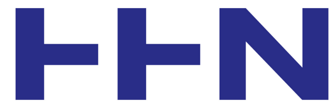

Bachelorarbeit

im Studiengang Medizinische Informatik

Bedeutung von Telemetrie für den Software Development Life Cycle

---

Michael Graf

Matr.-Nr: 199193

mgraf@stud.hs-heilbronn.de

21. September 2020

Referent:	Prof. Dr.-Ing. Andreas Heil
Koreferent:	Prof. Dr.-Ing. Andreas Mayer
Betreuer:	Richard Zowalla M.Sc.

Abstract

Die heutige Software-Entwicklung ist davon geprägt, dass Anwendungen immer komplexer und aufwändiger werden. Gleichzeitig steigen die Erwartungen der Kunden an die Qualität der Software.

Für die Software-Entwickler ist die Telemetrie zu einem unverzichtbaren Werkzeug geworden. Sie ist ein wesentlicher Baustein, um die Beobachbarkeit („Observability“) von Applikationen zu erhöhen und somit die Grundlage für eine bessere Qualität in der Software Entwicklung. Hierbei ist zu analysieren, welche Bedeutung die Telemetrie für den gesamten Software Development Life Cyle hat.

Der aktuelle Stand der Software-Telemetrie wird am Beispiel des Projektes OpenTelemetry dargelegt.

OpenTelemetry hat sich zum Ziel gesetzt, die universelle Plattform für den Austausch von Telemetriedaten zu werden.

Die Ergebnisse des Projektes OpenTelemetry werden analysiert und bewertet.

Inhaltsverzeichnis

Abbildungsverzeichnis I

Tabellenverzeichnis II

Abkürzungsverzeichnis III

1 Einleitung 1

1.1 Gegenstand und Motivation . . . 1
1.2 Problemstellung der Arbeit . . . 2
1.3 Ziel der Arbeit . . . 2
1.4 Aufbau der Arbeit . . . 3

2 Stand der Wissenschaft 4

2.1 Telemetrie . . . 4
2.1.1 Traditionelle Werkzeuge . . . 5
2.1.2 Neue Anforderungen . . . 6
2.2 Software Development Life Cycle (SDLC) . . . 6
2.3 Observability . . . 8
2.3.1 Logging . . . 11
2.3.2 Tracing und Span . . . 11
2.3.3 Metriken . . . 12
2.3.4 Monitoring . . . 13
2.4 Nutzungsdaten . . . 13
2.5 Zusammenfassung . . . 14

3 Analyse 15

3.1 Telemetrie im Alltag . . . 15
3.1.1 Windows 10 . . . 15
3.1.2 Microsoft Office . . . 19
3.2 Evaluierungskriterien . . . 19
3.3 Das Open-Source-Projekt OpenTelemetry . . . 20
3.3.1 OpenCensus . . . 21
3.3.2 OpenTracing . . . 23
3.4 Architektur von OpenTelemetry . . . 24
3.4.1 OpenTelemetry API . . . 24
3.4.2 OpenTelemetry SDK . . . 31

3.5 Monitoring-Tools . . . 34
3.5.1 Jaeger . . . 34
3.5.2 Azure Monitor . . . 37
3.6 OpenTelemetry in der Praxis . . . 38
3.6.1 Hypertext Transfer Protocol (HTTP)-Transaktion . . . 38
3.6.2 Java Auto-Instrumentation . . . 41
3.7 Telemetrie im Software Development Life Cycle (SDLC) mit Continuous Integration / Continuous Delivery (CI/CD)-Pipeline . . . 44

4 Ergebnis **46**
4.1 Evaluation . . . 46
4.1.1 Interoperabilität . . . 46
4.1.2 Plattformunabhängigkeit . . . 47
4.1.3 Standardisierung . . . 47
4.1.4 Erweiterbarkeit . . . 48
4.1.5 Dokumentation . . . 49
4.1.6 Mehrwert . . . 49
4.2 Bewertung der Evaluation . . . 50
4.3 Telemetrie & SDLC . . . 50

5 Diskussion & Ausblick **53**
5.1 Diskussion . . . 53
5.2 Ausblick . . . 53

Literatur **55**

Abbildungsverzeichnis

2.1 Ablauf des ELK-Stack 5
2.2 Software Development Life Cycle 7
2.3 Prozesskette des DevOps-Ansatz [15] 9
2.4 Die Säulen der Observability 10
2.5 Beispiel eines Traces mit Spans 12

3.1 Aktivierung von Diagnosdaten in Microsoft 16
3.2 Aufbau von OpenCensus 22
3.3 Aufbau von OpenTelemetry 25
3.4 Funktionsweise von Context and Propagation 30
3.5 Zusammenspiel der einzelnen APIs 31
3.6 Jaeger User Interface (UI)-Trace Search 35
3.7 Jaeger UI-Detail 36
3.8 Aufbau von Azure Monitor [43] 37
3.9 Trace Ablaufverfolgung 43
3.10 CI/CD-Pipeline mit Observability 44
3.11 Gitlab-Einstellungen für Telemetriedaten 45

4.1 Telemetrie in Verbindung mit DevOps 51

5.1 Aufbau des Context von Heute und in Zukunft 54

Tabellenverzeichnis

3.1 Verfügbare Exporter für Monitoring Tools in der jeweiligen Programmiersprache 33

Abkürzungsverzeichnis

APM Application Performance Monitoring
ELK-Stack `Elasticsearch, Logstash und Kibana`
SDLC Software Development Life Cycle
SDK Software Development Kit
API Application Programming Interface
CI/CD Continuous Integration / Continuous Delivery
HTTP Hypertext Transfer Protocol
JSON JavaScript Object Notation
RPC Remote Procedure Call
UI User Interface
URL Uniform Resource Locator

1 Einleitung

1.1 Gegenstand und Motivation

Unter dem Begriff Telemetrie wird allgemein die Übertragung von Messwerten eines an einem Messort befindlichen Sensors zu einer Empfangsstelle, die sich an einem anderen Ort befindet, verstanden [1]. Für die heutige Industrie ist die Telemetrie von großer Bedeutung. Ihr Einsatzgebiet ist vielfältig, sie reicht von der ölfördernden Industrie bis hin zur Formel 1. Sie ist ein wichtiger Bestandteil geworden, um Informationen in einer gegebenen Situation zu erhalten und auf diese schnellst möglichst reagieren zu können. Auch in der Software-Entwicklung spielt die Telemetrie eine wichtige Rolle. Die heutige Software-Entwicklung ist davon geprägt, dass Anwendungen immer komplexer und aufwändiger werden. Gleichzeitig erwarten die Kunden Produkte mit hoher Qualität. Dabei muss aber die Relation zwischen benötigter Zeit und Kosten berücksichtigt werden.

Die frühere Software-Entwicklung folgte meist einem linearen Vorgehensmodell, wie zum Beispiel dem Wasserfallmodell oder dem V-Modell. Bei diesen Modellen besitzt jede Phase einen eindeutigen Start- und Endpunkt mit einem definierten Ergebnis. Die Nachteile dieser Vorgehensweisen sind jedoch, dass es einen geringen Spielraum für Anpassungen im Projektablauf gibt oder Endanwender erst nach dem Produktionsprozess eingebunden werden.

Um auf die zunehmende Komplexität und die steigenden Qualitätsanforderungen reagieren zu können, entstand der Ansatz der agilen Softwareentwicklung. Durch diese Vorgehensweise soll den Software-Entwicklern die Möglichkeit gegeben werden, größere und komplexere Systeme schneller entwickeln und besser auszuliefern zu können.

Um das zu entwickelnde System in seinem Programmablauf überwachen zu können, spielt die Telemetrie eine wichtige Rolle. Das Konzept der Nachverfolgung bei der Ausführung einer Anwendung ist nicht neu. Die Fähigkeit, das Verhalten eines Programms zu überwachen, ist jedoch keine triviale Aufgabe. Um dieses Verhalten leichter und besser analysieren zu können, wurden Konzepte wie das Profiling und Debugging entwickelt. [2]

Moderne Software-Architekturen meist basierend auf verteilten Systemen oder Microservices [3] können jedoch nicht mehr ausreichend mit den oben genannten klassischen Methoden wie Profiling und Debugging analysiert und überwacht werden. [2]

1.2 Problemstellung der Arbeit

Entwickler benötigen Informationen über technische Fehlfunktionen in verteilten Systemen und Anwendungen, um schnell reagieren und Ausfälle verhindern zu können. Moderne Softwareentwicklung setzt zudem zunehmend auf die Verwendung von neuen Technologien wie Mikroservices, um eine flexible Architekturentwicklung zu unterstützen. Mikroservices laufen entweder auf verschiedenen Maschinen oder in unterschiedlichen Containern. Dabei kommen auch verschiedene Technologien und/oder Programmiersprachen zum Einsatz. Gerade bei einer verteilten Software Architektur muss die Qualitätssicherung von Anfang an ein fester Bestandteil des Entwicklungsprozesses sein. Dies zu gewährleisten, ist eine Herausforderung. Wenn hunderte von Mikroservices miteinander interagieren, ist es nicht trivial, die Ursache von Fehler zu finden. Um diese Problematik in Griff zu bekommen spielen Telemetriedaten, die durch zentralisierte Werkzeuge koordiniert werden, eine wichtige Rolle.
Wie in Kapitel 1.1 angedeutet, steigt die Bedeutung der Überwachung eines Systems in der heutigen Software-Entwicklung stark an. Die Beobachbarkeit eines Systems wird in der Software-Entwicklung als Observability bezeichnet. Trotz der hohen Software-Komplexität dürfen die Qualitätsanforderungen an Benutzerfreundlichkeit, hohe Verfügbarkeit oder Stabilität nicht in den Hintergrund rücken. Dieses hohe Ziel zu erreichen, erweist sich als nicht trivial. Um die Observability zu erhöhen und damit den Entwicklern bei der Verfügbarkeit und Stabilität der Anwendung zu unterstützen, muss verstärkt Telemetrie zum Einsatz kommen.

1.3 Ziel der Arbeit

Es soll im Rahmen dieser Bachelorarbeit dargestellt werden, welchen Mehrwert Telemetrie für den SDLC bringt, wie Telemetrie die Software-Entwicklung unterstützen kann und welches Entwicklungspotenzial noch in diesem Themenbereich steckt. Es soll dargestellt werden, wie man die Observability von Applikationen mit den aktuell vorhandenen Tools bzw. Werkzeugen verbessern kann und wie Standards bei dieser Umsetzung helfen können. Dabei soll der Fokus auf das Open-Source Projekt `OpenTelemetry` gelegt werden. Kann dieses Projekt durch die Etablierung von Standards und durch einfache Integration in Applikationen den Software-Hersteller das Erzeugen und Weiterverarbeiten von Telemetriedaten erleichtern? Des Weiteren soll

auch ein Blick auf Monitoring-Systeme gelegt werden. Es soll dargestellt werden, welche Arten von Monitoring-Software es gibt und wie durch das Weiterverarbeiten und Kombinieren von Telemetriedaten Probleme schneller erkannt und behoben werden können.

1.4 Aufbau der Arbeit

- Kapitel 1 beschreibt die Problemstellung und das Ziel der Arbeit
- Kapitel 2 dient als Einführung in die Grundlagen der Software-Telemetrie
- Kapitel 3 dient der Analyse verfügbarer Datenquellen, aus denen die nötigen Informationen für die Telemetrie geschöpft werden. Weiterhin wird das Grundkonzept von OpenTelemetry dargestellt
- Kapitel 4 dient der Evaluierung des Open-Source-Projekts OpenTelemetry
- Kapitel 5 bietet eine kritische Bewertung der Ergebnisse und einen Ausblick auf zukünftige Einsatzbereiche für die Software-Telemetrie

2 Stand der Wissenschaft

Dieses Kapitel beschäftigt sich mit den grundlegenden Begriffen rund um die Telemetrie.

2.1 Telemetrie

Telemetrie ist ein weitgefasster Begriff, der in vielen verschiedenen Themenbereichen verwendet wird. Er setzt sich aus den altgriechischen Worten „fern“ und „Maß“ zusammen. Unten den Begriff Telemetrie wird die Übertragung von Messwerten eines an einem Messort befindlichen Sensors zu einer Empfangsstelle, die sich an einem anderen Ort befindet, verstanden [1].
Der Ursprung von Telemetrie geht auf das 19. Jahrhundert zurück. Die ersten Aufzeichnungen stammen aus dem Jahr 1845, in dem eine der ersten Datenübertragungsleitungen zwischen dem Winterpalast des russischen Zaren und dem Hauptquartier der Armee entwickelt wurde. Einige Jahre später 1874 ließen französische Ingenieure auf dem Mont Blanc ein System von Wetter- und Schneehöhensensoren bauen, um Echtzeitinformationen nach Paris zu übertragen [4]. Seit dieser Zeit ist Telemetrie aus unserem Alltag nicht mehr wegzudenken.
Der Begriff Telemetrie im Kontext von Software trat zum ersten Mal im Jahre 1973 in dem Dokument der NASA „The Deep Space Network“ in Erscheinung. In diesem Bericht wurden die Fortschritte in den Bereichen Flugprojektunterstützung, Netzwerktechnik, Hardware- und Softwareimplementierung und Betrieb dargelegt. In diesem Artikel wurde Telemetrie wie folgt beschrieben:

> *Sammelt neu erfasste Daten zur Übertragung an den Benutzer in Echtzeit. Überwacht funktional alle telemetriebezogenen Baugruppen (Softwarekonfigurationen) und meldet dem Benutzer ihren Status.*[5]

Telemetrie wird heutzutage intensiv in Softwareprodukten eingesetzt. Die bekannteste und älteste Form, wie Telemetriedaten erzeugt werden können, ist das Logging, das im nachfolgenden Kapitel am Beispiel `Elasticsearch, Logstash und Kibana` (ELK-Stack) vorgestellt wird.

2.1.1 Traditionelle Werkzeuge

Ein bekanntes Beispiel, wie Logging-Daten schnell und effizient gewonnen und ausgewertet werden können, sind die Programme Elasticsearch, Logstash und Kibana. Bei ELK-Stack handelt es sich um drei Open-Source-Projekte, die einen Worklflow, wie in Abbildung 2.1 zu sehen, bilden, um Log-Nachrichten zu sammeln, zu speichern, zu analysieren und zuletzt zu visualisieren. Im Gegensatz zur Abkürzung ist der Datenfluss im ELK-Stack nicht Elasticsearch ⇒ Logstash ⇒ Kibana, sondern wie in Abbildung 2.1 zu sehen Logstash ⇒ Elasticsearch ⇒ Kibana.

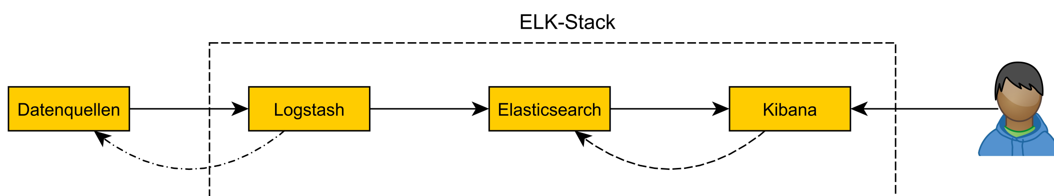

Abbildung 2.1: Ablauf des ELK-Stack

Logstash ist ein Open-Source-Projekt, das ursprünglich für das Sammeln von reinen Logdateien konzipiert wurde, aber heutzutage durch den Einsatz von Plugins die Möglichkeit bietet, viele verschiedene Datentypen zu verarbeiten. Diese Daten können anschließend gefiltert, transformiert und weitergeleitet werden. [6, 7, 8]
Elasticsearch ist eine Suchmaschine, die im Jahr 2010 vom Unternehmen Elastic veröffentlicht wurde. Sie basiert auf Apache Lucene, eine Programmbibliothek zur Volltextsuche. Diese Bibliothek ermöglicht eine vollständige Textsuche in verschiedenen Arten von strukturierten und unstrukturierten Daten wie Textdaten und numerischen Daten. Sobald die Rohdaten an Elasticsearch gesendet werden, werden diese indexiert. Während des Indexierungsprozesses speichert Elasticsearch die Dokumente und erstellt gleichzeitig einen invertierten Index, damit die Dokumentdaten in Echtzeit durchsucht werden können.[8, 9]
Kibana ist ein Tool zum Visualisieren und Verwalten von Daten aus Elasticsearch und bietet aktualisierte Histogramme, Liniendiagramme, Tortendiagramme und Landkarten in Echtzeit an. Darüber hinaus enthält Kibana weiterführende Anwendungen, wie zum Beispiel Canvas zum Erstellen individuell angepasster dynamischer Infografiken

auf der Basis der Daten des Nutzers und Elastic Maps zum Visualisieren von Geodate [9].

