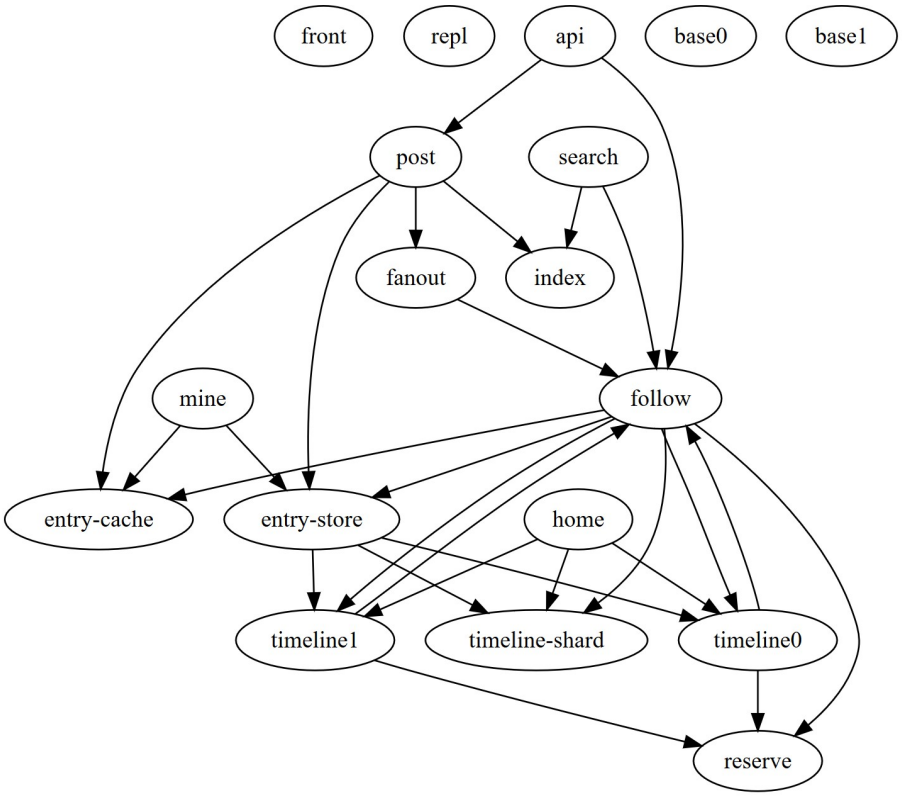

Abbildung 5.1: Architektur des Beispiel-Microservice-Systems ramanujan.

Das System besteht aus 16 einzigartigen und insgesamt 18 Microservices mit vielfältigen Abhängigkeiten, zu sehen in Abbildung 5.1. Die Dienste „base“ und „timeline“ werden jeweils mit zwei Instanzen gestartet, um eine Lastverteilung zu erreichen. Der Timeline-Dienst beispielsweise setzt dies um, indem jede der Instanzen für die Hälfte der möglichen Benutzernamen zuständig ist. Mehrere Instanzen eines Dienstes werden in dieser Arbeit als eigenständige Dienste betrachtet. Der Ausfall einzelner Dienste kann das Gesamtsystem in den meisten Fällen nicht zum Absturz bringen, lediglich einzelne Funktionalitäten fallen aus. Das System verfolgt eine strikte Trennung von Funktionalitäten auf einzelne Microservices, wodurch eine ausführliche Kommunikation zwischen den einzelnen Diensten entsteht.

Das verwendete System besitzt außerdem bereits eine Anbindung an Zipkin, wodurch keine weitere Anpassung des Quellcodes notwendig ist.

5.2 Aufbau der Demonstration

Abbildung 5.2 zeigt den Aufbau der Demonstration. Es wurden zwei Gruppen (Gruppe 1 und Gruppe 2) mit jeweils drei Personen gebildet, welche dieselben zwei Aufgaben erfüllen sollten. Die erste Aufgabe bestand darin, innerhalb von fünf Minuten alle Funktionen des Systems zu finden und zu benutzen. Die zweite Aufgabe war es, über das System fünf Minuten lang mit den anderen

Gruppenmitgliedern ein Gespräch über ein beliebiges Thema zu führen. Zwischen den einzelnen Aufgaben wurde das Microservice-System zurückgesetzt (die Daten im Microservice-System sind nicht relevant, es werden nur die von Zipkin gesammelten Daten zur Auswertung benötigt).

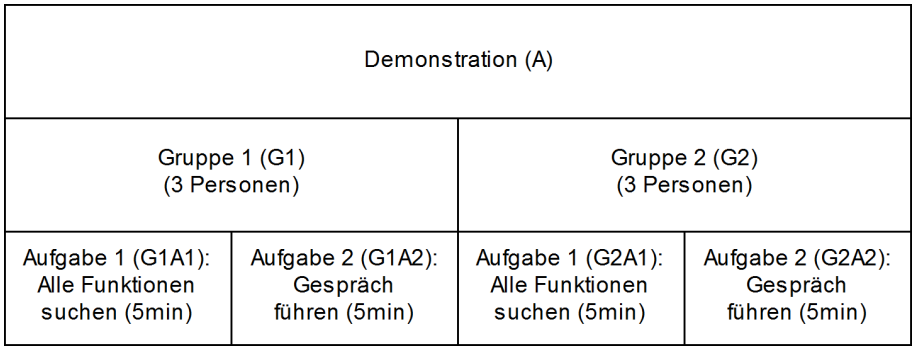

Abbildung 5.2: Aufbau der Demonstration.

Die erste Aufgabe (A1) dient dazu, möglichst alle Abhängigkeiten und Funktionalitäten innerhalb des Systems abzudecken. Die zweite Aufgabe (A2) dient einer realitätsnahen Nutzung des Systems, bei welcher nicht unbedingt immer alle Funktionalitäten des Systems genutzt werden.

5.3 Auswertung der Ergebnisse

Nachdem alle Gruppen ihre Aufgaben abgeschlossen hatten, wurde das Framework auf folgende Teilmengen der gesammelten Laufzeitdaten angewendet:

- Vollständig auf alle vier Aufgaben
- Alle vier Aufgaben einzeln
- Jede Gruppe einzeln
- Jede Aufgabe einzeln

Im Folgenden wird das Ergebnis über alle vier Aufgaben (A) auf seine Vollständigkeit und Korrektheit überprüft, danach werden die Ergebnisse über Teilmengen der gesammelten Laufzeitdaten mit dem Ergebnis über den vollständigen Datensatz verglichen, um Unterschiede bei verschiedenen Nutzungsmustern zu finden.

Bei allen Auswertungen werden die dynamischen Metriken DRDS, DRIS, DRDSS und DRISS nicht beachtet, da sich diese nicht sinnvoll in Bezug auf die Korrektheit und Vollständigkeit auswerten lassen und sie sich je nach Nutzungsmuster unterschieden. Allerdings weisen sie in allen Ergebnisdatensätzen Zahlen aus, welche dem angewendeten Nutzungsmuster innerhalb der Aufgabe entsprechen.

Ergebnisse bei Anwendung auf den vollständigen Laufzeitdatensatz (A)

Tabelle 5.1 zeigt die Ergebnisse bei Anwendung des Frameworks auf den vollständigen Laufzeitdatensatz. Es wurden 14 der 18 Microservices gefunden (NS). Nicht gefunden wurden die beiden Dienste, welche für das Routing der Nachrichten verantwortlich sind („base0“, „base1“), sowie der Dienst, welcher den Webserver implementiert („front“), da diese keine Nachrichten verschicken oder empfangen und deshalb in den gesammelten Laufzeitdaten nicht auftauchen. Ein weiterer nicht gefundener Dienst ist der Dienst, welcher es erlaubt, manuell Nachrichten in das Kommunikationsnetzwerk einzubringen („repl“). Dieser ist über das Webinterface nicht erreichbar und wurde deshalb während der Erfüllung der Aufgaben nicht genutzt.