2.1.2 Neue Anforderungen

ELK-Stack ist ein effizientes Tool um Log-Daten zu sammeln und auszuwerten. Es ist einfach anzuwenden, ist plattformunabhängig und skaliert sehr gut. Bei verteilten Systemen stoßen die herkömmlichen Überwachungstools wie ELK-Stack an ihre Grenzen, diese können die Beobachtbarkeit nur für das gesamte System gewährleisten. Entwickler müssen jedoch in der Lage sein, zu jeder Zeit in Teilbereiche von verteilten Umgebungen zoomen zu können, um Fehler und Performanceprobleme exakt zu lokalisieren. [2, 10]
Dieser Thematik stellt sich das Open-Source-Projekt `OpenTelemetry`, das in Kapitel 3.3 detailliert vorgestellt wird.

2.2 Software Development Life Cycle (SDLC)

Das Wasserfall-Modell von Royce aus dem Jahr 1970 war das erste öffentlich dokumentierte Life-Cycle-Modell. Das Modell wurde entwickelt, um mit der zunehmenden Komplexität von Luft- und Raumfahrtprodukten umgehen zu können. Der nächste revolutionäre neue Blick auf den Entwicklungslebenszyklus war das 1985 von Boehm vorgestellte „Spiralmodell“. Das Spiralmodell konzentriert sich auf das Risikomanagement. Life-Cycle-Modelle beschreiben die Wechselbeziehungen zwischen Softwareentwicklungsphasen. Die Technologie entwickelt sich jedoch schnell weiter, die Systeme werden immer komplexer und die Benutzer haben sich an gut funktionierende Technologie gewöhnt. Es wurden daher Modelle und Frameworks entwickelt, um Unternehmen durch einen organisierten Lebenszyklus der Systementwicklung zu führen. Die traditionellen Ansätze zur Entwicklung von Technologiesystemen wurden angepasst, um den sich ständig ändernden, komplexen Anforderungen jeder einzelnen Organisation und ihrer Benutzer gerecht zu werden. [11]
1995 wurde die Norm ISO/IEC 12207 „Systems and software engineering - Software life cycle processes“ veröffentlicht [12]. Sie dient dazu, ein besseres Verständnis über die Produktion von Software und der zugehörigen Serviceleistungen zu erlangen. Durch

das verbesserte Verständnis sollten Verhandlungen und Verträge zwischen Kunden und Lieferanten von Software-Projekten in Hinblick auf die Entwicklung, den Betrieb und die Wartung von Softwaresystemen vereinfacht werden.
Der SDLC besteht aus sechs Phasen: Planung, Analyse, Design, Implemetierung, Test & Integration und Wartung & Instandhaltung.

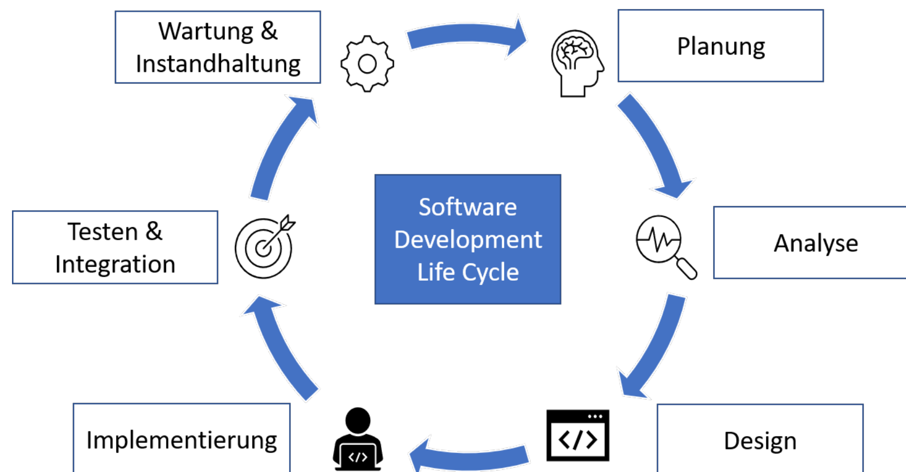

Abbildung 2.2: Software Development Life Cycle

Wie in Abbildung 2.2 gezeigt,beginnt der SDLC mit einer Planungsphase. Das Ziel der Planungsphase ist es, die Software zu definieren und eine Machbarkeitsstudie in den folgenden Punkten durchzuführen: technisch, operativ, finanziell, personell und rechtlich. Nach Abschluss der Planung werden die Anforderungen an die Software analysiert und definiert. Ziel dieser Analyse-Phase ist es, alles von der Planungsphase bis zur Recherche zu erfassen und detailliert zu dokumentieren. Mit Abschluss dieser Phase ist das Lastenheft wie auch das Pflichtenheft fertiggestellt. Die anschließende Designphase orientiert sich eng an diesen Lasten- und Pflichtenheften.
Ziel der Designphase ist es, für jede definierte Anforderung einen umsetzbaren Plan zu erstellen, der später implementiert wird. Die Implementierung verantwortet die Umsetzung der geplanten Aufgaben aus der Designphase. Dies ist die längste Phase des SDLC-Modells. Das wesentliche Ziel der Phase ist, ein Softwaresystem bereitzustellen, das alle vom Kunden definierten Anforderungen erfüllt. In der Phase „Integration & Testing“ des SDLC wird sichergestellt, dass die implementierte Software aus der Ent-

wicklungsphase alle Kundenanforderungen erfüllt, korrekt funktioniert und keine Fehler aufweist. Ziel ist es, eine gut funktionierende Software bereitzustellen, die sorgfältig getestet und kontinuierlich aktualisiert wird, um die bestmögliche Qualität sicherzustellen. Im letzten Schritt wird die Software veröffentlicht, wenn diese die Anforderungen des Auftraggebers erfüllt. Meist wird eine Software zunächst als Betatest auf den Markt gebracht. Das Support-Team sammelt das Feedback der ersten Benutzer. Werden Fehler gemeldet, werden diese vom Entwicklungsteam umgehend behoben. Danach wird die finale Version ausgerollt. Software-Updates werden regelmäßig in der Wartungsphase implementiert, um insbesondere sicherzustellen, dass die Software auf den aktuellen Sicherheitsstand ist. Weiterhin werden in Software-Updates neue gewünschte Funktionalitäten bereitgestellt.
Zu den am SDLC beteiligten Personen gehören auch die Führungskräfte, aber es sind die Projekt-/ Programmmanager, Software- und Systemingenieure, Benutzer und das Entwicklungsteam, die den mehrschichtigen Prozess abwickeln. Jedes Projekt hat seine eigene Komplexität hinsichtlich Planung und Ausführung. Oft verwenden Projektmanager innerhalb einer Organisation unterschiedliche SDLC-Methoden. Selbst bei gleicher Methode können sich die Projektwerkzeuge und -techniken erheblich unterscheiden. [11]

2.3 Observability

Bereits kleine Änderungen in der Anfangsphase der Systementwicklung können langfristig im SDLC zu nicht vorhersehbaren Ergebnissen führen. Dieses Verhalten wird in der Software-Entwicklung als sogenannter „Schmetterlingseffekt" bezeichnet. [13]
Um diesen Auswirkungen entgegenzuwirken, entstand der agile Ansatz des SDLC. Das Ziel dieser Vorgehensweise ist die Transparenz und Flexibilität im Softwareentwicklungsprozess zu erhöhen und dadurch Risiken in der Entwicklung zu minimieren. Auftraggeber und Entwickler stehen im engen Austausch, wodurch eine kontinuierliche Verbesserung an den Arbeitsprozessen stattfindet. Dieser agile Ansatz wird mit dem DevOps Ansatz konsequent weitergeführt.
Der DevOps-Ansatz ist kein vorgegebenes Schema wie das Wasserfallmodell, sondern ist vielmehr ein Teil der Unternehmenskultur. Durch einheitliche Tools und Prozesse soll eine effektivere und effizientere Zusammenarbeit ermöglicht werden, dabei soll es eine enge Verzahnung zwischen Planung, Entwicklung, Testing und Vertrieb geben.[14]

Abbildung 2.3: Prozesskette des DevOps-Ansatz [15]

Der Software-Lifecycle des DevOps-Ansatzes, wie in Abbildung 2.3 dargstellt, besteht aus verschiedenen Phasen, wie kontinuierlicher Entwicklung, kontinuierlicher Integration, kontinuierlichem Testen, kontinuierlicher Bereitstellung und kontinuierlicher Überwachung. Diese Stufen werden fortlaufend in einer Schleife ausgeführt, bis das gewünschte Ziel erreicht wurde. [16]
In der Phase kontinuierlicher Überwachung müssen Daten über CPU-Leistung, Speicherbedarf, Datendurchfluss und Bandbreite gesammelt werden. Dies ist auch unter den Begriff Application Performance Monitoring (APM) bekannt. Diese Daten werden genutzt, um negative Einflüsse und Probleme bei der Anwendungs-Performance zu erkennen und zu beseitigen. In dieser Phase reicht es heutzutage nicht mehr aus, nur APM-Daten zu sammeln. Hier kommt der Begriff Observability ins Spiel. Dieser steht für einen umfassenden Ansatz, der zahlreiche Faktoren für die Überwachung und Beobachtung des Verhaltens von Software mit einbezieht. Dabei ersetzt die Observability nicht das Monitoring, sondern Monitoring ist eine Teilmenge der Observability geworden. [17]
Um Observability in der modemen Softwareentwicklung zu erreichen, ist die Telemetrie zu einer entscheidenden Größe geworden. Das wird in der Software-Entwicklung als „Der zentrale chronische Konflikt“ bezeichnet [18].
Um diesen Konflikt so klein wie möglich zu halten, spielt die Observability beim DevOps-Ansatz in der Software-Entwicklung eine zentrale Rolle.
Die Observability ist eine Eigenschaft eines Systems, die zum Zeitpunkt des Systemdesigns in einem System verankert werden muss. Dabei ist zu berücksichtigen, dass die Observability stetig weiterentwickelt, getestet, bereitgestellt, betrieben, überwacht

und gewartet werden muss. [19]
Um eine gute Observability zu erreichen, haben vier Elemente einen Einfluss auf diese, wie in Abbildung 2.4 dargestellt. In der Literatur findet man meist nur die drei Elemente Logs, Metriken und Traces. Die Praxis zeigt jedoch, dass nur mit einem geeignetem Monitoring (Visualisation) der Zusammenhang der gelieferten Daten schnell und übersichtlich dargestellt werden kann.

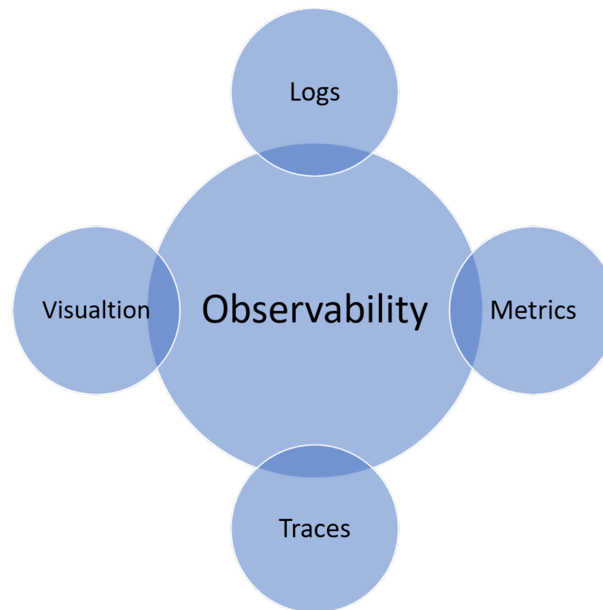

Abbildung 2.4: Die Säulen der Observability

Logs (2.3.1), Metriken (2.3.3), Traces (2.3.2) und Monitoring (2.3.4) bilden die vier Säulen der Observability. Ziel ist es, Anwendungen und Systeme exakt zu beobachten, Informationen über technische Fehlfunktionen zu erhalten und schnell an die Verantwortlichen zu übermitteln. Logfiles, Informationen zur Ressourcennutzung und Anwendungs-Traces sollen den Administratoren und Entwicklern dabei helfen, die Fehlerursachen zu erkennen, die Probleme zu beheben und künftige Ausfälle nach Möglichkeit zu vermeiden. [19]

2.3.1 Logging

Das Logging (Protokollieren) wird in der Informatik als automatisches Aufzeichnen von Informationen über Programmzustände während der Ausführung einer Software bezeichnet. Es dient dazu Fehlerzustände zu identifizieren, um später den Ablauf und Zustände von Programmen und Prozessen rekonstruieren und verstehen zu können. In den meisten Programmiersprachen sind bereits Logging-Application Programming Interface (API) vorhanden. Durch diese lassen sich Meldungen auf die Konsole oder auf externe Speicher wie Text- oder XML-Dateien oder in Datenbanken schreiben. [20] Diese Logging-Daten dürfen keine vertraulichen Informationen beinhalten. Der typische Aufbau einer Log-Datei ist zeilenorientiert. Jede Zeile entspricht einem einzelnen Ereignis. Jedes Ereignis besteht aus weiteren Informationselementen. Diese Informationselemente beschreiben die Wichtigkeit der Meldung: Warnung, Fehler, Information. Außerdem gibt es zu jedem Ereignis einen Informationstext über die Art des Ereignisses und wann das Ereignis aufgetreten ist, auch Zeitstempel genannt. Die bekanntesten Formate für Log-Files sind NCSA, W3C Extended Log File Format und IIS Log File Format. [21, 22]

2.3.2 Tracing und Span

Multithreadanwendungen und verteilte Anwendungen (Microservices) sind schwierig zu debuggen. Durch das Einschalten eines Debuggers wird in der Regel das Verhalten geändert [23]. Unter Tracing, auf Deutsch Rückverfolgung, versteht man in der Programmierung eine zeitliche Ablaufverfolgung einer speziellen Anwendung oder Funktion. Jede einzelne Anfrage innerhalb eines Traces wird als „Span“ bezeichnet. Es beschreibt die Spanne zwischen Anfrageversand durch die Anwendung und die Request-Antwort durch den Server [24]. Ein Trace besteht typischerweise aus vielen verschiedenen Spans. Die Informationen des Traces werden von Programmierern zu Debugging-Zwecken verwendet. Durch diese Art des Debuggens werden Prozesse und Workflows zwischen Services und Anwendung gut sichtbar gemacht.

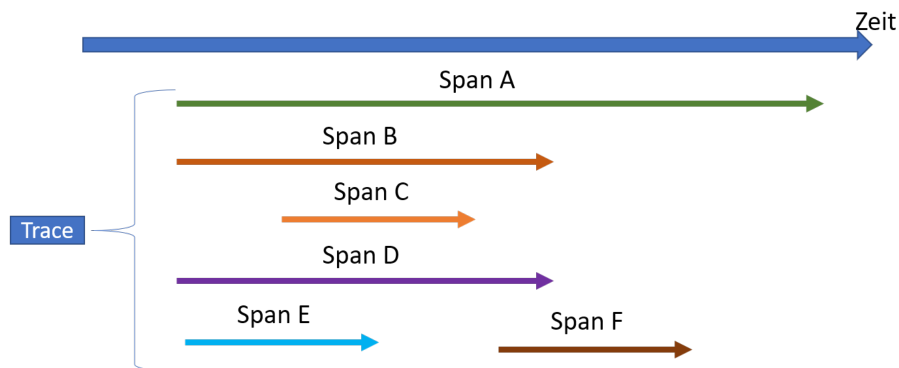

Abbildung 2.5: Beispiel eines Traces mit Spans

Ein Span, der keinem übergeordneten Span besitzt, wird als „Root Span“ bezeichnet. In Abbildung 2.5 stellt „Span A“ diesen dar. Untergeordnete Spans, werden „Child Span“ genannt. Der „Root Span“ beschreibt somit die Ende-zu-Ende-Latenz des gesamten Traces.

2.3.3 Metriken

Unter Metriken versteht man in der Informatik eine Funktion, die Informationen zu Prozessen, die im System ausgeführt werden, sammelt und diese dann als Maßzahlen oder Histogramme abbildet. Metriken kann man in fünf Kategorien einteilen:

- Hostbasierten Metriken
- Anwendungsmetriken
- Netzwerk- und Konnektivitätsmetriken
- Serverpool-Metriken
- Externe Abhängigkeitsmetriken

Die einfachsten Metriken sind die Hostmetriken, die bereits vom Betriebssystem erstellt werden, Daten wie zum Beispiel Speicherplatz und CPU-Auslastung oder RAM-Nutzung. Bei den Anwendungsmetriken handelt es sich um die Darstellung der Verfügbarkeit der Anwendungen, Fehler- und Erfolgsraten bei Anfragen sowie Leistung und

Latenz der Antworten von Verarbeitungsprozessen. Auf Grundlage dieser Indikatoren kann festgestellt werden, ob die Anwendung ordnungsgemäß und effizient funktioniert. Mit Hilfe der Netzwerk- und Konnektivitätsmetriken werden die Netzverfügbarkeit, die Paketverluste, Latenzen und Bandbreitennutzung aufgezeichnet. Durch diese Überwachung der Netzwerkschicht können Latenzen von internen und externen Anwendungen optimiert werden. Durch Serverpool-Metriken wird die Skalierbarkeit und Auslastung der Server gemessen. Dies dient dazu, um das System besser auf unterschiedliche Lastsituationen einstellen zu können. Unter Externe Abhängigkeitsmetriken versteht man die Überwachung von externen APIs, um zum Beispiel deren Verfügbarkeit und Status zu ermitteln. [25, 26]

2.3.4 Monitoring

Während Metriken wichtige Daten in Systemen und Anwendungen auswertet und in Maßzahlen fasst, werden die Daten vom Monitor aufbereitet und grafisch dargestellt. Monitor-Tools sollen in der Lage sein, einem Überblick über den Zustand des System mit den erzeugten Daten zu geben, verschiedene Datenquellen in einen Kontext zu bringen und bei Überschreitung wichtiger Metriken einen Hinweis auf die Auswirkungen des Fehlers zu geben und Warnhinweise an die jeweiligen zuständigen Abteilungen zu schicken. [19]
Im Kapitel 3.5 werden zwei Monitor-Systeme näher betrachtet.

2.4 Nutzungsdaten

Nutzungsdaten beinhalten im Gegensatz zu Telemetriedaten in der Regel personenbezogene Daten. Dabei handelt es sich um Daten, die ein Anbieter benötigt, um die Nutzung seines Telemediendienstes zu ermöglichen und abzurechnen. Die Verwendung von Nutzungsdaten werden in §15 Telemediengesetz geregelt. Nutzungsdaten entstehen bei jeder Interaktion zwischen dem Nutzer und dem Dienst. Dabei wird protokolliert, welche Web-Seiten besucht, welche Dienste in Anspruch genommen und mit welchem Browser und von welcher IP-Adresse die Seite besucht wurde. Da es sich bei IP-Adressen um personenbezogene Daten handelt, dürfen diese nicht gespeichert werden. Ohne ausdrückliche Einverständnis des Benutzers sind diese Daten zu löschen

oder zu anonymisieren. Nach der Anonymisierung dürfen die Nutzungsdaten als Telemetriedaten verwendet werden. [27, 28]

2.5 Zusammenfassung

Die in diesem Kapitel erläuterten Begriffe sind elementar für das weitere Verständnis von Telemetrie und bilden die inhaltliche Grundlage für die folgenden Kapitel.
Insbesondere die Begriffe Tracing, Span, Logging, Metriken und Monitoring sind in der heutigen Telemetrie eng miteinander verknüpft. Metriken können verwendet werden, um Traces mit schlechtem Verhalten zu identifizieren. Weiterhin können mit diesen Traces verknüpfte Logging-Protokolle dazu beitragen, die Hauptursache für dieses Verhalten zu ermitteln. Anschließend können basierend auf dieser Ermittlung neue Metriken erstellt werden, die durch einen Monitor ausgewertet werden. [26]

3 Analyse

3.1 Telemetrie im Alltag

Die Telemetrie nehmen wir im Alltag in der Regel selbst nicht wahr. Bei der Verwendung des Smartphones oder des Computers werden immer Telemetriedaten im Hintergrund übertragen. Es gibt Hersteller, die mit diesem Thema offen umgehen. Sie ermöglichen dem Benutzer die aufgezeichneten Telemetriedaten einzusehen (vgl. 3.1.1). Aber die große Mehrheit der Hersteller hält sich sehr bedeckt zum Thema Telemetrie, geben keinerlei Information zum Einsatz von Telemetriedaten in ihren Produkten und bieten auch keine Möglichkeit diese einzusehen.
In diesem Kapitel werden zwei große Softwareprodukte, die täglich von Millionen von Menschen verwendet werden, hinsichtlich ihres Umgangs mit Telemetriedaten analysiert.

3.1.1 Windows 10

Das erste untersuchte Softwareprodukt ist Windows 10. Windows 10 sendet Telemetriedaten an Microsoft, um zum Beispiel Probleme mit Updates zu ermitteln und diese für Nutzer bestimmter Hardware zu sperren oder ganz allgemein Probleme rund um Windows zu ermitteln. Telemetriedaten werden unter Windows als Diagnosedaten bezeichnet. Ziel von Microsoft ist es, mit diesen Diagnosedaten ihre Software zu verbessern.
Seit Windows10-Version 1809, die am 27.03.2020 erschienen ist, ist es möglich für jeden Benutzer von Windows die erzeugten Telemetriedaten abzurufen. „Um diese Funktion zu aktivieren, muss man in die Systemeinstellungen von Windows gehen und in das Suchfeld „Diagnose und Feedback“ eingeben. In diesem Abschnitt findet man dann den Reiter „Diagnosedaten anzeigen“, wie in Abbildung 3.1 zu sehen.“

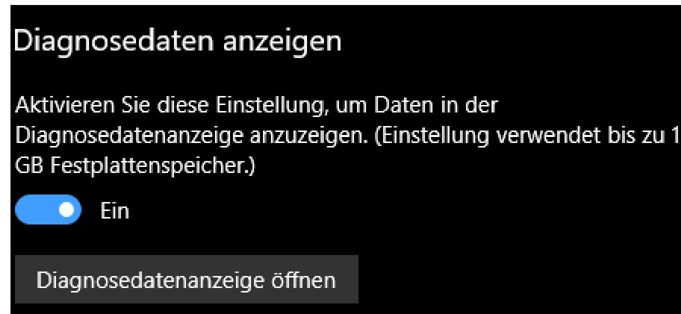

Abbildung 3.1: Aktivierung von Diagnosdaten in Microsoft

Dieser Schalter muss aktiviert werden, um die Microsoft-App Diagnosedatenanzeige nutzen zu können. Ab diesem Zeitpunkt speichert die App alle anfallenden Telemetriedaten von Windows. Die Telemetriedaten werden als JavaScript Object Notation (JSON)-File[29] abgelegt. Das Listing 3.1 zeigt einen Beispiel-Diagnose-Datensatz. Der Aufbau der Diagnosedaten ist unter der Webseite[1] einsehbar. Auf dieser findet man alle Informationen über die Kategorien und ihren einzelnen Datentypen.
In der App Diagnosedatenanzeige findet man auf der rechten Seite drei Kategorien: Diagnosedaten, Problemberichte und eine Statistik zu den erzeugten Daten. Im nachfolgenden Beispiel wird näher auf eine Diagnosedatei eingegangen, die unter der Kategorie Diagnosedaten zu finden ist. Dort sind Telemetriedaten-Daten über die verwendete Hardware, Peripheriegeräte und die installierte Software zu finden.
Zu Beginn jeder Diagnosedatei ist in der Zeile 1 spezifiziert, für welche Kategorie die Daten erhoben und übermittelt wurden. In diesem Fall für die Kategorie „Softwareeinrichtung und -bestand“, das der Hauptkategorie „Microsoft Store-Ereignisse“ zugeordnet werden kann.

Listing 3.1: Beispiel einer Diagnosedatei von Windows 10 [2]

```
1 Microsoft.Windows.StoreAgent.Telemetry.EndScanForUpdates
2 {   "ver": "4.0",
3     "name": "Microsoft.Windows.StoreAgent.Telemetry.EndScanForUpdates",
4     "time": "2020-05-30T16:10:43.7009497Z",
5     "iKey": "o:0a89d516ae714e01ae89c96d185e9ae3",
6     "ext": {
```

[1] https://docs.microsoft.com/de-de/windows/privacy/basic-level-windows-diagnostic-events-and-fields-1809

```
7         "utc": {
8             "eventFlags": 514,
9             "pgName": "WINCORE",
10            "flags": 905970224,
11            "epoch": "15703954",
12            "seq": 338
13        },
14        "metadata": {
15            "f": {
16                "HResult": 2
17            }
18        },
19        "mscv": {
20            "cV": "Cle5YEjzOEytPX+P.1.0.1"
21        },
22        "os": {
23            "bootId": 157,
24            "name": "Windows",
25            "ver": "10.0.18363.836.amd64fre.19h1_release.190318-1202"
26        },
27        "app": {
28            "id": "W:0000f519feec486de87ed73cb92d3cac802400000000!000075
                  c5a97f521f760e32a4a9639a653eed862e9c61!svchost.exe",
29            "ver": "1997/01/10:22:26:24!10A1C!svchost.exe",
30            "asId": 212
31        },
32        "device": {
33            "localId": "s:8CBC2DB0-22C3-455E-B448-C47A9CDC1870",
34            "deviceClass": "Windows.Desktop"
35        },
36        "protocol": {
37            "devMake": "LENOVO",
38            "devModel": "20B30079GE"
39        },
40        "user": {
41            "localId": "w:2A3B4819-AF32-9C6C-48A2-678662E8B2A4"
42        },
43        "xbl": {
44            "sbx": "RETAIL",
45            "did": "F900ABCB7CE7D794"
46        },
47        "loc": {
```

```
48                "tz": "+02:00"
49            }},
50        "data": {
51            "IsOnline": 2,
52            "IsApplicability": 0,
53            "IsInteractive": 0,
54            "ClientAppId": "Update;ScanForUpdates",
55            "HResult": 0
56        }}
```

Der Grundaufbau der einzelnen Diagnosedateien ist nahe zu gleich. Zunächst werden die allgemeinen Informationen aufgelistet wie zum Beispiel mit welcher Version die Diagnosedaten aufgenommen wurden. Danach werden weitere Informationen über die Windows-Version und den Rechner-Hersteller (37) abgelegt. Die Datenpunkte, die je nach Ereignistyp variieren, befinden sich am Ende der JSON-Datei unter dem Punkt „data“ (50). [30]
Bei diesem Beispiel handelt es sich um ein Ereignis, das zur Überprüfung von Produktupdates gesendet wird, um festzustellen, ob Pakete bereits vorhanden sind oder installiert werden müssen. Es wird verwendet, um Windows auf dem neuesten Stand zu halten. [30]

1. Zeile 41 beschreibt eine lokal definierte eindeutige ID für das Gerät. Dies ist der nicht lesbare Name des Geräts.
2. Unter dem Punkt „Data“ in Zeile 50 befinden sich spezifische Datentypen je nach Ereignis. Zum Beispiel wird unter dem Datenpunkt „IsOnline“ aufgelistet, welche App diesen Vorgang gestartet hat (Zeile 51) und der Datentyp „ISApplication“ gibt an, ob eine Anforderung überprüft werden soll oder ob anwendbare Pakete vorhanden sind, die installiert werden müssen. [30]

Da viele Inhalte der einzelnen Datentypen mit internen Bezeichnungen gefüllt werden, die nicht unmittelbar interpretierbar sind, ist es schwierig herauszufinden, von welchen Anwendungen die Diagnosedaten stammen. Auf Grund der AppID könnten in diesem Beispiel die Daten von der App „Xbox Game Bar“ stammen.

[2] Dieses Beispiel wurde mit Windows 10 Pro mit der Version 1909 erstellt.

3.1.2 Microsoft Office

Das zweite Produkt, das analysiert wurde, ist Microsoft Office, das ebenso wie Windows 10 auf Millionen von Rechnern installiert ist.
Das Ministerium „Ministry of Justice and Security“ aus der Niederlande lies die Software-Anwendung Microsoft Office App untersuchen, ob diese gegen die aktuellen Datenschutz-Richtlinien von Telemetriedaten verstoßen. Dabei wurden fünf Verstöße, die mit einem hohen Risiko eingestuft wurden, festgestellt. Zu einem hat Microsoft direkt identifizierbare personenbezogene Daten wie Benutzername und E-Mail-Adresse sowie die Zeiten gesammelte, zu denen einzelne Mitarbeiter Aktionen in den Anwendungen ausgeführt haben. Wenn Anwender zudem die Cloud-Speicher- und E-Mail-Dienste SharePoint Online, OneDrive for Business und Exchange Online verwenden, sammelt Microsoft Inhaltsdaten zu Titeln und Pfadnamen. Außerdem wurden auch ohne Wissen der Anwender Daten an das US-Marktforschungsunternehmen Braze geschickt. Dieses Versenden der Daten konnte auch nicht vom Anwender oder Administratoren abgeschaltet werden. Das Ergebnis dieser Untersuchung war, dass der Einsatz von Microsoft Office nicht DSGVO konform war. Diese Probleme wurden laut Microsoft behoben und es gibt mittlerweile die Möglichkeit das Versenden von Daten zu unterbinden. [31]

3.2 Evaluierungskriterien

In den letzten Jahren entstanden unterschiedlichste Tools für Distributed Tracing. Diese Tools stellen eigene Bibliotheken bereit, die das Erzeugen und Nachverfolgen von Traces möglich machen. Diese Bibliotheken sind in der Regel an ein festes Monitoring-System gekoppelt.
Aus diesem Grund wurde in dieser Arbeit das Open-Source-Projekt OpenTelemetry ausgewählt, da es den aktuellen Stand der Technik für das Erzeugen und Weiterleiten von Telemetriedaten darstellt. Es gibt den Benutzer die Möglichkeit plattformunabhängig verschiedene Datenströme zu kapseln und weiterzuverarbeiten.
Das Open-Source-Projekt OpenTelemetry soll anhand von sechs Kriterien bewertet werden, wie es die Observability in der Entwicklung eines System unterstützen und verbessern kann und wie es sich von anderen Tools abhebt.

1. Interoperabilität

   Das Tool soll die Fähigkeit besitzen Informationen effizient auszutauschen, ohne dass dazu gesonderte Absprachen zwischen den Systemen notwendig sind.

2. Plattformunabhängkeit

   Da bei verteilten Systemen oft mehrere Programmiersprachen zum Einsatz kommen können, sollte das Tool am Besten von Programmiersprachen unabhängig sein

3. Standardisierung

   Das Tool sollte vorhandene Standards unterstützen und Wegbereiter für neue Standards sein

4. Erweiterbarkeit

   Das Tool sollte einfach erweiterbar sein, um es auf spezifische Bedürfnisse anpassen zu können

5. Dokumentation

   Es sollte eine Dokumentation zur Beschreibung und Anwendung des Tools vorhanden sein

6. Mehrwert

   Das Tool soll einen entschiedenen Vorteil gegenüber bereits existierenden Tools bieten

3.3 Das Open-Source-Projekt OpenTelemetry

OpenTelemetry ist eine Open-Source-Projekt, das im Mai 2019 offiziell vorgestellt wurde. Dieses Projekt gehört der Cloud Native Computing Foundation an und wurde von dem Enterprise-Cloud-Spezialisten Dynatrace gemeinsam mit Google und Microsoft ins Leben gerufen. Diese Foundation hat sich zum Ziel gesetzt, frei verfügbare Open-Source-Projekte für Cloud Native Computing zu unterstützten und die Zusammenarbeit zwischen Herstellern, Entwicklern und Anwendern in diesem Bereich zu fördern.

Die Entwurfsziele von OpenTelemetry sind neben der Bereitstellung einer Standard-API und eines Software Development Kit (SDK) für die Instrumentierung, die Bereitstellung einer hochwertigen automatischen Instrumentierung für gängige Frameworks und Bibliotheken, was mit Hilfe von Wrapper und Agenten umgesetzt werden soll.