Alle Metriken zu Abhängigkeiten innerhalb des Systems (NSIC, SIY, AIS, ADS, ACS, RCS, RIS, SCF, SCP, MAIDS, MACS, CSD) weisen auf Basis der gefundenen Dienste korrekte Zahlen aus, ebenfalls die Metriken, welche sich auf Größen des Systems beziehen (WSIC, IAUM, RFO, TRS) sowie die Metriken zum Zusammenhalt der Dienste (SIUC, SIDC). Da in zwei Aufgaben das komplette System genutzt werden sollte waren hier auch korrekte Ergebnisse zu erwarten.

Ergebnisse bei Anwendung auf die Laufzeitdaten der einzelnen Gruppen und Aufgaben (G1A1, G1A2, G2A1, G2A2)

Die Ergebnisse der Aufgaben 1 und 2 der ersten Gruppe (G1A1, G1A2) gleichen den Ergebnissen des vollständigen Datensatzes (A), d.h. alle betrachteten Metriken liefern hier dieselben Ergebnisse (siehe Tabelle 5.1).

Die Ergebnisse der einzelnen Aufgaben der zweiten Gruppe (G2A1, G2A2) lieferten abweichende Zahlen. In den Ergebnissen zu Aufgabe 1 (G2A1) (siehe Tabelle 5.2) wurden zwar wie im vollständigen Datensatz (A) alle 14 Microservices gefunden (NS), allerdings fehlen vier Abhängigkeiten, weshalb einige der darauf aufbauenden Metriken zu Abhängigkeiten (NSIC, AIS, ADS, ACS, RCS, RIS, SCF, MAIDS, MACS) einen geringen Fehler aufweisen und die Metrik CSD keine Zyklen finden kann. SIY und SCP hingegen bestimmten die richtigen Werte. Die Metriken zu Größen des Systems WSIC und IAUM wurden korrekt berechnet, RFO hingegen fehlen wenige Folgeoperationen, wodurch auch TRS geringfügig fehlerhaft ist. Die Zusammenhaltsmetrik SIUC ist ebenfalls geringfügig verfälscht, während SIDC korrekt berechnet wurde.

Die Fehler lassen sich alle auf die fehlenden Abhängigkeiten zurückführen. Die Quelle der fehlenden Abhängigkeiten liegt in der Auswahl der Benutzernamen der Teilnehmer der zweiten Gruppe. Diese Benutzernamen lagen alle auf demselben Timeline-Dienst, weswegen die zweite Instanz des Dienstes kaum angesprochen wurde und die Abhängigkeiten dieser Instanz nicht vollständig erkannt werden konnten.

In den Ergebnissen der zweiten Gruppe zu Aufgabe 2 (G2A2) (siehe Tabelle 5.3) tritt aus demselben Grund dasselbe Problem auf, wenn auch durch eine andere Wahl der Benutzernamen in geringerem Ausmaße. Die Anzahl der gefundenen Microservices (NS) ist auch hier 14, allerdings liegt die Anzahl an nicht gefundenen Abhängigkeiten hier nur bei einer einzigen Abhängigkeitsrelation. Durch den kleinen Fehler werden nur wenige Abhängigkeitsmetriken geringfügig verfälscht (AIS, ADS, ACS, RCS, RIS, MAIDS, MACS). Die restlichen Metriken liefern dieselben Resultate wie die Ergebnisse des vollständigen Datensatzes (A).

Metrik	Ergebnisse
Number of Services Involved in the Compound Service (NSIC)	mine: 8, fanout: 8, entry-cache: 1, index: 1, follow: 7, timeline0: 7, home: 8, timeline1: 7, search: 9, post: 10, reserve: 1, api: 11, timeline-shard: 1, entry-store: 7
Services Interdependence in the System (SIY)	2
Absolute Importance of the Service (AIS)	mine: 0, fanout: 1, entry-cache: 3, index: 2, follow: 5, timeline0: 4, home: 0, timeline1: 4, search: 0, post: 1, reserve: 3, api: 0, timeline-shard: 4, entry-store: 3
Absolute Dependence of the Service (ADS)	mine: 2, fanout: 4, entry-cache: 0, index: 0, follow: 6, timeline0: 2, home: 3, timeline1: 2, search: 2, post: 4, reserve: 0, api: 2, timeline-shard: 0, entry-store: 3
Absolute Criticality of the Service (ACS)	mine: 0, fanout: 4, entry-cache: 0, index: 0, follow: 30, timeline0: 8, home: 0, timeline1: 8, search: 0, post: 4, reserve: 0, api: 0, timeline-shard: 0, entry-store: 9
Number of Services (NS)	14
Relative Coupling of Service (RCS)	mine: 14%, fanout: 28%, entry-cache: 0%, index: 0%, follow: 43%, timeline0: 14%, home: 21%, timeline1: 14%, search: 14%, post: 28%, reserve: 0%, api: 14%, timeline-shard: 0%, entry-store: 21%
Relative Importance of Service (RIS)	mine: 0%, fanout: 7%, entry-cache: 21%, index: 14%, follow: 36%, timeline0: 28%, home: 0%, timeline1: 28%, search: 0%, post: 7%, reserve: 21%, api: 0%, timeline-shard: 28%, entry-store: 21%
Service Coupling Factor (SCF)	0.16
Weighted Service Interface Count (WSIC)	mine: 0, fanout: 2, entry-cache: 3, index: 3, follow: 2, timeline0: 2, home: 0, timeline1: 2, search: 0, post: 1, reserve: 2, api: 0, timeline-shard: 4, entry-store: 2
Service Composition Pattern (SCP)	71%
Inverse of Average Number of Used Message (IAUM)	0.61
Service Interface Usage Cohesion (SIUC)	timeline1: 50%, fanout: 100%, post: 100%, reserve: 100%, entry-cache: 67%, index: 50%, timeline-shard: 75%, follow: 50%, entry-store: 50%, timeline0: 50%
Service Interface Data Cohesion (SIDC)	timeline1: 100%, fanout: 0%, post: 100%, reserve: 100%, entry-cache: 67%, index: 67%, timeline-shard: 100%, follow: 100%, entry-store: 100%, timeline0: 100%
Response for Operation (RFO)	timeline1: [timeline:insert]: 0, [timeline:list]: 1 fanout: [fanout:entry]: 1, [info:entry]: 1 post: [post:entry]: 1 reserve: [reserve:remove]: 0, [reserve:create]: 0 entry-cache: [kind:entry, store:save]: 0, [kind:entry, store:list]: 0, [cache:true, kind:entry, store:*]: 0 index: [info:entry]: 0, [search:insert]: 0, [search:query]: 0 timeline-shard: [timeline:insert]: 0, [shard:0, timeline:*]: 1, [shard:1, timeline:*]: 1, [timeline:list]: 1 follow: [follow:list]: 2, [follow:user]: 2 entry-store: [kind:entry, store:save]: 1, [kind:entry, store:list]: 0 timeline0: [timeline:insert]: 0, [timeline:list]: 1
Total Response for Service (TRS)	13
Mean Absolute Importance/Dependence in the System (MAIDS)	2.14
Mean Absolute Coupling in the System (MACS)	4.29
Cyclic Service Dependencies (CSD)	2

Tabelle 5.1: Ergebnisse über den vollständigen Laufzeitdatensatz (A), gleicht den Ergebnissen der Teilmengen G1, G2, A1, A2, G1A1 und G1A2.