> *Das ultimative Ziel von OpenTelemetry ist es, die Standardmethode zu werden, mit der Entwickler und Betreiber die Leistungsdaten ihrer Dienste erfassen.*[14]

Dieses Zitat stammt vom Google-Produktmanager Morgen McLean aus einem Interview. Dies bedeutet, das Hauptziel von OpenTelemetry besteht darin, eine Sammlung von sprachspezifischen APIs, Bibliotheken, Agenten und Kollektoren bereitzustellen, mit denen verteilte Traces, Metriken und verwandte Metadaten aus jeglicher Art von Anwendungen leicht und unkompliziert erfasst werden können. Dabei steht im Vordergrund eine verbesserte Transparenz für Transaktionen in verteilten Systemen anzubieten. Das OpenTelemtry-Team arbeitet daran, eine Standardisierung für das Erzeugen und Sammeln auf der Transaktionsebene zu etablieren. Die Telemetriedaten sollen in geeigneter Form für Monitoring-Systeme bereitgestellt werden, die diese Daten dann auswerten und darstellen. [32]
Das OpenTelemetry Projekt setzt auf den beiden vorangegangen Open-Source Projekten OpenCensus und OpenTracing auf. OpenTracing und OpenCensus haben bei der Bereitstellung von Tracing-Software eine Vorreiterrolle gespielt. Auch wenn jedes Projekt unterschiedliche architektonische Entscheidungen traf, war das größte Problem für die beiden Projekten die Tatsache, dass es zwei davon gab. Ein weiteres Problem ergab sich daraus, dass die beiden Projekte nicht zusammen arbeiteten und nicht nach gegenseitiger Kompatibilität strebten.
Im nächsten Abschnitt werden die Projekte OpenCensus und OpenTracing kurz vorgestellt.

3.3.1 OpenCensus

Das Projekt OpenCensus wurde von Google ins Leben gerufen, um automatisch Traces und Metriken von Diensten zu erfassen. Dabei spielt es keine Rolle, welche Software-Architektur zugrunde liegt. Dies kann ein komplexer Microservice sein, wie auch nur eine einfache monolithische Anwendung.

OpenCensus stellt für viele Programmiersprachen wie C++, Java, PHP oder Python Bibliotheken zur Verfügung. Die Kernfunktion von OpenCensus soll den Entwicklern die Möglichkeit bieten, einfach Traces und Metriken zu erzeugen und mit Hilfe eines beliebigen Analysetools diese untersuchen zu können. [2, 33] OpenCensus gliedert sich in vier Teile: OpenCensus Library, OpenCensus Agent, OpenCensus Exporter und OpenCensus Collector (siehe auch Abbildung 3.2). In den einzelnen Services können

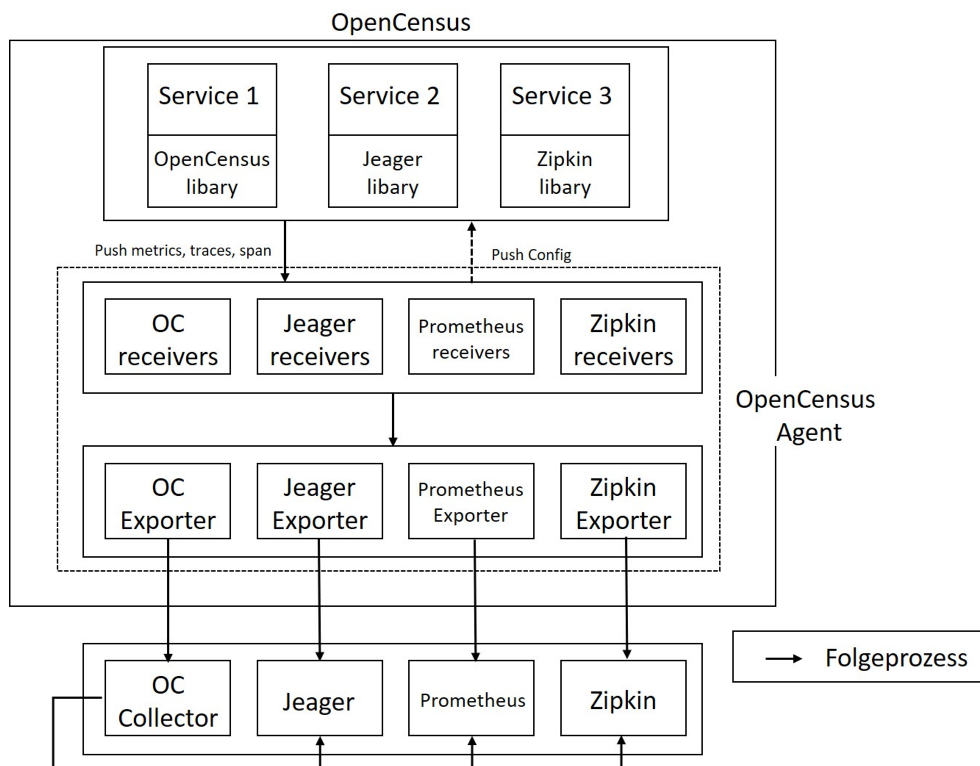

Abbildung 3.2: Aufbau von OpenCensus

unterschiedliche Monitoring und Tracing Bibliotheken wie z.B. von Jaeger, Prometheus, Zepkin oder OpenCensus eigene Bibliotheken integriert werden. Die Traces und Metriken werden mit Hilfe der OpenCensus Library in den einzelnen Services erzeugt. Bei Programmiersprachen, die bereits einen Kontext implementiert haben, wird dieser verwendet. Dies gilt zum Beispiel für die Programmiersprachen Go und C#. Bei einem Kontext in der Software-Telemetrie handelt es sich um einen Satz global eindeutiger Identifikatoren für Telemetriedaten.

Für die anderen Programmiersprachen stellt die OpenCensus Library die passenden API für Tracing bereit, um die explizite oder implizite Weitergabe von Spans innerhalb eines Prozesses zu gewährleisten. [2, 33, 34]
Der OpenCensus Agent kann als Daemon in einer VM, im Container als Side Car oder als Agent ausgeführt werden. Der Agent kann unabhängig von der Bibliothek bereitgestellt werden. Sobald der Agent ausgeführt wird, sollte er in der Lage sein, Bereiche, Statistiken und Metriken aus der Bibliothek abzurufen und in andere Backends zu exportieren.
Der OpenCensus Exporter sendet Traces und Metriken an jedes Backend, das diese Telemetriedaten verarbeiten kann. OpenCensus bietet viele Exporter für Open-Source-Backends für den sofortigen Einsatz an. [2, 33, 34]
Der OpenCensus Collector ist eine Komponente, die Ablaufsverfolgungsbereiche und Metriken aus der Collector Ebene empfängt und als Schnittstelle zu den Anwendungskomponenten eines Benutzers fungiert. Der Collector stellt einen zentralen Austrittspunkt für den Export von Traces und Metriken in ein oder mehrere Tracing- und/oder Metrik-Backends dar und bietet Pufferung und Wiederholungsversuche sowie erweiterte Funktionen für Aggregation, Filterung, Annotation und Stichproben.[2, 34]

3.3.2 OpenTracing

Die Herangehensweise von OpenTracing ist im Vergleich zu OpenCensus grundverschieden. Während OpenCensus ein Rundum-Sorglos Open-Source SDK sein will, stellt OpenTracing nur eine API für Tracing bereit. [35]
OpenTracing wurde 2016 gestartet mit dem Ziel den mangelnden Zustand der Tracing-Instrumentierung zu beheben. Es unterstützt 9 Programmiersprachen: Go, JavaScript, Java, Python, Ruby, PHP, Objective-C, C++, C# .
Das Ziel von OpenTracing ist, ein Framework und herstellerunabhängige APIs bereitzustellen, die es erlauben, Traces zu protokollieren. Dabei soll sichergestellt werden, dass die darunterliegende Implementierung leicht ausgetauscht werden kann. Dies bedeutet, dass der Entwickler die Konfiguration des Tracers einfach ändern kann, wenn ein Entwickler ein anderes verteiltes Tracing-System testen möchte, anstatt den gesamten Instrumentierungsprozess für das neue verteilte Tracing-System zu wiederholen. Um dieses Ziel zu erreichen wurde eine semantische Spezifikation, die über alle Programmiersprachen hinweg portierbar ist, entworfen. Somit wird ein Schnittstellenpaket für

Entwickler bereitgestellt. [2, 35]
Der Grundaufbau von Traces und Spans ist sehr ähnlich zum OpenCensus-Projekt. Jedoch bietet OpenTracing den Benutzern nur ein Tracing-Framework an. Es wird keine Metrik-API bereitgestellt, um die Aufzeichnung von Zählern, Messgeräten oder anderen gängigen Metriken zu ermöglichen.

3.4 Architektur von OpenTelemetry

Das Projekt OpenTelemetry hat die beiden Projekte OpenCensus und OpenTracing zusammengeführt. Dabei wurde die Grundidee von OpenCensus übernommen, eine SDK für die bekanntesten Programmiersprachen bereitzustellen, um somit eine einheitliche Plattform für Distributed Tracing und Metriken zu schaffen.
OpenTelemetry bildet ein Framework für viele Programmiersprachen und Monitoring-Systeme, es ist flexibel und erweiterbar (4.1.4), aber gleichzeitig auch komplex.
Wenn in einer Software bereits OpenTracing oder OpenCensus verwendet wird, bietet OpenTelemetry eine Abwärtskompatibilität für beide Projekte. [2, 36]
Grundsätzlich wird im OpenTelemetry-Projekt zwischen API und SDK unterschieden, damit zur Laufzeit verschiedene SDKs konfiguriert werden können [37]. Eine API ist eine Schnittstelle, die für die Interaktion zwischen verschiedenen Programmen oder Plattformen entwickelt wurde. Dabei wird die Kommunikation durch eine Reihe von Regeln definiert, die Datenstrukturen, Protokolle usw. festlegen. Die SDK beinhaltet eine oder mehrere APIs sowie Codebeispiele, Bibliotheken und weitere Komponenten. Alle SDKs enthalten APIs, aber nicht alle APIs sind Teil des SDKs. [38]

3.4.1 OpenTelemetry API

Die Architektur von OpenTelemetry gliedert sich, wie in Abbildung 3.3 grafisch dargestellt, in drei Hauptkomponenten:

1. OpenTelemetry API
    - Tacer API
    - Context API

   - Metric API
   - Logs API
2. OpenTelemetry SDK
   - Context Layer
   - Tracing Path
   - Metric Path
3. OpenTelemetry Collector

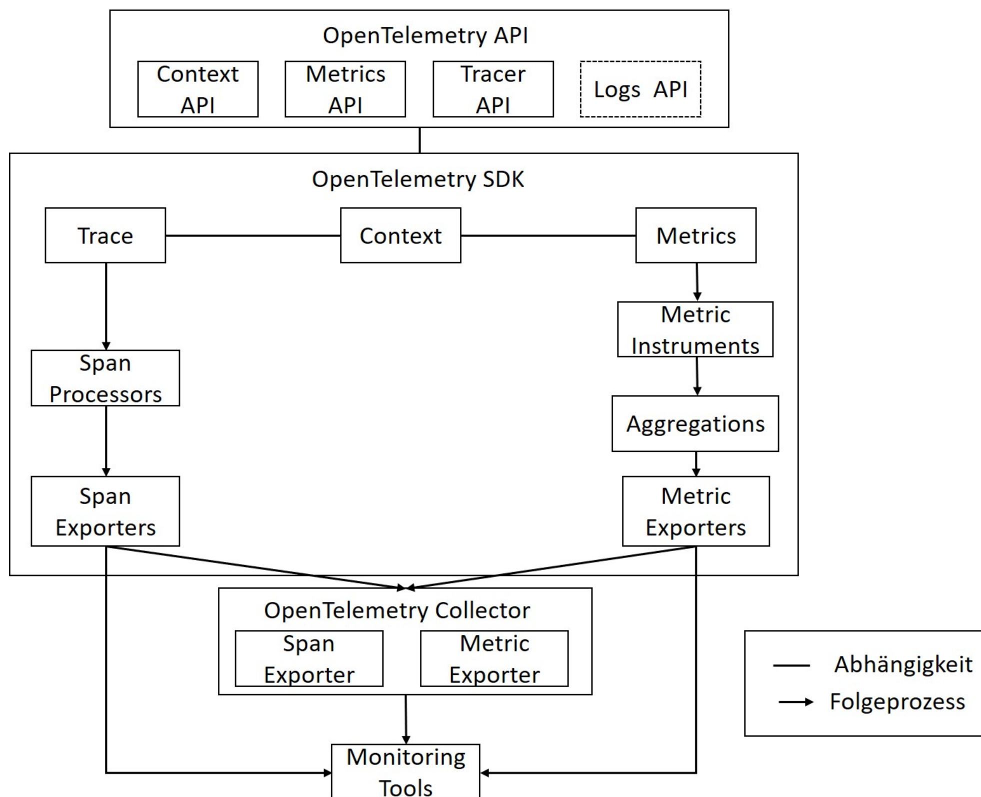

Abbildung 3.3: Aufbau von OpenTelemetry

Die API besteht aus vier Komponenten:

1. Log
2. Tracer
3. Metric
4. Context

Das Datenmodell für die Log API ist bereits definiert, jedoch noch nicht implementiert. Ziel ist, dass das entwickelte Log-Format auf bereits bestehende Log-Formate semantisch korrekt gemappt wird und diese von einem Format zum anderen einfach und schnell umgewandelt werden können, ohne Information zu verlieren. Hierfür wurden die existierenden Log-Formate wie Log4j, das für Java ein Quasi-Standard darstellt, oder Windows Event Log ausgewählt und die Zuordnung zwischen den Datenmodellen in einer Mapping-Tabelle vorgenommen. [2, 36]
Die OpenTelemetry-API besitzt eine Reihe von „Semantic Conventions", die Richtlinien und Regeln zum Benennen von Bereichen, Attributen oder zum Zuordnen von Fehlern zu Bereichen festhalten. Durch die Standardisierung der API-Spezifikation stellt das OpenTelemetry-Projekt sicher, dass alle Instrumente, unabhängig von Autor oder Sprache, dieselben semantischen Informationen enthalten. Diese können jeweils in den Unterordnern der APIs unter dem Link [3] gefunden werden.
Die Tracer API besteht aus drei wichtigen Klassen:

- Tracer-Provider ist der Einstiegspunkt der API. Es bietet Zugriff auf Tracer.
- Tracer ist die Klasse, die für die Erstellung von Spans verantwortlich ist.
- Span ist die API zum Verfolgen einer Operation.

Der Tracer bietet Methoden zum Erstellen und Aktivieren neuer Span-Objekte sowie die Möglichkeit, den aktiven Span im Prozesskontext zu verfolgen und zu verwalten.
Jeder Span muss einen SpanContext enthalten, bei dem es sich um eine vorgegebene Datenstruktur handelt und der eindeutig den Span identifiziert. Die OpenTelemetry

[3] https://github.com/open-telemetry/opentelemetry-specification/tree/master/specification

Span-Context-Darstellung entspricht der W3C Trace-Context-Spezifikation. Es enthält zwei eindeutige IDs, eine TraceId und eine SpanId, sowie eine Reihe allgemeiner TraceFlags und systemspezifische TraceState-Werte [39].

TraceId

: Eine gültige Trace-ID ist ein 16-Byte-Array mit mindestens einem Byte ungleich Null.

SpanId

: Eine gültige Span-ID ist ein 8-Byte-Array mit mindestens einem Byte ungleich Null.

TraceFlags

: Enthalten Details zum Trace. Sind in allen Traces vorhanden.

TraceState

: Enthält systemspezifische Konfigurationsdaten, die als Liste von Schlüssel-Wert-Paaren dargestellt werden. Mit TraceState können mehrere Ablaufverfolgungssysteme an derselben Ablaufverfolgung teilnehmen.

IsValid

: Ein boolesches Flag, das true zurückgibt, wenn der SpanContext eine TraceID ungleich Null und eine SpanID ungleich Null hat.

IsRemote

: Ein Boolescher Wert, der wahr ist, wenn der SpanContext von einem entfernten übergeordneten Element weitergegeben wurde. Beim Extrahieren eines SpanContext über die Propagators-API MUSS das IsRemote-Flag auf true gesetzt werden, während der SpanContext aller untergeordneten Spans auf false gesetzt sein muss.

Die OpenTelemetry Metric-API unterstützt die Erfassung einer Messung bei Ausführung einer Anwendung zur Laufzeit. Die Metric-API wurde explizit für die Verarbeitung von Rohmessungen entwickelt. Im Allgemeinen mit der Absicht, kontinuierlich Messungen effizient zu berechnen und darzustellen. Jeder Messung ist ein bestimmter Typ eines Instrumentes zugeordnet, mit dem die Messung durchgeführt wurde, wodurch die Messung ihre semantischen Eigenschaften erhält. Die Metrik-API bietet

grundsätzlich Zugriff auf zwei Arten von Metrik-Instrumenten, die jeweils in 3 Unterfunktionen aufgeteilt sind. Diese werden durch Aufrufe einer Meter-API erstellt und definieren den Einstiegspunkt in die SDK. Mit „Countern“ werden Werte gezählt. Mit „Observer“ können Werte zu bestimmten Zeitpunkten gemessen werden. Observer werden zum Beispiel verwendet, um Werte zu überwachen, die nicht im Kontext eines Bereichs stattfinden, zum Beispiel die aktuelle CPU-Auslastung oder die Gesamtzahl der auf einer Festplatte freien Bytes. Bei den Instrumenten unterscheidet man zwischen zwei Messungsarten: synchron und asynchron. [40]
Synchrone Instrumente werden innerhalb einer Anforderung aufgerufen, das bedeutet, dass ihnen ein verteilter Kontext zugeordnet ist. Es gibt zwei synchrone additive Instrumente, Counter und UpDownCounter und ein nicht additives Instrument den ValueRecorder. [40]
Asynchrone Instrumente werden dagegen durch einen Rückruf gemeldet und nur einmal pro Erfassungsintervall aufgerufen. Sie werden keinem Kontext zugeordnet. Sie dürfen nur einen Wert pro eindeutigem Metrik-Satz im Erfassungsintervall angeben. Wenn die Anwendung in einem einzigen Rückruf mehrere Werte für denselben Metrik feststellt, wird der letzte Wert übernommen. Es gibt zwei asynchrone additive Instrumente, SumObserver und UpDownObserver und einen asynchrone nichtadditive Instrument den ValueObserver. [40]

Counter

Der Zähler dient dazu einen bestimmten Wert zu zählen mit Hilfe der Methode add(Inkrement). Dabei darf das Inkrement nur positive Werte annehmen. Es wird zum Beispiel die Anzahl der empfangen Bytes oder die Anzahl abgeschlossener Anforderungen erfasst.

UpDownCounter

Dieses Instrument hat die gleiche Funktion wie der Counter, er darf auch negative Werte verarbeiten. Dies ist nützlich bei der Überwachung von Warteschlangen

ValueRecorder

Dient zum Aufzeichnung von nicht additiven Werten, die nicht zu einer Summe addiert werden soll oder die einzelnen Wert von Bedeutung sind.

SumObserver

Dieses Instrument dient zur Erfassung einer positiven monotonen Zählung mit

der Funktion Observe. Diese Methode soll dort eingesetzt werden, wenn die Berechnung einer Messung bei jedem Systemaufruf zu teuer würde. Die Messung wird regelmäßig, aber nicht dauernd wie bei Counter ausgeführt.

UpDownSumObserver

Sie hat die gleiche Funktion wie SumObserver, kann aber sowohl negative als auch positive Werte verarbeiten, dient zur Erfassung einer nicht monotonen Zählung.

ValueObserver

Ist sehr ähnlich zu ValueRecorder und dient zur Erfassung nicht additiver Messungen, die mit der Methode Observe (Wert) verwendet wird. Diese Instrumente sind besonders nützlich, um Messungen zu erfassen, deren Berechnung teuer ist, da das SDK die Kontrolle darüber hat, wie oft sie ausgewertet werden.

Die Context-API stellt Informationen über den verwendeten Kontext bereit, von welcher Bibliothek die Traces und Metriken bereitgestellt werden. Es kann zum Beispiel bei Traces zwischen dem W3C-Trace-Context oder der Zipkin B3-Header ausgewählt werden. Der Kontext wird in zwei Typen unterteilt, den Span-Context und den Correlation Context. [2, 36]
Spans sind als eigenständige Daten nicht sehr aussagekräftig. Es wird eine Möglichkeit gebraucht, bestimmte Details über die Spans an andere Dienste oder andere Teile des Prozesses zu übermitteln. Der Mechanismus, mit dem diese Details an andere Dienste übermittelt werden, ist allgemein als Kontextpropagierung bekannt.
Der Correlation Context API ist eine Funktion von OpenTelemetry, die es ermöglicht, Indexwerte während einer Anfrage zu propagieren, um Metadaten, die in einem Dienst auftreten, mit anderen Daten zu korrelieren. Durch diese Verknüpfung kann eine kausale Verbindung zwischen den den Daten hergestellt werden. Der Correlation Context dient dazu Telemetriedaten zu annotieren, dabei werden Kontext und Informationen zu Metriken, Spans und Logs hinzugefügt. Die API für Korrelationen basiert auf der W3C-Correlation-Context-Spezifikation. [37]

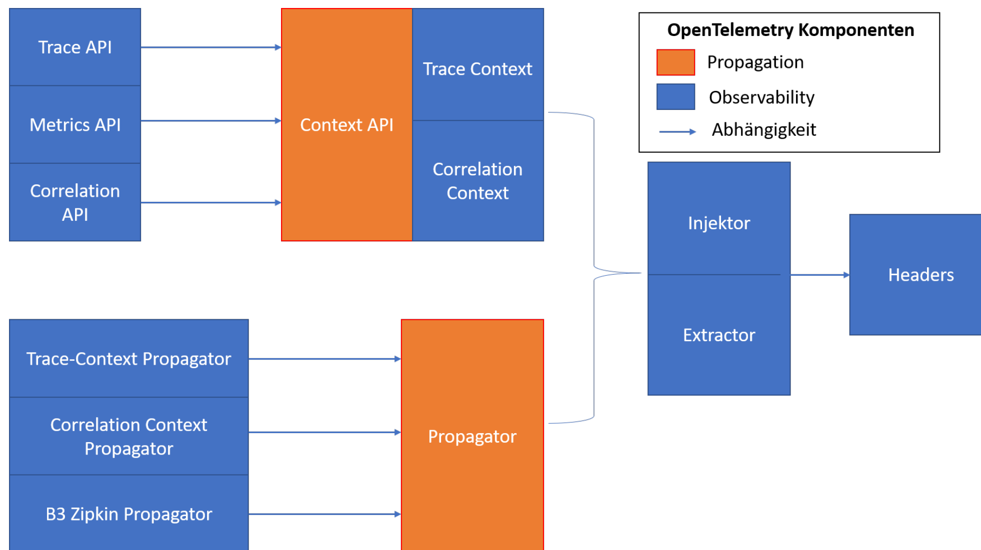

Abbildung 3.4: Funktionsweise von Context and Propagation

Um Korrelationen zu finden, müssen die Komponenten in einem verteilten System in der Lage sein, Metadaten, die als Kontext bezeichnet werden, zu sammeln, zu speichern und zu übertragen.
Ein Kontext verfügt oft über Informationen, die die aktuellen Spans und Traces identifizieren und beliebige Korrelationen als Schlüssel-Werte-Paare enthalten können. Die Propagierung ist das Mittel, mit dem Kontext gebündelt wird. Dieser Kontext kann in und zwischen Diensten übertragen werden.
Zusammen stellen Kontext und Propagierung den Motor hinter Distributed Tracing dar. Die Abbildung 3.4 gibt einen Überblick über die Architektur der Kontextpropagierung.
Der häufigste Anwendungsfall der Propagierung ist das Senden eines Trace-Kontextes von einem Client, der eine Anforderung zu einem Server sendet. Das Mittel, mit dem Kontext gebündelt und zwischen Diensten übertragen wird, kann über HTTP- oder Remote Procedure Call (RPC)-Headers erfolgen.
Die Propagierung ist der Mechanismus, durch den ein Trace zu einem verteilten Trace wird. Dadurch wird die Bewegung des Kontexts zwischen Diensten und Prozessen erleichtert. Der Kontext wird in eine Anforderung injiziert und von einem empfangenden Dienst extrahiert, um untergeordnete neue Spans zu erstellen. Dieser Dienst kann zu-

sätzliche Anfragen stellen und Kontext einfügen, der dann an andere Dienste gesendet wird. Es gibt mehrere Protokolle, für die Kontext-Propagierung, die OpenTelemetry unterstützt. [36]
Durch die vorgestellten API ist es nun möglich, den in Abbildung 3.5 gezeigten Ablauf, darzustellen.

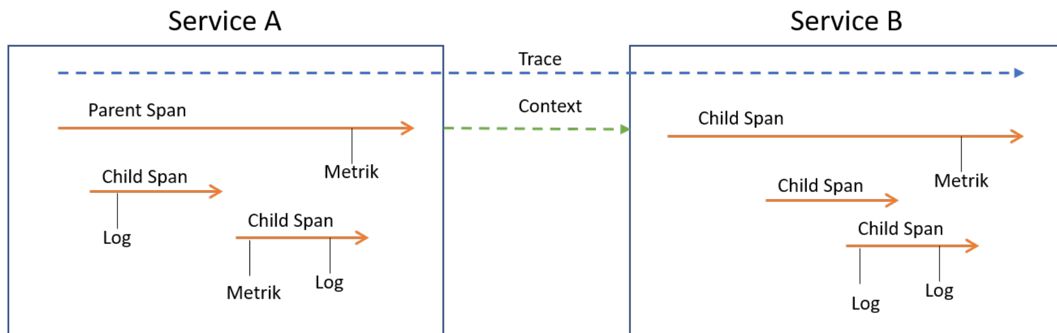

Abbildung 3.5: Zusammenspiel der einzelnen APIs

Mit Hilfe der Tracing API ist es möglich Spans zu erstellen. Durch den Correlation Context ist es möglich Metriken und Logs an den Trace anzuhängen. Somit wäre man in der Lage die Abhängigkeit eines bestimmten Fehlers mit der Betriebssystem- oder Browser-Version zu verknüpfen.

3.4.2 OpenTelemetry SDK

Das OpenTelemetry SDK ist die Implementierung der OpenTelemetry API. Das SDK besteht aus drei Teilen: dem Tracer-Path, dem Metric-Path und einer gemeinsamen Kontextebene, die alles miteinander verbindet.
Der Trace-Ablauf besteht aus dem Span Processor und und dem Span Exporter.
Der Span-Prozessor ist eine Schnittstelle, die Aufrufe für Span-Start- und -Ende-Methode ermöglicht. Bei der Konfiguration des SDK können ein oder mehrere Span Prozessoren einer Tracer-Pipeline zugeordnet werden. Span-Prozessoren sind für die Verarbeitung und Konvertierung von Spans verantwortlich. Diese werden in einem bestimmten konfigurierbaren Intervall an die Exporter weitergegeben. [2, 36]
Am Ende der Tracer-Pipeline befindet sich der Span-Exporter. Dieser konvertiert die

Bereiche von der OpenTelemetry-Darstellungen in die Darstellungsformat des Telemetrie-Backends und sendet diese Daten an dieses Backend.
Die Metrik-Path ist im Gegensatz zum Trace-Pfad wesentlich komplexer aufgebaut und unterschiedet sich je nach Programmiersprache. Über Metrik-Instrumnets können die verschiedenen Metriktypen initialisiert werden. Die Metrik-Sammlungen werden an den Metric Exporter weitergegeben. Auch hier können die Anbieter, wie bei den Spans, ihren eigenen Exporter bereitstellen, um die von den Metrik-Aggregatoren erzeugten Daten in die für das Telemetrie-Backend benötigten Datenstruktur zu konvertieren. OpenTelemetry unterstützt zwei Arten von Exporter: „Push"-basierte Exporter, bei denen der Exporter Daten in einem bestimmten Zeitintervall an das Backend sendet, und „Pull"-basierte Exporter, bei denen das Backend die Daten abfragt, wenn es sie benötigt. New Relic ist ein Beispiel für ein push-basiertes Backend, Prometheus ist ein pull-basiertes Backend. [2, 36]
Es existieren zwei Möglichkeiten die erzeugten Telemetriedaten an das gewünschte Backend zu senden. Die erste Möglichkeit ist, wie schon in Kapitel 3.3.1 beschrieben, die Telemetriedaten über einen geeigneten Programmiersprachen abhängigen Exporter in der SDK zu einem bestimmten Backend zu schicken. Dabei muss jeder Monitor-Hersteller für die passende Programmiersprache einen Exporter zur Verfügung stellen. Aus diesem Grund existiert nicht für alle Programmiersprachen ein passender Exporter für das gewünschte Monitoring-Tool, wie Tabelle 3.1 verdeutlicht.

Programmiersprache	Monitoring-Tool	Telemetriedaten
Java	Jaeger	Trace
	New Relic	Trace/Metrik
	Prometheus	Metrik
	Zipkin	Trace
Python	Azure Monitor	Trace/Metrik
	GoogleCloud	Trace/Metriks
	Jaeger	Trace
	Prometheus	Metrik
	Zipkin	Trace
GO	Google Cloud	Trace
	Jaeger	Trace
	New Relic	Trace/Metrik
	Zipkin	Trace
Node.JS	Azure Monitor	Trace
	Google Cloud	Trace
	Jaeger	Trace
	Zipkin	Trace

Tabelle 3.1: Verfügbare Exporter für Monitoring Tools in der jeweiligen Programmiersprache

Die zweite Möglichkeit ist, die Telemetriedaten an den OpenTelemetry Collector zu senden.
Der OpenTelemetry Collector ist ein herstellerunabhängiger Dienst, der Metriken und Spans aus verschiedenen Anwendungen empfangen, verarbeiten und exportieren kann. Dabei können die Daten sowohl von OpenTelemetry als auch von anderen Tracing-Bibliotheken wie Zipkin, Jaeger und OpenCensus stammen. Diese empfangenen Daten kann der Collector aggregieren und transformieren. Anschließend können diese Daten an einen oder mehrere Monitoring-Tools exportiert werden. Der Kollektor kann wie beim Projekt OpenCensus als eigenständiger Dienst oder als Dämon, der lokal neben der Anwendung läuft, betrieben werden. Durch die standardisierte Schnittstelle des Collectors, ist eine einfache Anbindung an die OpenTelemetry Plattform gegeben.

Es ist nicht mehr notwendig für jede Programmiersprache einen passenden Exporter zu schreiben, sondern nur einen Exporter bereit zustellen, der die Telemetriedaten vom Collector in das passende Dateiformat umwandeln kann. Dies eröffnet die Möglichkeit, eine Vielzahl von weiteren Monitoring Tools, wie zum Beispiel AWS X-Ray oder Light-Step, an die OpenTelemetry-Plattform anzubinden. [2, 36]

3.5 Monitoring-Tools

Anhand des OpenTelemetry Projekts wurde aufgezeigt, wie für Applikationen Telemetriedaten erzeugt werden können. In einem nächsten Schritt gilt es nun diese Daten auszuwerten und zu visualisieren. Diese Aufgabe ist nicht Bestandteil des Projektes OpenTelemetry, sondern muss von Monitoring Tools übernommen werden.
Heute stehen den Anwendern eine Vielzahl von Monitoring-Tools zur Auswahl. In diesem Kapitel werden zwei Monitore genauer betrachtet. Der erste Monitor ist das weit verbreitete Open-Source-Projekt Jaeger, das jedem Entwickler kostenlos zur Verfügung steht. Danach wird das Monitoring-Tool, Azure Monitor, vorgestellt, das komerziell von Microsoft vertrieben wird.

3.5.1 Jaeger

Jaeger ist ein Moinitoring-Tool für Distributed Tracing, das von Uber Technologies entwickelt wurde und als Open Source-Projekt veröffentlicht wurde. Es wird insbesondere zur Überwachung und Fehlerbehebung in verteilten Systemen auf der Basis von Mikrodiensten verwendet. Jaeger ist ein Werkzeug, das die Programmiersprachen Go, Java, Node, Python, C++ und C# unterstützt.
Das Jaeger-Backend wird als eine Sammlung von Docker-Images betrieben. Das Jaeger Web UI ist in Javascript implementiert und verwendet bekannte Open-Source-Frameworks wie React. Jeager ist in der Lage effizient mit großen Datenmengen umzugehen und kann Traces mit Zehntausenden von Spans anzeigen. [41]
Das Jaeger User Interface gliedert sich in zwei Bereiche. Beim Öffnen gelangt man in die sogenannte „Trace Search“-Ansicht.