Ergebnisse bei Anwendung auf die Laufzeitdaten der einzelnen Gruppen (G1, G2)

Die Ergebnisse der ersten Gruppe (G1) (siehe Tabelle 5.1), welche einzeln bereits dem Datensatz A gleichen, entsprechen zusammen wiederum dem Datensatz A, d.h. die Ergebnisse der ersten Gruppe für Aufgabe 1 (G1A1), Aufgabe 2 (G1A2) und Aufgabe 1 und 2 kombiniert (G1) sind gleich.

Die Ergebnisse der Aufgaben 1 und 2 der zweiten Gruppe waren wegen unvollständiger Laufzeitdaten einzeln betrachtet (G2A1, G2A2) leicht verfälscht. Die Ergebnisse der zweiten Gruppe zusammen (G2) (siehe Tabelle 5.1) wiederum sind gleich den Ergebnissen des Datensatzes A. Dies bedeutet, dass in beiden Aufgaben verschiedene fehlende Abhängigkeiten zur Verfälschung der Ergebnisse geführt haben und dass die Auswertung über beide Aufgaben die fehlenden Abhängigkeiten ergänzt und so zu korrekten Ergebnissen geführt hat.

Metrik	Ergebnisse
Number of Services Involved in the Compound Service (NSIC)	mine: 8, fanout: 7, entry-cache: 1, index: 1, follow: 6, timeline0: 6, home: 7, timeline1: 6, search: 8, post: 10, reserve: 1, api: 11, timeline-shard: 1, entry-store: 7
Services Interdependence in the System (SIY)	2
Absolute Importance of the Service (AIS)	mine: 0, fanout: 1, entry-cache: 3, index: 2, follow: 5, timeline0: 3, home: 0, timeline1: 3, search: 0, post: 1, reserve: 3, api: 0, timeline-shard: 3, entry-store: 2
Absolute Dependence of the Service (ADS)	mine: 2, fanout: 1, entry-cache: 0, index: 0, follow: 5, timeline0: 2, home: 3, timeline1: 2, search: 2, post: 4, reserve: 0, api: 2, timeline-shard: 0, entry-store: 3
Absolute Criticality of the Service (ACS)	mine: 0, fanout: 1, entry-cache: 0, index: 0, follow: 25, timeline0: 6, home: 0, timeline1: 6, search: 0, post: 4, reserve: 0, api: 0, timeline-shard: 0, entry-store: 6
Number of Services (NS)	14
Relative Coupling of Service (RCS)	mine: 14%, fanout: 7%, entry-cache: 0%, index: 0%, follow: 36%, timeline0: 14%, home: 21%, timeline1: 14%, search: 14%, post: 28%, reserve: 0%, api: 14%, timeline-shard: 0%, entry-store: 21%
Relative Importance of Service (RIS)	mine: 0%, fanout: 7%, entry-cache: 21%, index: 14%, follow: 36%, timeline0: 21%, home: 0%, timeline1: 21%, search: 0%, post: 7%, reserve: 21%, api: 0%, timeline-shard: 21%, entry-store: 14%
Service Coupling Factor (SCF)	0.14
Weighted Service Interface Count (WSIC)	mine: 0, fanout: 2, entry-cache: 3, index: 3, follow: 2, timeline0: 2, home: 0, timeline1: 2, search: 0, post: 1, reserve: 2, api: 0, timeline-shard: 4, entry-store: 2
Service Composition Pattern (SCP)	71%
Inverse of Average Number of Used Message (IAUM)	0.61
Service Interface Usage Cohesion (SIUC)	timeline1: 50%, fanout: 100%, post: 100%, reserve: 100%, entry-cache: 56%, index: 50%, follow: 50%, timeline-shard: 75%, timeline0: 50%, entry-store: 50%
Service Interface Data Cohesion (SIDC)	timeline1: 100%, fanout: 0%, post: 100%, reserve: 100%, entry-cache: 67%, index: 67%, follow: 100%, timeline-shard: 100%, timeline0: 100%, entry-store: 100%
Response for Operation (RFO)	timeline1: [timeline:insert]: 0, [timeline:list]: 1 fanout: [fanout:entry]: 1, [info:entry]: 1 post: [post:entry]: 1 reserve: [reserve:remove]: 0, [reserve:create]: 0 entry-cache: [kind:entry, store:list]: 0, [kind:entry, store:save]: 0, [cache:true, kind:entry, store:*]: 0 index: [info:entry]: 0, [search:insert]: 0, [search:query]: 0 follow: [follow:user]: 2, [follow:list]: 1 timeline-shard: [timeline:insert]: 0, [shard:0, timeline:*]: 0, [shard:1, timeline:*]: 1, [timeline:list]: 1 timeline0: [timeline:insert]: 0, [timeline:list]: 1 entry-store: [kind:entry, store:save]: 1, [kind:entry, store:list]: 0
Total Response for Service (TRS)	11
Mean Absolute Importance/Dependence in the System (MAIDS)	1.86
Mean Absolute Coupling in the System (MACS)	3.71
Cyclic Service Dependencies (CSD)	0

Tabelle 5.2: Ergebnisse über die Laufzeitdaten der ersten Aufgabe der zweiten Gruppe (G2A1).

Ergebnisse bei Anwendung auf die Laufzeitdaten der einzelnen Aufgaben (A1, A2)

Die Ergebnisse der beiden Aufgaben, d.h. beide Aufgaben 1 (A1) und beide Aufgaben 2 (A2) gemeinsam, sind gleich und entsprechen den Ergebnissen des Datensatzes A (siehe Tabelle 5.1).

Da die beiden Laufzeitdatensätze der ersten Gruppe (G1A1, G1A2) vollständig und die der zweiten Gruppe (G2A1, G2A2) geringfügig unvollständig sind, beide Aufgaben einzeln (A1, A2) allerdings dieselben Ergebnisse liefern wie der Datensatz A, müssen die fehlenden Abhängigkeiten durch die jeweilige Aufgabe der ersten Gruppe (G1A1, G1A2) ausgebessert worden sein.