Jaeger UI | Lookup by Trace ID... | Search | Compare | System Architecture

Search | JSON File

Service (12)
paymentservice

Operation (2)
all

Tags
http.status_code=200 error=true

Lookback
Last Hour

Min Duration
e.g. 1.2s, 100ms, 500us

Max Duration
e.g. 1.2s, 100ms, 500us

Limit Results
20

Find Traces

Abbildung 3.6: Jaeger UI-Trace Search

Auf dieser Seite muss zunächst der Service, der betrachtet wird, eingestellt werden. Alle weitere Optionen sind für den Benutzer optional auswählbar. So kann der Benutzer angeben, welche Operationen er in dem ausgewählten Service betrachten möchte. Über das Feld „Tags" können zum Beispiel bestimmte Status-Codes von HTTP abgefragt werden. Alle Einstellungen, die in Abbildung 3.6 zu sehen sind, unterstützen den Anwender dabei, spezifische Traces einfacher zu lokalisieren. [41]
Nachdem die gewünschten Einstellungen vorgenommen wurden, sucht Jaeger nach den

passenden Traces und listet diese nach ihrer Relevanz auf. Durch Auswählen eines Traces öffnet sich die Detailansicht des Traces (siehe Abbildung 3.6). Oben links ist der ausgewählte Service mit der aufgerufenen Operation und der dazugehörigen TraceID zu finden. Darunter werden der Startzeitpunkt des Traces, die Dauer, die Anzahl der Services und die Gesamtanzahl der Spans mit ihrer Maximaltiefe angezeigt. Anschließend werden die Spans in einem Zeitstrahl dargestellt. Jeder Span erhält dabei eine eigene Farbe, durch diesen Aufbau wird der Ablauf des Services und die Zusammenhänge der einzelnen Services übersichtlich dargestellt. Durch Aufklappen der einzelnen Spans können weitere Attribute wie Tags oder SpanID abgerufen werden. [41]

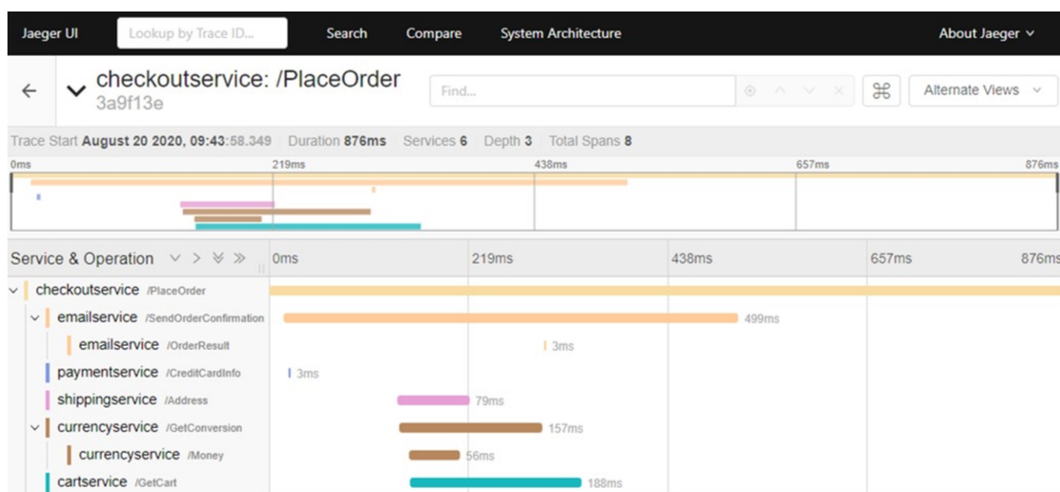

Abbildung 3.7: Jaeger UI-Detail

Zusammenfassend lässt sich sagen, dass das Jaeger-Monitoring-Tool eine gut durchdachte Weboberfläche für die Darstellung von Traces und Spans bietet. Durch die vielen Einstellmöglichkeiten wird dem Benutzer das Auffinden bestimmter Traces erleichtert. Durch die Unterstützung vieler Programmiersprachen und die Kompatibilität zum OpenTelemetry-Projekt ist dieses Tool für eine breite Masse von Interesse.

3.5.2 Azure Monitor

Azure Monitor ist im Gegensatz zu Jaeger eine kostenpflichtige Anwendung. Dieser Monitor wurde von Microsoft für ihre Cloud-Anwendung Microsoft Azure entwickelt. Azure Monitor ist die native Überwachungslösung für Azure, eignet sich aber auch (zum Teil) für andere Anwendungen. Es sammelt Metriken und Protokolle von allen verwendeten Azure-Ressourcen und kann eingesetzt werden, um Alerts zu erstellen, die Leistung zu überwachen, Probleme zu beheben und Dashboards zu erstellen, so dass die Entwickler oder Anwender einen vollständigen Überblick über ihre Azure-Anwendung haben.
Unternehmen nutzen Azure Monitor, um Überwachungsdaten von firmeninternen Rechenzentren, Azure-Ressourcen und Cloud-Management-Tools zu sammeln und zu organisieren. Der Dienst stellt anschließend all diese Informationen auf einer einzigen Oberfläche dar. Es verwendet maschinelles Lernen, um Probleme darzustellen und zu lösen. Dieser Dienst kann auch für die Netzwerkdiagnose und zur Analyse der Infrastruktur, virtueller Maschinen und Azure Kubernetes Service verwendet werden. [42]
Der Azure Monitor ist wie folgt aufgebaut/strukturiert:

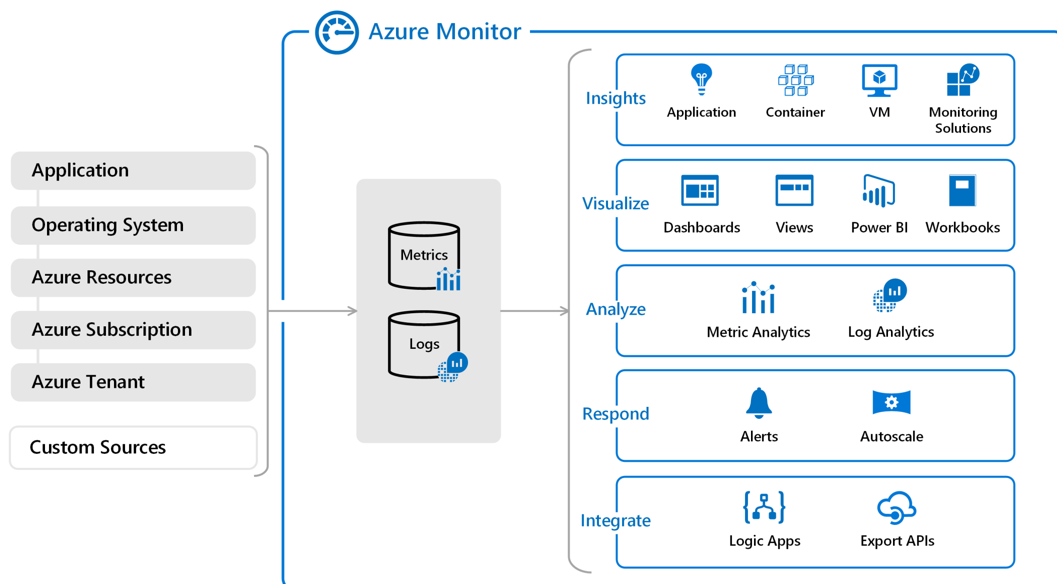

Abbildung 3.8: Aufbau von Azure Monitor [43]

Wie in Abbildung 3.8 zu sehen, werden die erzeugten Telemetriedaten an den Azure Monitor gesendet und in einer Datenbank gespeichert. Auf der rechten Seite der Abbildung sind die verschiedenen Funktionen abgebildet, die Azure Monitor anbietet, wie etwa Analyse, Ausgeben von Warnungen und Ausgabe der Daten. [42, 43]
Der Nachteil von Azure Monitor ist, dass dieser zur Überwachung von Komponenten und Diensten ausgelegt ist, die innerhalb von Microsoft Azure laufen. Grundsätzlich bietet der Azure Monitor auch offene Schnittstellen an, im Kern ist die dieses Tool aber für die Azure Umgebung konzipiert. Wer eine Multi-Cloud-Strategie verfolgt oder eine Überwachungslösung mit hoher Granularität angestrebt, für den ist dieser Monitor nicht die richtige Wahl. [44]

3.6 OpenTelemetry in der Praxis

In diesem Kapitel soll nun anhand von zwei Anwendungsbeispielen der Einsatz von OpenTelemetry in der Praxis gezeigt werden. Dabei werden zwei sehr unterschiedliche Wege aufgezeigt, wie Telemetriefunktionalität in Anwendungen integriert werden kann. Zu einen über den klassischen Weg der Erweiterung des Source-Codes der Anwendung und zum anderen über die neue „Auto-Instrumentation“ Funktionalität von OpenTelemetry.

3.6.1 HTTP-Transaktion

Im ersten Anwendungsbeispiel soll eine HTTP-Transaktion über Tracing nachverfolgbar gemacht werden. Da es sich bei einer HTTP-Transaktion um einen Client/Server-Prozess handelt, der in der Regel auf verschiedenen Systemen abläuft, muss eine Instrumentierung mit OpenTelemetry an unterschiedlichen Ablaufpunkten erfolgen: im HTTP-Client und im HTTP-Server.
Im vorliegenden Fall gehen wir von einer Java-Anwendung aus. Die hier beschriebene Vorgehensweise gilt aber auch entsprechend für andere Programmiersprachen, die von OpenTelemetry unterstützt werden.
Listing 3.2 zeigt, wie ein HTTP-Client in Java mit Tracing Funktionalität von OpenTelemetry instrumentiert werden kann.

Listing 3.2: Beispiel einer HTTP-Transaktion mit OpenTelemetry (Client Seite) [45]

```
1  public class HttpClient {
2
3    private static class HttpClientHandler implements HttpHandler {
4
5          //Erstellung des Spans, als Client Span
6   Span span = tracer.spanBuilder("/").setSpanKind(Span.Kind.CLIENT).
       startSpan();
7      try (Scope scope = tracer.withSpan(span)) {
8             //Setzen der Attribute des Spans
9        span.setAttribute("component", "http");
10       span.setAttribute("http.method", "GET");
11       span.setAttribute("http.url", url.toString());
12
13           //Anlegen der Anfrage mit dem aktuellen Span
14       OpenTelemetry.getPropagators().getHttpTextFormat().inject(Context.
           current(), connection, setter);
15
16       try {
17         // Durchfuehrung der Anfrage
18         connection.setRequestMethod("GET");
19         status = connection.getResponseCode();
20         BufferedReader in =
21             new BufferedReader(
22                 new InputStreamReader(connection.getInputStream(),
                     Charset.defaultCharset()));
23         String inputLine;
24         while ((inputLine = in.readLine()) != null) {
25           content.append(inputLine);
26         }
27         in.close();
28         // Beenden des Spans
29         span.setStatus(Status.OK);
30       } catch (Exception e) {
31         span.setStatus(Status.UNKNOWN.withDescription("HTTP Code: " +
             status));
32       }
33     } finally {
34       span.end();
35     }
36   }
37 }
```

Im ersten Schritt wird ein neuer Span erzeugt. Da dies auf Client Seite erfolgt, wird dieser als „Client Span“ deklariert. Im nächsten Schritt werden die Attribute des Spans gesetzt: http, GET und Uniform Resource Locator (URL). In Schritt 3 wird der HTTP-GET-Request an den HTTP-Server abgesetzt und der erzeugte Span mit übermittelt. Danach wird auf die Antwort der Server Seite gewartet und dann der Client Span geschlossen.
Entsprechend zum HTTP-Client muss auch der HTTP-Server mit entsprechender Tracing-Funktionalität von OpenTelemetry instrumentiert werden. Dies zeigt Listing 3.3.

Listing 3.3: Beispiel einer HTTP-Transaktion mit OpenTelemetry (Server Seite) [45]

```
1  public class HttpServer {
2
3    private static class HttpServerHandler implements HttpHandler {
4
5      public void handle(HttpExchange exchange) throws IOException
6
7      // Der Span Kontext aus der HTTP  Anfrage wird extrahiert
8        Context context =
9            OpenTelemetry.getPropagators()
10               .getTextMapPropagator()
11               .extract(Context.current(), exchange, getter);
12         // Erstellung des "Server"-Spans
13
14       Span span =
15           tracer
16               .spanBuilder("/")
17               .setParent(TracingContextUtils.getSpan(context))
18               .setSpanKind(Span.Kind.SERVER)
19               .startSpan();
20
21
22       try (Scope scope = tracer.withSpan(span)) {
23         // Setzen der Span Attribute
24         span.setAttribute("component", "http");
25         span.setAttribute("http.method", "GET");
26         span.setAttribute("http.scheme", "http");
27         span.setAttribute("http.host", "localhost:" + HttpServer.port);
28         span.setAttribute("http.target", "/");
29
30       // Prozessierung der HTTP Anfrage (server-seitig)
31
```

```
32        OutputStream os = exchange.getResponseBody();
33        os.write(response.getBytes(Charset.defaultCharset()));
34        os.close();
35
36
37        Attributes eventAttributes =
38            Attributes.of("answer", AttributeValue.stringAttributeValue(
                  response));
39        span.addEvent("Finish Processing", eventAttributes);
40
41          // Beenden des Spans
42        span.setStatus(Status.OK);
43        } finally {
44          // Close the span
45          span.end();
46        }
47      }
48  }
```

Nachdem die HTTP-Anfrage des Client an den Server übermittelt wurde, muss der Trace weiterverarbeitet werden. Zunächst wird der Span-Kontext aus der HTTP-Anfrage extrahiert. Im nächsten Schritt wird ein neuer „Child"-Span, deklariert als „Server-Span", mit seinen Attributen erstellt. Im Schritt 3 wird die Antwort des Servers an den Client zurück übermittelt. Anschließend wird der Server Span beendet.

3.6.2 Java Auto-Instrumentation

Als zweites Anwendungsbeispiel wurde ein Software-Projekt ausgewählt, das bei der Entwicklung auf den Einsatz von Telemetriedaten verzichtet hat. Bei dem gewählten Programm-Beispiel handelt es sich um eine Tierklink-Verwaltungssystem, das eine Spring-Applikation darstellt [46]. Den Entwicklern war es seiner Zeit zu aufwändig, eine Überwachung der Applikation mit Hilfe von Telemetriedaten zu implementieren. Mit der Veröffentlichung des neuen Beta-Release von OpenTelemetry wurde eine Bibliothek zur Verfügung gestellt, die es ermöglicht, Telemetriedaten in einer Anwendung zu erzeugen, ohne den Code zu verändern. Es wird dem Entwickler die Java-Agent-JAR zur Verfügung gestellt, die an jede Java 7+ Anwendung angehängt werden kann. Dieser Agent kann dynamisch Bytecode zur Erfassung von Telemetriedaten durch Auto-

Instrumentation aus einer Reihe populärer Bibliotheken und Frameworks erfassen. Dabei werden bis jetzt ca. 50 Bibliotheken und Frameworks unterstützt, darunter gRPC, Spring Web MVC und JDBC. Aktuell ist der oben genannte Auto Instrumentierungsagent noch auf die Programmiersprache Java beschränkt.
Diese Telemetriedaten können in einer Vielzahl von Formaten exportiert werden. Darüber hinaus kann der Agent und der Exporteur über die Konsole und Umgebungsvariablen konfiguriert werden. Dabei kann unter anderem festgelegt werden, welches Traceformat durch den Propagator verwendet wird. Dabei ist es auch möglich mehrere Formate gleichzeitig zu verwenden.
Das Ergebnis ist die Fähigkeit, Telemetriedaten aus einer Java-Anwendung ohne Code-Änderungen zu erfassen.
Für das Programmierbeispiel kommt jetzt dieser Java-Agent ins Spiel. Dazu lädt man die aktuelle Java-Agent JAR-Version von OpenTelmetry herunter und fügt diese dem Projekt bei. Dabei reicht es die Datei auf der obersten Datei-Ebene abzulegen. Mit dem Befehl

```
java -javaagent:opentelemetry-javaagent-all.jar -Dotel.exporter
=jaeger -Dotel.jaeger.service.name=Tierklink -jar target/*.jar
```

kann die Anwendung mit dem Agenten gestartet werden.
In diesem Beispiel ist die Ausgabe der Telemetriedaten auf den Exporter Jaeger konfiguriert. Alternativ stände noch der Exporter Zipkin zur Auswahl. Ohne Veränderung des Exporters, würden die Telemetriedaten standardmäßig über den OpenTelemetry-Kollektor ausgegeben werden.
Mit der Option „Dotel.jaeger.service.name=Tierklink“ wird der Name des Services festgelegt. Standardmäßig würde „Unknown“ als Service-Name stehen.
Damit die Telemetriedaten an das ausgewählte Backend geschickt werden können, muss dieses in einem Docker gestartet werden. Die passenden Befehle hierzu findet man entweder in den Programmierbeispielen von OpenTelemetry oder bei Docker-Hub. Mit dem Befehl

```
docker run --rm -it --name jaeger -p 16686:16686 -p 14250:
14250 jaegertracing/all-in-one:1.16
```

wird ein Docker-Container mit Jaeger generiert und gestartet. Das Empfangen von Daten wird standardmäßig auf den Port 14250 gesetzt. Um auf die UI zugreifen zu

können, muss im Browser die vordefinierte URL aufgerufen werden.
Mit diesen Befehlen sind nun alle Komponenten gestartet. Telemetriedaten werden bei jeder Aktion, die ein Benutzer ausführt, erzeugt.

Abbildung 3.9 zeigt das Ergebnis der Auto-Instrumentation für das Tierklinik-Verwaltungsprogramm. Der nachfolgende Trace wurde automatisch bei der Suche eines Hundes anhand seines Besitzers erzeugt.

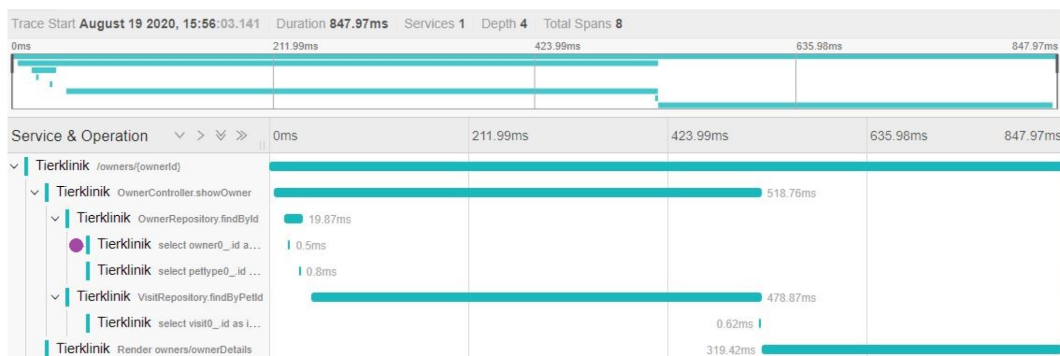

Abbildung 3.9: Trace Ablaufverfolgung

In diesem Beispiel hat der Trace acht Spans mit einer maximalen Tiefe von vier. In Abbildung 3.9 erkennt man die einzelnen Methoden, die aufgerufen werden, um die Tiere eines Besitzers zu ermitteln. Zuerst wird mit Hilfe des Nachnamens die passende Besitzer-ID aus der Datenbank ausgelesen. Anschließend wird mit dieser ID eine SQL-Anfrage gestartet, die die zugehörigen Tier-IDs ermittelt. Falls die SQL-Abfrage (lila markiert) eine zu lange Anfragezeit aufweist, könnte dies der Entwickler schnell erkennen und die Abfrage effizienter programmieren. Im letzten Span werden dem Benutzer die Details des Tieres bereitgestellt.
Sobald diese Anwendung den Benutzern nun zur Verfügung gestellt wird, kann das Entwicklungsteam durch die kontinuierliche Überwachung auf Anwendungsprobleme schnell reagieren. Weiterhin können Status-Seiten und weitere Metriken erstellt werden, um den Zustand der Infrastruktur effektiv zu bewerten.
Die in diesem zweiten Anwendungsbeispiel gezeigten neuen Möglichkeiten von OpenTelemetry stellen einen weiteren Erfolg für die Software Telemetrie dar. Ohne Eingriff in den Source-Code können nun auf einfache Art Telemetriedaten gewonnen werden.

Dies war bisher nur mit Programmieraufwand und entsprechenden Eingriffen in den Source-Code möglich.

3.7 Telemetrie im SDLC mit CI/CD-Pipeline

Wie in Kapitel 2.3 dargestellt, spielt die Telemetrie eine zentrale Rolle in der Monitoring-Phase des DevOps-SDLC. Daher sollte der Einsatz von Telemetriedaten in der frühen Phase des Systementwurfsprozesses mit eingeplant werden.
An Hand des Anwendungsbeispieles 3.6.1 erkennt man, wie bedeutsam fest definierte Abläufe für das Sammeln und Weiterleiten von Telemetriedaten bereits in der Entwicklungsphase der Software sind.
Darüber hinaus zeigt das Beispiel, dass die Verwendung von OpenTelemetry den Entwicklern eine Implementierung einer standardisierten Geschäftslogik für Telemetriedaten ermöglicht.
Die CI/CD-Pipeline ist das Herzstück der DevOps-Strategie. Sie sorgt für eine kontinuierliche Automatisierung und Überwachung über den gesamten SDLC hinweg, von der Integrations- und Test- bis hin zur Bereitstellungs- und Implementierungsphase.
Daher ist zu empfehlen, der CI/CD-Pipeline eine „Monitor+Observe“-Phase als weiteres Glied der Kette hinzuzufügen, wie in Abbildung 3.10 veranschaulicht.

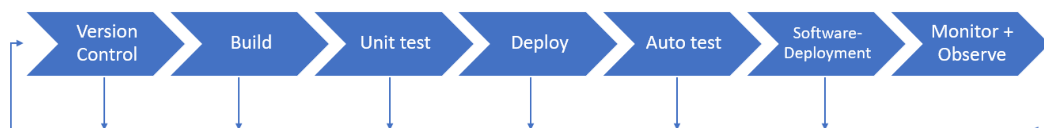

Abbildung 3.10: CI/CD-Pipeline mit Observability

Die „Monitor+Observe“-Phase wurde an das Ende der Pipeline angehängt, da Telemetriedaten in der Betriebsphase der Anwendung die Hauptrolle spielen.
Da die Automatisierung einer der wichtigsten Bestandteile einer effizienten CI/CD-Pipeline darstellt, macht es Sinn, auch die Observability und das Monitoring zu automatisieren. Dies sollte in der gleichen Art und Weise erfolgen, wie für Build, Test und Deployment.

Wie die CI/CD-Pipeline in der Praxis umgesetzt werden kann, lässt sich an Hand des Versionsverwaltungsprogramm GitLab darstellen. Für die „Monitor+Observe"-Phase bietet GitLab die Möglichkeit unterschiedliche Monitoring-Systeme zu integrieren. Zu einem wird das Monioring-Tool Jaeger für Traces, zum anderen Prometheus für Metriken unterstützt.

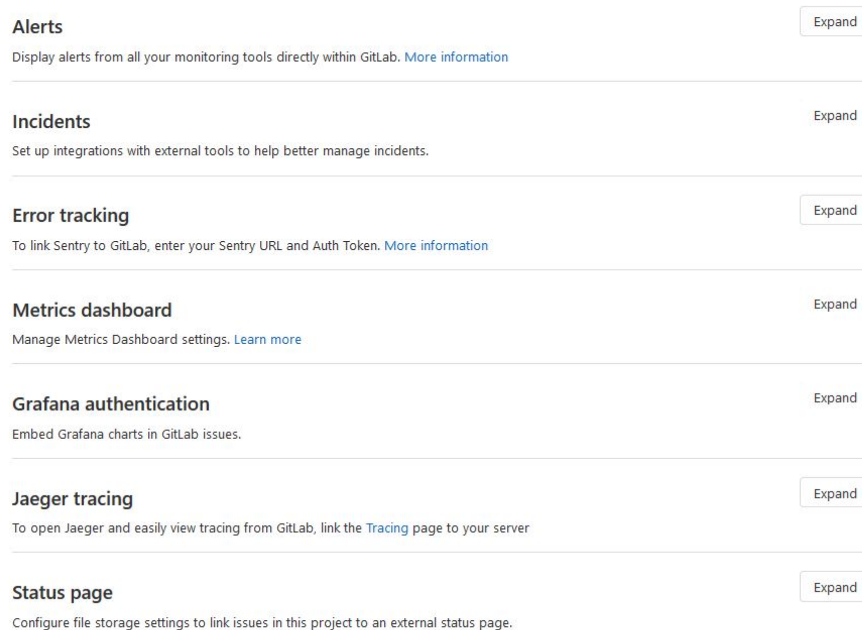

Abbildung 3.11: Gitlab-Einstellungen für Telemetriedaten

Wie in Abbildung 3.11 zu sehen, können verschiedene Optionen bezüglich Telemetriedaten eingestellt werden. Im Folgenden werden die ersten zwei Funktionen näher betrachtet, da diese die Automatisierung der „Monitor+Observe"-Phase unterstützen. Die Funktion „Alerts" ermöglicht dem Benutzer die eingestellten Warnungen von Monitoring-Tools automatisch nach GitLab zu übertragen und anzeigen zu lassen. Über die zweite Funktion „Incidents" können für die übertragenen Warnungen automatisiert neue Issues erstellt werden. Dieser Alarmierungsmechanismus, der die Pipeline für kontinuierliche Integration und kontinuierliche Lieferung ergänzt, ist für die Unterstützung der Technik und der Produktgeschwindigkeit von entscheidender Bedeutung.

4 Ergebnis

4.1 Evaluation

In diesem Kapitel soll das Projekt OpenTelemetry auf die in Kapitel 3.2 definierten Kriterien hin bewertet werden.

4.1.1 Interoperabilität

Interoperabilität beschreibt die Fähigkeit eines Programms oder Systems mit anderen Programmen oder Systemen nahtlos zu interagieren.
Bei der Implementierung jeder Art von verteilter Telemetrie ist die Interoperabilität ein zentraler Bestandteil. Es muss immer sicher gestellt sein, dass in einem Gesamtsystem die Telemetriedaten eines beliebigen Dienstes mit den Telemetriedaten eines anderen Dienstes kompatibel sind. Nur auf diese Weise ist es möglich, die Leistung der einzelnen Dienste im Ganzen zu verstehen.
OpenTelemetry löst dieses Problem, indem es einen weitreichenden Satz von Schnittstellen und Bibliotheken für seine Telemetrie Dienste bereitstellt. Somit stehen alle OpenTelemetry-Funktionen einer Vielzahl von Programmiersprachen zur Verfügung. Gleichzeitig ist gewährleistet, dass alle Telemetriefunktionen auf den unterschiedlichsten Systemplattformen zum Einsatz kommen können. OpenTelemetry präsentiert sich somit als interoperables Framework.
Der Entwickler muss sich nicht damit beschäftigen, wie zum Beispiel der Korrelations-Context und deren Identifikatoren weitergegeben werden. Alle Dienste funktionieren über Systemgrenzen hinweg. Jeder neue Dienst, der an die Plattform angeschlossen wird, spricht die gleiche Sprache, wenn es um die Generierung und Verbreitung von Telemetriedaten geht.
OpenTelemetry kann aufgrund seiner Herstellerneutralität eine Instrumentierungsoption sowohl für andere Open-Source-Projekte, die ihren Endbenutzern Telemetriedaten zur Verfügung stellen möchten, als auch für konventionelle Projekte sein.
Eine weiterer wichtiger Vorteil von OpenTelemetry ist die Abwärtskompatibilität zu ihren Vorgängerprojekten OpenCensus und OpenTracing. Damit bietet OpenTelemetry eine weitreichende Interoperabilität zu allen Softwareprojekten, die auf Basis von OpenCensus und OpenTracing entwickelt wurden.

4.1.2 Plattformunabhängigkeit

Plattformunabhängigkeit beschreibt in der Informationstechnik die Eigenschaft, dass ein Programm auf verschiedenen Plattformen ausgeführt werden. Plattformen werden dabei nach Architektur, Prozessor, Übersetzer, Betriebssystem, etc. unterschieden. Plattformunabhängigkeit ist ein klares Designziel von OpenTelemetry. Diese Plattformunabhängigkeit wird im Folgenden aus zwei unterschiedlichen Perspektiven betrachtet.
Die erste Perspektive ist die Unabhängigkeit von der verwendeten Programmiersprache. OpenTelemetry unterstützt heute bereits die gängigsten Programmiersprachen, im einzelnen: .NET, Java, JavaScript, C++, Python, Go, und Erlang. Somit kann OpenTelemetry bereits heute als weitgehend plattformunabhängig bezeichnet werden. Es ist davon auszugehen, dass im Rahmen der Weiterentwicklung von OpenTelemetry weitere Programmiersprachen hinzukommen, sodass das Projekt langfristig als vollständig plattformunabhängig eingestuft werden kann.
Auch aus der Perspektive der Monitoring-Tools kann behauptet werden, dass das Open-Telemetry-Projekt plattformunabhängig ist. Das Projekt gibt jedem Monitoring-Hersteller die Möglichkeit die OpenTelemetry Plattform zu unterstützen. Zu einem werden bereits vorhandene Bibliotheken von Drittherstellern unterstützt und zum anderen wird den Monitoring-Entwicklern eine einfache Schnittstellenumgebung angeboten.

4.1.3 Standardisierung

Standardisierung und Homogenisierung sind wichtige Faktoren in der Informatik. Ihr Ziel ist, zu einer erhöhten Transparenz und Kompatibilität beizutragen.
Die Standardisierung spielte bei der Entwicklung von OpenTelemetry eine wichtige Rolle. Zum einen haben die Entwickler alle existierenden Standards für Telemetriedaten übernommen. Dazu gehören die bereits erwähnte W3C-Trace-Context-Spezifikation und die W3C-Correlation-Context Spezifikation. Die Trace-API verwendet die weit verbreitete W3C Trace-Context-Spezifikation.
Darüber hinaus versucht OpenTelemetry selbst Standards zu entwickeln. Damit selbst definierte Eigenschaften an einen Span angefügt werden können, wurde die W3C-Correlation-Context-Spezifikation vorgestellt.
Bei der Entwicklung der Logging API, hat das OpenTelemetry-Team versucht, bereits

implementierte Bibliotheken zu unterstützen und gegebenenfalls für diese Mapper bereitzustellen.
Da OpenTelemetry auch andere Tracing-Bibliotheken unterstützt, die nicht nach den vorgestellten Standards aufgebaut sind, müssen die Telemetrie-Daten lediglich in das standardisierte Format konvertieren werden.
Es wäre ein großer Erfolg für das Projekt OpenTelemetry, wenn es selbst zum Standard in der Software-Telemetrie werden würde. Dabei ist die Frage unerheblich, ob OpenTelemetry zu einem offiziellen Standard wird oder es sich wie TCP/IP in der Kommunikationswelt zu einem de-facto Standard entwickelt.

4.1.4 Erweiterbarkeit

Im Rahmen des Software Development Life Cycle unterliegt die Software einer kontinuierlichen Veränderung. Ein wichtige Aufgabe ist dabei die Erweiterung der Software um neue Funktionalitäten.
Bezüglich der Erweiterbarkeit der OpenTelemetry Sofware lassen sich folgende Aussagen treffen.
Bei OpenTelemetry handelt es sich um Open-Source Software. Somit steht diese allen Software-Entwicklern zur Verfügung und kann im Rahmen der GNU Lizenz beliebig verwendet und erweitert werden.
Das OpenTelemetry-Projekt folgt einem klar strukturierten Aufbau und versucht die einzelnen Komponenten zu entkoppeln. Somit können problemlos neue Module entstehen und in das Framework eingebunden werden. Diese Art der Erweiterbarkeit stellt eine Grundvoraussetzung dar, damit sich eine Software erfolgreich am Markt positionieren kann.
Weiterhin können andere Hersteller sich problemlos an die offenen Schnittstellen von OpenTelemetry andocken und ergänzende Produkte bereit stellen. Gerade im Bereich der Monitoring Tools kann OpenTelemetry damit punkten. Die Bereitstellung eins Monitors gehört nicht zum Projektumfang von OpenTelemetry, wird aber durch die problemlose Erweiterbarkeit des Gesamtsystems und die Integrierbarkeit von Drittherstellern gelöst.
Eine weitere Möglichkeit für Hersteller am Telemetry-Ökosystem teilzunehmen, ist die Bereitstellung eines eigenen passenden Span und Metric Exporter. In diesen Exporter müssen die generierten Daten lediglich in das erforderliche Datenformat für das

Telemetrie-Backend konvertiert werden.
Zusammenfassend lässt sich festhalten, dass das Design von OpenTelemetry auf Erweiterbarkeit ausgelegt ist und somit vielen die Chance eröffnet, Teil des Telemetrie-Ökosystems zu werden.

4.1.5 Dokumentation

Dokumentation von Software wird oft als lästige Pflichtaufgabe angesehen. Dabei hängt die Akzeptanz von Software in starkem Maße von einer guten Dokumention ab.
Für OpenTelemetry steht eine umfangreiches Dokumentationspaket zur Verfügung. Auf der Website `www.opentelemetry.io` sind alle wichtigen Aspekte zum Projekt beschrieben. Um mehr über die einzelnen Komponenten zu erfahren, wird man zum OpenTelemetry GitHub Repository weitergeleitet.
Für jede unterstützte Programmiersprache existiert eine genaue Beschreibung, wie die OpenTelemetry Module richtig in das Software-Projekt einzubinden sind. Zusätzlich wird gezeigt, wie die wichtigsten Methoden in den jeweiligen Sprachen anzuwenden sind.
Um sich über den Funktionsumfang einen genaueren Überblick zu verschaffen, stehen Anwendungsbeispiele für verschiedenste Szenarien bereit. In diesen wird Schritt für Schritt erklärt, wie diese anzuwenden sind.
Auch für die Entwickler von Monitoring-Tools stehen umfangreiche Informationen bereit, um einen Exporter für die jeweilige Programmiersprache oder Collector schreiben zu können.
Zusammenfassend lässt sich festhalten, dass das OpenTelemetry-Team sehr viel Wert auf eine ausführliche Dokumentation gelegt hat, was eine wichtige Voraussetzung für die Akzeptanz der Software darstellt.