Metrik	Ergebnisse
Number of Services Involved in the Compound Service (NSIC)	mine: 8, fanout: 8, entry-cache: 1, index: 1, follow: 7, timeline0: 7, home: 8, timeline1: 7, search: 9, post: 10, reserve: 1, api: 11, timeline-shard: 1, entry-store: 7
Services Interdependence in the System (SIY)	2
Absolute Importance of the Service (AIS)	mine: 0, fanout: 1, entry-cache: 3, index: 2, follow: 5, timeline0: 4, home: 0, timeline1: 4, search: 0, post: 1, reserve: 2, api: 0, timeline-shard: 4, entry-store: 3
Absolute Dependence of the Service (ADS)	mine: 2, fanout: 4, entry-cache: 0, index: 0, follow: 6, timeline0: 1, home: 3, timeline1: 2, search: 2, post: 4, reserve: 0, api: 2, timeline-shard: 0, entry-store: 3
Absolute Criticality of the Service (ACS)	mine: 0, fanout: 4, entry-cache: 0, index: 0, follow: 30, timeline0: 4, home: 0, timeline1: 8, search: 0, post: 4, reserve: 0, api: 0, timeline-shard: 0, entry-store: 9
Number of Services (NS)	14
Relative Coupling of Service (RCS)	mine: 14%, fanout: 28%, entry-cache: 0%, index: 0%, follow: 43%, timeline0: 7%, home: 21%, timeline1: 14%, search: 14%, post: 28%, reserve: 0%, api: 14%, timeline-shard: 0%, entry-store: 21%
Relative Importance of Service (RIS)	mine: 0%, fanout: 7%, entry-cache: 21%, index: 14%, follow: 36%, timeline0: 28%, home: 0%, timeline1: 28%, search: 0%, post: 7%, reserve: 14%, api: 0%, timeline-shard: 28%, entry-store: 21%
Service Coupling Factor (SCF)	0.16
Weighted Service Interface Count (WSIC)	mine: 0, fanout: 2, entry-cache: 3, index: 3, follow: 2, timeline0: 2, home: 0, timeline1: 2, search: 0, post: 1, reserve: 2, api: 0, timeline-shard: 4, entry-store: 2
Service Composition Pattern (SCP)	71%
Inverse of Average Number of Used Message (IAUM)	0.61
Service Interface Usage Cohesion (SIUC)	timeline1: 50%, fanout: 100%, post: 100%, reserve: 100%, entry-cache: 67%, index: 50%, timeline-shard: 75%, follow: 50%, entry-store: 50%, timeline0: 50%
Service Interface Data Cohesion (SIDC)	timeline1: 100%, fanout: 0%, post: 100%, reserve: 100%, entry-cache: 67%, index: 67%, timeline-shard: 100%, follow: 100%, entry-store: 100%, timeline0: 100%
Response for Operation (RFO)	timeline1: [timeline:insert]: 0, [timeline:list]: 1 fanout: [fanout:entry]: 1, [info:entry]: 1 post: [post:entry]: 1 reserve: [reserve:remove]: 0, [reserve:create]: 0 entry-cache: [kind:entry, store:save]: 0, [kind:entry, store:list]: 0, [cache:true, kind:entry, store:*]: 0 index: [info:entry]: 0, [search:insert]: 0, [search:query]: 0 timeline-shard: [timeline:insert]: 0, [shard:0, timeline:*]: 1, [shard:1, timeline:*]: 1, [timeline:list]: 1 follow: [follow:list]: 2, [follow:user]: 2 entry-store: [kind:entry, store:save]: 1, [kind:entry, store:list]: 0 timeline0: [timeline:insert]: 0, [timeline:list]: 1
Total Response for Service (TRS)	13
Mean Absolute Importance/Dependence in the System (MAIDS)	2.07
Mean Absolute Coupling in the System (MACS)	4.14
Cyclic Service Dependencies (CSD)	2

Tabelle 5.3: Ergebnisse über die Laufzeitdaten der zweiten Aufgabe der zweiten Gruppe (G2A2).

5.4 Fazit der Ergebnisse

Bei der Auswertung über den vollständigen Datensatz (A) wurden alle Dienste und Abhängigkeiten gefunden, welche auch genutzt wurden und dadurch Laufzeitdaten erzeugt haben. Die nicht genutzten Dienste wurden nicht gefunden.

Die Auswertungen der einzelnen Gruppen (G1, G2), einzelnen Aufgaben (A1, A2) und Aufgaben der Gruppe 1 (G1A1, G1A2) lieferten dieselben Ergebnisse wie die Ergebnisse des vollständigen Datensatzes (A), d.h. es wurden alle genutzten Dienste und Abhängigkeiten gefunden. Die Ergebnisse der Aufgaben der Gruppe 2 (G2A1, G2A2) waren wegen nicht gefundener Abhängigkeiten aufgrund einer nicht ausreichend genutzten Funktionalität leicht verfälscht.

Zusammengefasst lässt sich sagen, dass alle genutzten Dienste und Abhängigkeiten bei der Analyse der Laufzeitdaten gefunden werden. Umso umfangreicher ein System genutzt wird, umso genauer sind auch die aus den Laufzeitdaten gewonnenen Ergebnisse. Wenn nicht alle Funktionalitäten genutzt werden, können diese Teile der Architektur nicht erkannt werden.

6 Diskussion

Im Folgenden werden die gesammelten Erkenntnisse über den vorgestellten Ansatz beschrieben und analysiert.

6.1 Performanz und Skalierbarkeit des Frameworks

Das Framework ist für eine gute Skalierbarkeit entwickelt worden. Prinzipiell könnte es bei diesem Ansatz an vier Stellen zu Performance-Problemen kommen:

- Innerhalb der Werkzeuge zum Sammeln der Laufzeitdaten
- In den Integratoren
- Bei der Metrikberechnung
- In den Exporteuren

Die Werkzeuge zum Sammeln der Laufzeitdaten müssen für die anfallende Datenmenge geeignet sein, d.h. sie dürfen nicht mit der anfallenden Datenmenge überlastet sein, um Informationsverlust und Ausfällen vorzubeugen. Gerade in großen und viel genutzten Microservice-Systemen fallen viele Laufzeitdaten an, welche gespeichert werden sollten. Die genaue anfallende Datenmenge hängt vom verwendeten Werkzeug ab und kann daher stark variieren. Eine Möglichkeit zur Reduzierung der Datenlast ist die Beschränkung auf eine Teilmenge der Laufzeitdaten. Da oftmals nur ein Teil des Gesamtsystems betrachtet werden soll reicht es, nur die Laufzeitdaten, welche diesen Bereich betreffen, zu sammeln. Eine weitere Möglichkeit ist das Sammeln nur jedes x-ten Laufzeitdatensatzes, um die anfallende Datenmenge zu reduzieren. Hierbei können allerdings leicht Informationen verloren gehen, da unabhängig vom Informationsgehalt aussortiert wird. Auch ist es möglich, Laufzeitdaten nur über einen bestimmten Zeitraum zu sammeln oder gesammelte Laufzeitdaten automatisiert nach einer bestimmten Zeit archivieren oder löschen zu lassen, um den Datenbestand des Werkzeugs zu verkleinern.

Weiterhin darf durch das verwendete Werkzeug keine allzu große zusätzliche Latenz eingeführt werden, um die Antwortzeiten nicht zu sehr zu erhöhen. Eine Möglichkeit dies zu vermeiden ist es, das Sammeln und Speichern der Laufzeitdaten soweit wie möglich asynchron zu gestalten. Außerdem sollten nur an den Stellen Daten gesammelt werden, an denen es auch unbedingt notwendig ist. Auf die doppelte Sammlung derselben Daten sollte verzichtet werden.

Die Integratoren müssen die gesammelten Laufzeitdaten in das kanonische Laufzeitdatenformat umwandeln. Dafür ist es in den meisten Fällen nötig, den gesamten Datenbestand für den auszuwertenden Zeitraum zu betrachten, wodurch die Laufzeit der Integratoren abhängig von der Implementierung meistens linear ansteigen dürfte. Zur Beschleunigung dieses Prozesses wäre

es beispielsweise möglich, den Datenbestand vorauszusortieren, um noch vor weitergehenden Analyseschritten die Laufzeitdatenmenge zu verkleinern. Die Verarbeitung der Laufzeitdaten lässt sich oftmals linear beschleunigen, indem Multithreading-Techniken eingesetzt werden. Verschiedene Teilmengen der Laufzeitdaten könnten so von verschiedenen Threads gleichzeitig verarbeitet werden, wodurch bei einer geschickten Implementierung eine lineare Verringerung der Laufzeit erreicht werden könnte. Die Laufzeitdaten sind normalerweise gut für eine parallele Verarbeitung geeignet, da die Schreibzugriffe auf gemeinsame Datenmodelle weit geringer als die Lesezugriffe sind und so nur wenig gegenseitiger Ausschluss nötig ist.

Die Metrikberechnung ist fast vollständig unabhängig von der Größe der kanonischen Laufzeitdaten. Die Laufzeitdaten liegen der Metrikberechnung als stark zusammengefasster Graph vor, welcher nur noch die nötigsten Informationen ohne Datenredundanz enthält. Jeder Microservice ist auf einen Knoten mit einigen Attributen und jede Abhängigkeit auf eine Kante im Graphen reduziert. Durch dieses Datenformat können viele der Metriken bereits mit einfachen Graphoperationen bestimmt werden. Auch komplexere Metriken lassen sich durch Möglichkeiten der Graphentheorie und simplen Arithmetik-Operationen effizient berechnen. Hierdurch wird innerhalb der Metrikberechnung eine nahezu konstante Laufzeit erreicht, bei sehr ungünstigen Laufzeitdaten im schlechtesten Fall eine lineare Laufzeit.

Die Exporteure erhalten eine Liste an Metriken und deren Attribute, inklusive der berechneten Ergebnisse. Die Laufzeit der Exporteure ist linear abhängig von der Anzahl an Metriken, die Größe der verwendeten Laufzeitdaten spielt hier (fast) gar keine Rolle mehr. Auf die Gesamtlaufzeit des Frameworks haben die Exporteure keinen nennenswerten Einfluss, solange keine zeitaufwendigen Exportmethoden (z.B. Speicherung auf Archivbändern) implementiert wurden.

Die Nutzung mehrere Instanzen des Frameworks macht nur bei verschiedenen Eingabeparametern Sinn, eine Beschleunigung der Berechnungen ist hierdurch nicht möglich. Die einzelnen Verarbeitungsschritte erzwingen eine sequentielle Ausführung, sodass nur geringe Möglichkeiten für die Parallelisierung vorhanden sind.

Alles in allem spielen die Wahl und die Konfiguration der Werkzeuge zum Sammeln der Laufzeitdaten sowie die Implementierung der Integratoren eine große Rolle bei der Performanz und Skalierbarkeit des Frameworks, die Metrikberechnung sowie die Exporteure eine geringe bis keine Rolle.

6.2 Erweiterbarkeit des Ansatzes

Das Framework verfolgt einen modularen Ansatz mit Fokus auf einfache Erweiterbarkeit, wodurch sich alle drei Hauptkomponenten (Integratoren, Metriken und Exporteure) leicht erweitern lassen.

Die Erstellung eines weiteren Integrators erfordert die Erstellung einer neuen Klasse, welche das Integratoren-Interface implementiert. Dem Integrator werden die globalen und eigenen Parameter sowie das bisherige kanonische Datenmodell, welches erweitert werden soll, übergeben. Als Rückgabe wird lediglich das erweiterte kanonische Datenmodell erwartet. Auf welche (externen) Daten zur Erweiterung des Modells zurückgegriffen wird ist nicht beschränkt, daher kann jede beliebige Datenquelle genutzt werden. Auch der Umfang und die Art der Berechnungen unterliegen keiner Beschränkung, sodass auch komplexe Transformationen von Daten möglich sind. Da das

bisherige kanonische Datenmodell mit an die Integratoren übergeben wird können auch Integratoren realisiert werden, welche Fehler bzw. Ungenauigkeiten der anderen Integratoren abmildern oder beheben.

Die Metriken verfolgen einen ähnlichen Ansatz: Zur Einführung einer neuen Metrik wird auch hier eine neue Klasse erstellt, welche das Metrik-Interface implementiert, als Eingabe das kanonische Laufzeitdatenmodell erhält und als Ausgabe eine Instanz eines Metrik-Objekts zurückgibt, welches Attribute wie den Namen der Metrik sowie die berechneten Ergebnisse enthält. Wie die Ergebnisse berechnet werden ist nicht limitiert, selbst die Laufzeitdaten müssen nicht zwingend verwendet werden. Dies eröffnet die Möglichkeit, jede denkbare Form von Metrik zu berechnen.

Auch zur Erstellung eines neuen Exporteurs wird eine neue Klasse erstellt, welche das Exporteur-Interface implementiert. Als Eingabe erhält der Exporteur die globalen und eigenen Parameter sowie die erstellte Metrikliste samt Attributen und Ergebnissen. Er transformiert die Ergebnisse in das Zielformat und sendet bzw. schreibt sie in das Ziel. Auch hier sind keine Beschränkungen bei der Transformation der Daten sowie bei der Wahl des Ziels gegeben, sodass das Framework im Einzelbetrieb, in Programm-Pipelines sowie in Continuous Delivery (CD) / Integration (CI) Umgebungen eingesetzt werden kann.

Durch diesen modularen Ansatz kann das Framework an fast jede Umgebung und jeden Anwendungsbereich angepasst werden. Es können Integratoren je nach vorhandenen und benötigten Laufzeitdaten erstellt werden, es können genau die Metriken berechnet werden, welche von Interesse sind, und die Ergebnisse können direkt in das benötigte Format konvertiert und weitergegeben werden. Hierdurch eröffnet sich ein breites, potenzielles Anwendungsspektrum.

6.3 Gefährdung der Validität

Die Qualität der Ergebnisse des Frameworks hängt maßgeblich von der Qualität der Eingangsdaten (= Laufzeitdaten) ab. Bei perfekten Eingangsdaten liefern die Metrikberechnung und die Exporteure perfekte Ergebnisse, daher sind möglichst optimale Eingangsdaten anzustreben.

Die Qualität der Eingangsdaten hängt allerdings vom Umfang und von der Genauigkeit der gesammelten Daten des genutzten Werkzeugs sowie von der Qualität des zugehörigen Integrators ab. Wenn die gesammelten Laufzeitdaten nicht vollständig oder ungenau sind, kann dies vom Integrator meistens nur zu einem begrenzten Umfang kompensiert werden, wodurch die Eingangsdaten eine geringere Qualität haben.

Durch eine fortgeschrittene Implementierung der Integratoren kann meistens die Qualität der gesammelten Laufzeitdaten erhöht werden, indem die Daten ausführlich analysiert werden. Der Grad der Verbesserung ist von den konkret genutzten Werkzeugen zum Sammeln der Laufzeitdaten und von der Effektivität des Integrators abhängig.

Die Gründe für minderqualitative Laufzeitdaten sind vielfältig. Laufzeitdaten, welche mit nicht-invasiven Werkzeugen gesammelt werden, sind meist qualitativ minderwertiger als diejenigen von invasiven Werkzeugen, da erstere sich auf die Beobachtung des Systems beschränken müssen und letztere die benötigten Daten direkt extrahieren können. Je nach benötigter Qualität der Erkenntnisse über die Wartbarkeit sollte ein entsprechend invasives Werkzeug gewählt werden, welches die nötige

Qualität an Laufzeitdaten erzeugt. Außerdem sollte umso mehr Aufwand in die Entwicklung des Integrators investiert werden, umso qualitativ hochwertigere Erkenntnisse über die Wartbarkeit benötigt werden.

Wie in der Auswertung der Demonstration zu sehen ist, hängt die Qualität der Eingangsdaten auch von der Vielfältigkeit der gesammelten Daten ab. Bei Auswertung der ersten Gruppe wurden alle Abhängigkeiten bereits in beiden Einzelaufgaben gefunden, in der zweiten Gruppe nicht, sondern erst bei Kombination der beiden Aufgaben der zweiten Gruppe. Die Kombination der beiden Gruppen hingegen hat keine weitere Qualitätssteigerung bewirkt. Dies lässt den Schluss zu, dass eine gewisse Menge an Laufzeitdaten nötig ist, um die Metriken korrekt zu berechnen, es allerdings auch ein Maximum gibt, hinter welchem sich die Ergebnisse nicht weiter verbessern.

Die Nutzung von Laufzeitdaten hat weiterhin den Nachteil, dass in den meisten Fällen nur Erkenntnisse über jene Microservices gewonnen werden, welche auch genutzt werden. Wenn ein Microservice nicht genutzt wird und keine Laufzeitdaten erzeugt, können auch keine Metriken hierfür berechnet werden und es stehen keine Erkenntnisse zur Wartbarkeit dieses Microservices zur Verfügung. Dasselbe Problem existiert bei den einzelnen Operationen der Microservices, auch hier werden nur jene erkannt, welche auch angesprochen werden. Durch die Nutzung eines geeigneten Werkzeugs, welches Laufzeitdaten von ungenutzten Microservices/Operationen erzeugt, kann dieses Problem unter Umständen umgangen werden. Ob es ein solches Werkzeug gibt, ist vom konkreten Anwendungsbereich und den genutzten Integratoren abhängig. Es könnte hierfür beispielsweise ein Ende-zu-Ende-Testwerkzeug genutzt werden.

Die Nutzung von Laufzeitdaten zur Bestimmung der Wartbarkeit setzt voraus, dass die gesammelten Laufzeitdaten valide sind, d.h. dass kein Microservice falsche Laufzeitdaten erzeugt und dass keine gefälschten Daten in das System eingeschleust werden. Um dies zu gewährleisten kann auf bewährte Konzepte zur Integritätsprüfung zurückgegriffen werden, z.B. Verschlüsselung und Signierung. Falsche Laufzeitdaten könnten ansonsten die Ergebnisse zur Wartbarkeit in unbegrenzter Höhe verfälschen.

6.4 Eignung von Laufzeitdaten zur Bestimmung der Wartbarkeit

Durch die verteilte und dezentrale Natur von Microservice-Systemen ist die Nutzung statischer Analysemethoden nur begrenzt möglich. Bei Vorliegen des Quellcodes kann dieser analysiert werden, was durch die technologische Heterogenität in Microservice-Systemen stark erschwert wird und wodurch keine Aussagen über dynamische Attribute des Systems getroffen werden können. Aus diesem Grund muss auf Laufzeitdaten oder hybride Ansätze zurückgegriffen werden.

Die Eignung von Laufzeitdaten zur Bestimmung der Wartbarkeit von Microservice-Systemen hängt maßgeblich von den konkret genutzten Laufzeitdaten und den zu berechnenden Metriken ab. Umso komplexere Laufzeitdaten für die Berechnung der Metriken nötig sind, umso schwieriger ist die Erhebung dieser Daten. Während sich die Laufzeitdaten für einfachere Metriken oftmals durch ein Beobachten des laufenden Systems sammeln lassen, muss für komplexere Metriken das System angepasst werden, damit die benötigten Informationen aus dem System ausgelesen werden können. Die Invasivität ist hier das entscheidende Attribut: Es muss entschieden werden, ob weniger und ungenauere Erkenntnisse ausreichend sind und dafür das System nicht verändert werden muss, oder ob ein größerer Umfang und eine höhere Genauigkeit der Informationen Vorrang haben,

aber dafür das System angepasst werden muss. Der Umfang der Anpassungen hat ebenfalls eine Auswirkung auf den Umfang der erhobenen Laufzeitdaten: Allgemein lässt sich sagen, dass umso umfangreicher die Anpassungen sind, umso umfangreicher sind die erhobenen Laufzeitdaten und umso mehr verschiedene Auswertungen dieser sind möglich, was detailliertere Erkenntnisse über Wartbarkeitsprobleme und deren Ursprung liefert.

In dieser Arbeit wird das Tracingwerkzeug Zipkin beispielhaft zur Erhebung der Laufzeitdaten genutzt. Dieses Werkzeug nutzt einen invasiven Ansatz, allerdings mit einem geringen Invasivitätslevel. Es benötigt die Einbindung einer Bibliothek in jeden einzelnen Microservice, wodurch Laufzeitdaten über die Kommunikation zwischen den einzelnen Diensten gesammelt werden und diese zentral zusammengeführt und gespeichert werden.

Die Nutzung von Zipkin liefert umfangreiche Laufzeitdaten mit relativ hoher Genauigkeit, wodurch eine große Anzahl an Metriken berechnet werden kann. Durch Kombination mit anderen Integratoren, welche die Schwächen von Zipkin (zum Beispiel dass nur genutzte Dienste und Operationen erkannt werden können) ausgleichen, könnte die Genauigkeit weiter gesteigert werden. Die durch die Nutzung von Zipkin gewonnenen Daten dürften allerdings bereits für viele Anwendungszwecke genügen, da nicht immer perfekt genaue Erkenntnisse benötigt werden und Tendenzen bereits genügen.

Es lässt sich sagen, dass Laufzeitdaten generell zur Bestimmung der Wartbarkeit eines Microservice-Systems geeignet sind, allerdings der Aufwand mit steigender benötigter Genauigkeit und steigendem Umfang zunimmt.

7 Fazit

Das Ziel dieser Arbeit ist es zu untersuchen, ob sich Laufzeitdaten dazu eignen, die Wartbarkeit von Microservice-basierten Systemen zu bestimmen. Um dies zu erreichen wurde ein Framework entworfen und implementiert, welches Wartbarkeitsmetriken anhand von Laufzeitdaten berechnen kann.

7.1 Zusammenfassung

Es wurden die Architektur und die Datenmodelle des Frameworks mit Fokus auf die Erweiterbarkeit modelliert. Daraufhin wurde eine Liste mit Wartbarkeitsmetriken erstellt: Als Basis diente die Metrikliste von Bogner et al., welche aktualisiert, erweitert und auf verschiedene Eigenschaften analysiert wurde. Um die nötigen Laufzeitdaten zu erhalten wurde eine Übersicht über vorhandene Werkzeuge zum Sammeln von Laufzeitdaten erstellt, welche auch die Eigenschaften der einzelnen Werkzeuge beleuchtet. Basierend auf dieser Übersicht wurde das Werkzeug Zipkin für diese Arbeit beispielhaft ausgewählt, da es die gestellten Anforderungen am besten erfüllt. Daraufhin wurde das entworfene Framework implementiert und anschließend anhand eines Beispiel-Microservice-Systems demonstriert. Zum Schluss wurde über verschiedene Aspekte des Ansatzes diskutiert.

Die Demonstration zeigt, dass Laufzeitdaten durchaus zur Bestimmung der Wartbarkeit von Service- und Microservice-basierten Systemen genutzt werden können. Es konnten relativ umfangreiche und relativ genaue Erkenntnisse über das Beispielsystem gesammelt werden, ohne dass das System aufwendig angepasst werden musste. Die Qualität der gesammelten Laufzeitdaten ist entscheidend: Perfekte Laufzeitdaten erzeugen perfekte Erkenntnisse über die Wartbarkeit. Wenn die Laufzeitdaten allerdings nicht genau oder umfangreich genug sind, verfälschen sie die Ergebnisse. Der Grad der Invasivität der genutzten Werkzeuge ist entscheidend für die Qualität der gesammelten Laufzeitdaten: Umso invasiver ein Werkzeug ist, umso genauere und umfangreichere Laufzeitdaten sind zu gewinnen. Die Nutzung von Laufzeitdaten hat allerdings auch Nachteile: Es können nur Erkenntnisse über Dienste gewonnen werden, welche auch irgendeine Form von Laufzeitdaten erzeugen. Ungenutzte Dienste und Operationen können deshalb nicht gefunden werden.

Alles in allem eignen sich Laufzeitdaten zur Bestimmung von vielen Attributen der Wartbarkeit. Lediglich manche Wartbarkeitsattribute können mit diesem Ansatz nicht bestimmt werden, wie interne Eigenschaften der einzelnen Dienste in den Kategorien Zusammenhalt, Kopplung, Größe und Komplexität.

7.2 Ausblick

Einzelne Attribute der Wartbarkeit können mit dem vorgestellten Ansatz nicht berechnet werden. Dienste, welche keine Laufzeitdaten erzeugen, können nicht hinsichtlich ihrer Wartbarkeit bewertet werden und fehlen in systemweiten Metriken. Um dieses Problem zu lösen, könnten statische Analysemethoden, wie Quellcode-Analysen, in den Ansatz eingefügt werden. Sie könnten als zusätzlicher Integrator umgesetzt werden, wodurch das Architekturmodell nicht angepasst werden müsste. Hierdurch könnte der angesprochene Nachteil ausgeglichen werden.

Da die Forschung im Bereich Wartbarkeit fortlaufend ist und ständig neue Architekturen und Technologien entstehen, können zukünftig weitere Integratoren und Exporteure erstellt werden, welche diese neuen Technologien unterstützen. So lässt sich das Framework immer auf dem technologisch aktuellen Level halten. Falls neue Attribute zur Berechnung der Wartbarkeit gefunden werden können diese als neue Metrikmodule in das Framework eingebunden werden.

Hinsichtlich der Skalierbarkeit und der Performanz könnte das Framework stärker parallelisiert werden, um größere Laufzeitdatenmengen besser zu unterstützen. Es wäre überlegenswert, das Framework selbst zu einer Microservice-basierten Architektur weiterzuentwickeln. Die einzelnen Funktionalitäten sind inhaltlich stark voneinander abgegrenzt und ließen sich einfach in einzelne Dienste aufteilen, sodass jeder Integrator, jede Metrik und jeder Exporteur zu einem eigenen Microservice werden könnte. Durch Möglichkeiten der Choreographie wäre keine zentrale Komponente nötig. Auch ließen sich noch einfacher neue Integratoren, Metriken und Exporteure hinzufügen, abändern und entfernen.

Literaturverzeichnis

[1] Z. Qingqing and L. Xinke, *Complexity Metrics for Service-Oriented Systems*, 2009, In 2009 Second International Symposium on Knowledge Acquisition and Modeling, Vol. 3. IEEE, 375–378. DOI: https://doi.org/10.1109/KAM.2009.90

[2] M. Hirzalla, J. Cleland-Huang and A. Arsanjani, *A Metrics Suite for Evaluating Flexibility and Complexity in Service Oriented Architectures*, 2009, In Service-Oriented Computing – ICSOC 2008 Workshops: ICSOC 2008 International Workshops, Sydney, Australia, December 1st, 2008, Revised Selected Papers, Bernd J. Krämer, Kwei-Jay Lin, and Priya Narasimhan (Eds.). Lecture Notes in Computer Science, Vol. 4749. Springer Berlin Heidelberg, Berlin, Heidelberg, 41–52. DOI: https://doi.org/10.1007/978-3-642-01247-1_5

[3] M. Perepletchikov, C. Ryan, K. Frampton and Z. Tari, *Coupling Metrics for Predicting Maintainability in Service-Oriented Designs*, 2007, In 2007 Australian Software Engineering Conference (ASWEC'07). IEEE, 329–340. DOI: https://doi.org/10.1109/ASWEC.2007.17

[4] W. Jin, T. Liu, Q. Zheng, D. Cui and Y. Cai, *Functionality-oriented Microservice Extraction Based on Execution Trace Clustering*, 2018, In 2018 IEEE International Conference on Web Services (ICWS). DOI: https://doi.org/10.1109/ICWS.2018.00034

[5] H. Hofmeister and G. Wirtz, *Supporting Service-Oriented Design with Metrics*, 2008, In 2008 12th International IEEE Enterprise Distributed Object Computing Conference. IEEE, 191–200. DOI: https://doi.org/10.1109/EDOC.2008.13

[6] B. Shim, S. Choue, S. Kim and S. Park, *A Design Quality Model for Service-Oriented Architecture*, 2008, In 2008 15th Asia-Pacific Software Engineering Conference. IEEE, 403–410. DOI: https://doi.org/10.1109/APSEC.2008.32

[7] M. Perepletchikov, C. Ryan and K. Frampton, *Cohesion Metrics for Predicting Maintainability of Service-Oriented Software*, 2007, In Seventh International Conference on Quality Software (QSIC 2007). IEEE, 328–335. DOI: https://doi.org/10.1109/QSIC.2007.4385516

[8] D. Rud, A. Schmietendorf and R. R. Dumke, *Product Metrics for Service-Oriented Infrastructures*, 2006, In IWSM/MetriKon.

[9] Dynatrace LLC., *Dynatrace*, https://www.dynatrace.com/

[10] New Relic, Inc., *New Relic*, https://newrelic.com/

[11] Zipkin Team, *Zipkin*, https://zipkin.io/

[12] Jaeger Team, *Jaeger*, https://www.jaegertracing.io/

[13] J. Mace, R. Roelke, and R. Fonseca, *Pivot Tracing: Dynamic Causal Monitoring for Distributed Systems*, 2015, USENIX Annual Technical Conference (USENIX ATC 16). ACM Press, pp. 378–393.

[14] HTrace Team, *Apache HTrace*, http://incubator.apache.org/projects/htrace.html

[15] B. H. Sigelman, L. A. Barroso, M. Burrows, P. Stephenson, M. Plakal, D. Beaver, S. Jaspan, and C. Shanbhag, *Dapper, a Large-Scale Distributed Systems Tracing Infrastructure*, 2010, https://research.google.com/archive/papers/dapper-2010-1.pdf

[16] Prometheus Team, *Prometheus*, https://prometheus.io/

[17] T. Engel, M. Langermeier, B. Bauer, A. Hofmann, *Evaluation of Microservice Architectures: A Metric and Tool-Based Approach*, 2018, Mendling, J. and Mouratidis, H. (eds.) Lecture Notes in Business Information Processing. pp. 74–89. Springer International Publishing, Cham

[18] Tillmann Bielefeld, Peer Brauer, Thomas Düllmann et al., *Kieker*, http://kieker-monitoring.net/

[19] B. Mayer, R. Weinreich, *An Approach to Extract the Architecture of Microservice-Based Software Systems*, 2018, IEEE Symposium on Service-Oriented System Engineering (SOSE). pp. 21–30.

[20] G. Granchelli, M. Cardarelli, P. Francesco, I. Malavolta, L. Iovino and A. Salle *Towards Recovering the Software Architecture of Microservice-Based Systems*, 2017, IEEE International Conference on Software Architecture Workshops (ICSAW). pp. 46–53.

[21] Tcpdump Team, *Tcpdump*, http://www.tcpdump.org/

[22] S. Rigal, P. van Eck, G. Wijnholds, R. van der Leek, J. Visser, *Building Maintainable Software, Java Edition*, 2016, http://shop.oreilly.com/product/0636920049159.do

[23] R. Rodger et al., *Seneca*, http://senecajs.org/

[24] R. Rodger et al., *ramanujan*, https://github.com/senecajs/ramanujan

[25] Node.js Foundation, *Node.js*, https://nodejs.org/

[26] J. Bogner, S. Wagner and A. Zimmermann, *Automatically measuring the maintainability of service- and microservice-based systems*, 2017, In: Proceedings of the 27th International Workshop on Software Measurement and 12th International Conference on Software Process and Product Measurement on - IWSM Mensura '17. pp. 107–115.

[27] E. Brewer, *Towards robust distributed systems*, 2000, (Invited Talk) Principles of Distributed Computing, Portland, Oregon

[28] J. Chhabra and V. Gupta, *A Survey of Dynamic Software Metrics*, 2010, In: Journal of Computer Science and Technology 25(5): pp. 1016–1029, DOI: 10.1007/s11390-010-1080-9

[29] S. Yacoub, T. Robinson and H. Ammar, *Dynamic metrics for object oriented designs*, 1999, In: Proceedings Sixth International Software Metrics Symposium, DOI: 10.1109/METRIC.1999.809725

[30] B. Boehm, *The TRW Series of Software Technology: Characteristics of Software Quality*, 1978

[31] N. Kukreja, *Measuring Software Maintainability*, 2017, https://quandarypeak.com/2015/02/measuring-software-maintainability/

[32] M. Frappier, S. Matwin and A. Mili, *Software Metrics for Predicting Maintainability*, 1994, http://www.dmi.usherb.ca/ frappier/Papers/tm2.pdf

[33] J. Bogner, A. Zimmermann and S. Wagner, *Analyzing the Relevance of SOA Patterns for Microservice-Based Systems*, 2018, 10th Central European Workshop on Services and their Composition (ZEUS'18), Dresden, Germany, http://ceur-ws.org/Vol-2072/paper2.pdf

[34] S. Newman, *Building Microservices*, 2015, ISBN: 9781491950340

[35] J. Thones, *Microservices*, 2015, DOI: 10.1109/MS.2015.11

[36] International Organization for Standardization, *ISO/IEC 25010:2011*, 2011, https://www.iso.org/standard/35733.html

[37] K.Bennett and V. Rajlich, *Software maintenance and evolution: a roadmap*, 2000, In ICSE '00 Proceedings of the Conference on The Future of Software Engineering pp. 73-87, DOI: 10.1145/336512.336534

[38] M. Endrei, J. Ang, A. Arsanjani, S. Chua, P. Comte, P. Krogdahl, M. Luo und T. Newling, *IBM: Patterns: service-oriented architecture and web services.*, 2004, ISBN: 9780738453170, https://www.redbooks.ibm.com/redbooks/pdfs/sg246303.pdf

[39] Apache Software Foundation, *Apache Maven*, https://maven.apache.org/

[40] C. Jones and O. Bonsignour, *The Economics of Software Quality*, 2011, Pearson Education

[41] B. Lientz and E. Swanson, *Software Maintenance Managemen*, 1980, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA

[42] Pivotal Software, *Spring Framework*, https://spring.io/

[43] Pivotal Software, *RabbitMQ*, https://www.rabbitmq.com/

[44] Apache Software Foundation, *Apache Kafka*, https://kafka.apache.org/

[45] A. Arsanjani, *Service-oriented modeling and architecture*, 2004, IBM developer works

[46] M. Klems, D. Bermbach and R. Weinert, *A Runtime Quality Measurement Framework for Cloud Database Service Systems*, 2012, Eighth International Conference on the Quality of Information and Communications Technology, Lisbon, pp. 38-46. DOI: 10.1109/QUATIC.2012.17

[47] T. McCabe, *A Complexity Measure*, 1976, IEEE Transactions on Software Engineering 2, 4 , 308–320. DOI: https://doi.org/10.1109/TSE.1976.233837

[48] M. Halstead, *Elements of Software Science (Operating and Programming Systems Series)*, 1977, Elsevier Science Inc., New York, NY, USA.

[49] W. Li and S. Henry, *Object-oriented metrics that predict maintainability*, 1993, Journal of Systems and Software 23, 2 (Nov 1993), 111–122. DOI: https://doi.org/10.1016/0164-1212(93)90077-B

[50] S. Chidamber and C. Kemerer, *A metrics suite for object oriented design*, 1994, IEEE Transactions on Software Engineering 20, 6 , 476–493. DOI: https://doi.org/10.1109/32.295895 arXiv:1011.1669

[51] S. Sarkar, G. Rama, and A. Kak, *API-based and information-theoretic metrics for measuring the quality of software modularization*, 2007, IEEE Transactions on Software Engineering 33, 1 , 14–32. DOI: https://doi.org/10.1109/TSE.2007.256942

[52] M. Lindvall, R. Tvedt and P. Costa, *An empirically-based process for software architecture evaluation*, 2003, Empirical Software Engineering 8, 1 , 83–108. DOI: https://doi.org/10.1023/A:1021772917036

[53] L. O'Brien, C. Stoermer, and C. Verhoef, *Software architecture reconstruction: Practice needs and current approaches*, 2002

[54] L. O'Brien and C. Stoermer, *Architecture reconstruction case study*, 2003, https://resources.sei.cmu.edu/library/asset-view.cfm?assetid=6431

[55] N. Alshuqayran, N. Ali, and R. Evans, *Towards Micro Service Architecture Recovery: An Empirical Study*, 2018, In 2018 IEEE International Conference on Software Architecture (ICSA). IEEE.

[56] M. Cardarelli, L. Iovino, P. Di Francesco, A. Di Salle, I. Malavolta and P. Lago, *An Extensible Data-driven Approach for Evaluating the Quality of Microservice Architectures*, 2019, In Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing, DOI: https://doi.acm.org/10.1145/3297280.3297400

[57] A. Ahmad, P. Jamshidi and C. Pahl, *Graph-Based Pattern Identification from Architecture Change Logs*, 2012, In: Bajec M., Eder J. (eds) Advanced Information Systems Engineering Workshops. CAiSE 2012. Lecture Notes in Business Information Processing, vol 112. Springer, Berlin, Heidelberg

[58] J. Cito, P. Leitner, H. Gall, A. Dadashi, A. Keller and A. Roth, *Runtime metric meets developer: building better cloud applications using feedback*, 2015, In: 2015 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward!) - Onward! 2015. pp. 14–27. ACM Press, New York, New York, USA

[59] B. Mayer and R. Weinreich, *A Dashboard for Microservice Monitoring and Management*, 2017, 2017 IEEE International Conference on Software Architecture Workshops (ICSAW), Gothenburg, pp. 66-69, DOI: 10.1109/ICSAW.2017.44

[60] SmartBear Software, *Swagger*, https://swagger.io/

Alle URLs wurden zuletzt am 12. 06. 2019 geprüft.

Erklärung

Ich versichere, diese Arbeit selbstständig verfasst zu haben. Ich habe keine anderen als die angegebenen Quellen benutzt und alle wörtlich oder sinngemäß aus anderen Werken übernommene Aussagen als solche gekennzeichnet. Weder diese Arbeit noch wesentliche Teile daraus waren bisher Gegenstand eines anderen Prüfungsverfahrens. Ich habe diese Arbeit bisher weder teilweise noch vollständig veröffentlicht. Das elektronische Exemplar stimmt mit allen eingereichten Exemplaren überein.

Ort, Datum, Unterschrift