4.1.6 Mehrwert

Entscheidend für den Erfolg von Software-Projekten ist es, inwieweit sie sich am Markt von konkurrierenden Entwicklungen abheben können.
OpenTelemetry setzt, wie in Kapitel 3.3 erläutert, auf den Projekten OpenTrace und Open Census auf, entwickelt deren Stärken weiter und hebt die Software Telemetrie damit auf ein neues Niveau. OpenTelemetry unterscheidet sich in puncto Funktionalität,

Standardisierung und Plattformunabhängigkeit deutlich von seinen Vorgängerprojekten und andere Entwicklungen im Bereich der Telemetrie.
OpenTelemetry möchte nicht als eine Bibliothek für Traces, Metriken und Logs verstanden werden, es will viel mehr eine unabhängige Plattform für Telemetriedaten sein Sie möchte den Entwicklern von Systemen die Möglichkeit geben, Telemetriedaten mit ihrer bevorzugten Bibliothek zu erzeugen und diese an das gewünschte Monitoring-Backend zu senden.
An dieser Stelle sei nochmals auf den großen Mehrwert von OpenTelemetry verwiesen, wie er im Anwendungsbespiel in Kapitel 3.6 vorgestellt wurde. Ohne Eingriff in den Source Code konnte für die Anwendung Telemetriefunktionalität erzeugt werden.

4.2 Bewertung der Evaluation

Anhand den Evaluationskriterien kann festgehalten werden, dass das Projekt OpenTelemetry uneingeschränkt für Interoperabilität, Plattformunabhängkeit und Erweiterbarkeit steht. Unabdingbare Kriterien für den Erfolg einer Software.
Es ist noch zu früh zu bewerten, ob sich das Projekt OpenTelemetry in der Praxis durchsetzen wird. Dafür ist die Projekt noch zu jung und steht in der Anfangshase seiner Entwicklung. Es besitzt aber gute Chancen sich gegenüber anderen Projekten durchzusetzen. Da große Firmen wie Google und Microsoft an diesem Projekt beteiligt sind und auch die Vorgängerprojekte OpenTracing und OpenCensus bereits großen Zuspruch bei den Entwicklern erlangt haben, ist von einer positiven Entwicklung auszugehen.

4.3 Telemetrie & SDLC

Wie in Kapitel 3 dargestellt, können Telemetriedaten für den SDLC einen wichtigen Beitrag zur Qualitätssicherung und zur Qualitätsverbesserung leisten. Dies gilt sowohl für die Development-Phase, in der die Softwareentwicklung stattfindet, wie auch für die anschließende Betriebsphase, in der Software dem Anwender im Produktivbetrieb zur Verfügung steht. Aufgrund der erkannten Bedeutung von Telemetriedaten in der modernen Softwareentwicklung stellt sich die Frage, ob Telemetrie und Observability nicht einen fest verankerten Platz im SDLC erhalten sollen bzw. müssen.

Als Vorbild könnte hier die DevSecOps Entwicklung dienen. In dieser erweiterten Form des DevOps-SDLC wurde die Sicherheit der Anwendung und der zugehörigen Systeminfrastruktur integraler Bestandteil des SDLC. In ähnlicher Weise wäre auch die Integration von Telemetrie in den SDLC vorstellbar, wie Abbildung 4.1 zeigt.

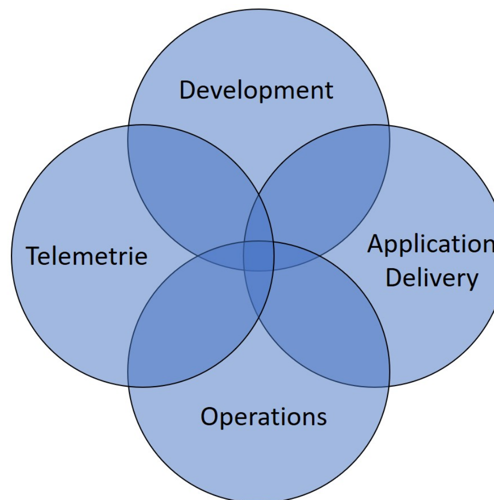

Abbildung 4.1: Telemetrie in Verbindung mit DevOps

Anwendungsbeispiel 3.6.1 hat gezeigt, dass die Einbindung von Telemetriefunktionalität in der Source-Code der Anwendung durchaus mit Aufwänden verbunden ist, auch wenn die Hürden hierfür durch konsequente Standardisierung niedrig gehalten werden können. Auf der anderen Seite hat die Erfahrung gezeigt, dass eine nachträgliche Instrumentierung der Anwendung mit Telemetriefunktionalität einen weitaus höheren Aufwand erfordert und somit mit wesentlich höheren Kosten verbunden ist. Daher macht es Sinn, Telemetrie gleich im ersten Entwicklungsschritt mit einzuplanen.
Auch für die Betriebsphase ist die Telemetrie zu einem unverzichtbaren Werkzeug geworden. Selbst wenn in der Software-Entwicklungsphase mit größter Sorgfalt gearbeitet wird, lassen sich niemals alle Fehler ausschließen und werden erst in der Betriebsphase erkannt. Fehler in der Betriebsphase haben in der Regel eine weit größere Auswirkung als in der Entwicklungsphase. Nun sind typischer Weise viele Anwender betroffen. Ohne für diesen Fall vorgesorgt zu haben, werden Fehler in der Betriebsphase von den Ent-

wicklern niemals schnell zu lokalisieren und zu beheben sein. Daher ist es unabdingbar vorzusorgen, damit Telemetriedaten in der Betriebsphase zur Verfügung stehen.
Es konnte gezeigt werden, dass viele gute Gründe existieren, Telemetrie in den SDLC zu integrieren. Wie dies in der Praxis umgesetzt werden kann und welche konkreten Mehrwerte damit erreicht werden können, bleibt einer weiteren Analyse vorbehalten.

5 Diskussion & Ausblick

5.1 Diskussion

Um die Ergebnisse dieser Arbeit diskutieren zu können, ist es nötig, einen Rückblick auf die in Kapitel 1.3 gesteckten Ziele zu werfen. Das grundlegende Ziel war zu zeigen, welchen Stand die Telemetrie in der heutigen Software-Entwicklung einnimmt und welche Bedeutung dies für die Software Entwicklung hat.
Es wurde gezeigt, dass Software Telemetrie zu einem unverzichtbarer Bestandteil für das Erreichen von Observability geworden ist. Teilaspekte von Observability können nur bedingt zu der gewünschten Verbesserung der Software beitragen. Nur die Kombination aller Telemetriedaten führt zu einer guten Beobachbarkeit des Gesamtsystems. Die Telemetrie in der Software-Entwicklung steht grundsätzlich noch am Anfang ihrer Entwicklung. Dies wurde auch in der durchgeführten Literatur-Recherche sichtbar. Aktuell existiert noch kein Buch auf dem Markt, das sich umfassend mit dem Thema Telemetrie in der Software-Entwicklung auseinander setzt. Die wenig vorhandenen Bücher konzentrieren sich auf Teilaspekte der Observability, meist auf das Thema Distributed Tracing.
Auch die Anzahl der Beiträge im Internet ist im Vergleich zu anderen Software Entwicklungsthemen als gering einzustufen. Die Beiträge sind in der Regel kurz und behandeln spezifische Aspekte.
Wie beschrieben, existiert bis zum heutigen Zeitpunkt noch kein universelles ausgereiftes Tool, das den Entwicklern bei der Beobachtbarkeit eines Systems optimal unterstützt. Um ein Software System beobachten zu können, müssen die Entwickler zum jetzigen Zeitpunkt auf unterschiedliche Werkzeuge zurückgreifen und diese miteinander kombinieren. Mit OpenTelemetry könnte in Zukunft diese Lücke geschlossen werden.

5.2 Ausblick

Die Ergebnisse aus Kapitel 4.3 zeigen, dass die Verzahnung von Software-Telemetrie und SDLC die Entwicklung von Software entscheidend verbessern kann.
Das entscheidende Ziel von Telemetrie ist, die entwickelte Software zu jeder Zeit beobachtbar zu machen. Dabei sollte der Grad der Beobachtbarkeit ständig optimiert werden, um den Entwicklern immer präzisiere Informationen liefern zu können.
In naher Zukunft könnte OpenTelemetry der de facto Standard für das Erzeugen und

Weiterleiten von Telemetriedaten werden. Das Anbieten einer unabhängigen Telemetrie Plattform könnte viele Entwickler überzeugen zukünftig auf OpenTelemetry zu setzen. Auch die Kontext-Propagierung, die OpenTelemetry eingeführt hat, wird zukünftig eine wichtige Rolle spielen. Diese Funktion ist eine Kernkomponente von plattformübergreifender Telemetrie. Der Mechanismus macht es erst möglich, Informationen über Komponenten und Maschinen hinweg kombinieren zu können. Verschiedene Tools propagieren unterschiedliche Kontexte – beispielsweise propagieren verteilte Trace-Tools TraceIDs, aber Metrik-Tools propagieren Tags. Diese Informationen müssen durch die Kontext-Propagierung sinnvoll kombiniert werden.
Des Weiteren versucht OpenTelemetry die Standardisierung von Telemetriedaten voranzutreiben. Dies ist ein wichtiger Schritt zu einer Vereinheitlichung von Telemetriedaten. Aktuell muss immer mit angegeben werden mit welchen Context kommuniziert werden soll, wie in Abbildung 5.1 zu sehen.

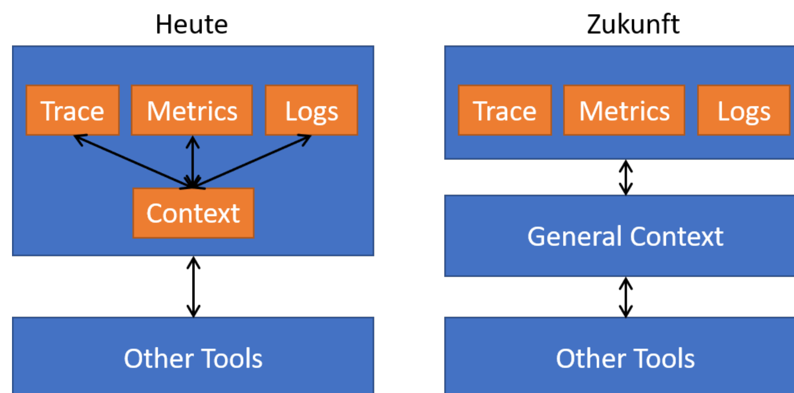

Abbildung 5.1: Aufbau des Context von Heute und in Zukunft

In Zukunft sollte nur noch ein Context für Telemetriedaten existieren, an den sich alle Programmiersprachen und Tools halten.
Auch die automatisierte Integration von Telemetriefunktionalitäten in bestehende Applikationen ohne Eingriff in den Source-Code (3.6.2) wird weiter an Bedeutung gewinnen und das Anwendungsspektrum für Telemetrie deutlich erweitern. Mit der Entwicklung von neuen Technologien sind auf diesem Gebiet noch große Fortschritte zu erwarten.

Literatur

[1] DocCheck Medical Services GmbH. *Telemetrie - DocCheck Flexikon*. 8.04.2020. URL: https://flexikon.doccheck.com/de/Telemetrie (besucht am 11. 05. 2020).

[2] Austin Parker. *Distributed tracing in practice: Instrumenting, analyzing, and debugging microservices*. First edition. Sebastopol, CA: O'Reilly Media, 2020. ISBN: 1492056634.

[3] Antonio Bucchiarone, Nicola Dragoni und Schahram Dustdar. *Microservices: Science and Engineering*. 1st ed. 2020. 2020. ISBN: 9783030316464. DOI: 10.1007/978-3-030-31646-4.

[4] *A short history of telemetry*. 2013. URL: http://orielsystems.com/short-history-telemetry/ (besucht am 26. 07. 2020).

[5] Vernon D. Jones. "Telemetry and Command Multiple-Mission Software (Model A)". In: *The Deep Space Network Progress Report* (1973), S. 163–166.

[6] James Turnbull. *The logstash book: Log management made easy*. Place of publication not identified, 2015. URL: http://proquest.tech.safaribooksonline.de/9780988820210.

[7] *Logstash: Sammeln, Parsen und Transformieren von Logdaten | Elastic*. URL: https://www.elastic.co/de/logstash (besucht am 27. 07. 2020).

[8] Vishal Sharma. *Beginning Elastic Stack*. Berkeley, CA: Apress, 2016. ISBN: 9781484216941. URL: https://ebookcentral.proquest.com/lib/gbv/detail.action?docID=4768389.

[9] *Was ist Elasticsearch? | Elastic*. URL: https://www.elastic.co/de/what-is/elasticsearch (besucht am 27. 07. 2020).

[10] Yuri Shkuro. *Mastering Distributed Tracing*. 1st edition. [Erscheinungsort nicht ermittelbar] und Sebastopol, CA: Packt Publishing und O'Reilly Media Inc, 2019. ISBN: 978-1-78862-759-7.

[11] *Ultimate Guide to System Development Life Cycle | Smartsheet*. 28.07.2020. URL: https://www.smartsheet.com/system-development-life-cycle-guide (besucht am 28. 07. 2020).

[12] Software & Systems Engineering Standards Committee of the IEEE Computer Society. *Systems and software engineering: Software life cycle processes*. Bd. ISO/IEC 12207. International standard. Geneva: ISO/IEC-IEEE, 2008. ISBN: 9780738156644. URL: http://ieeexplore.ieee.org/servlet/opac?punumber=4475822 (besucht am 20. 08. 2020).

[13] Uwe Vigenschow, Björn Schneider und Ines Meyrose. *Soft Skills für IT-Führungskräfte und Projektleiter: Softwareentwickler führen und coachen, Hochleistungsteams aufbauen*. 3. Aufl. s.l.: dpunkt, 2016. ISBN: 3864903955. URL: http://gbv.eblib.com/patron/FullRecord.aspx?p=4561817.

[14] Vogel Communications Group GmbH & Co. KG. *Projekt OpenTelemetry geht an den Start*. URL: https://www.dev-insider.de/projekt-opentelemetry-geht-an-den-start-a-898477/ (besucht am 18. 04. 2020).

[15] *DevOps Definition, Ziele und Best Practices, Tools (Docker, Jenkins, Selenium)*. URL: https://www.qytera.de/blog/devops-konzept-tools (besucht am 03. 09. 2020).

[16] *LifeCycle of DevOps*. 17.01.2020. URL: https://dzone.com/articles/life-cycle-of-devops (besucht am 10. 05. 2020).

[17] *Application Performance Monitoring vs. Observability Silo – The New Stack*. 2019. URL: https://thenewstack.io/application-performance-monitoring-vs-observability-silo/ (besucht am 10. 05. 2020).

[18] Administrator. *DevOps-Lexikon: Der zentrale chronische Konflikt*. 2018. URL: https://deutsche-devops-akademie.de/2018/08/devops-lexikon-der-zentrale-chronische-konflikt/ (besucht am 19. 08. 2020).

[19] Cindy Sridharan. *Distributed systems observability: A guide to building robust systems*. First edition. Sebastopol, CA: O'Reilly Media, 2018. ISBN: 9781492033424. URL: http://proquest.tech.safaribooksonline.de/9781492033431.

[20] Christian Ullenboom. *Rheinwerk Computing :: Java 7 - Mehr als eine Insel - 20 Logging und Monitoring*. 3.01.2020. URL: http://openbook.rheinwerk-verlag.de/java7/1507_20_001.html (besucht am 15. 04. 2020).

[21] Robert Schanze. *Was ist ein Log-File/Log-Datei? – Einfach erklärt*. 2018. URL: https://www.giga.de/downloads/windows-10/specials/was-ist-ein-log-file-log-datei-einfach-erklaert/ (besucht am 15. 04. 2020).

[22] Vogel Communications Group GmbH & Co. KG. *Was ist eine Log-Datei?* URL: https://www.ip-insider.de/was-ist-eine-log-datei-a-794350/ (besucht am 15. 04. 2020).

[23] *OpenTelemetry 101: What Is Tracing? | Lightstep Blog*. URL: https://lightstep.com/blog/opentelemetry-101-what-is-tracing/ (besucht am 16. 04. 2020).

[24] *PHP-Monitoring: Das Zusammenspiel von Services und Anwendungen.* 2019. URL: https://entwickler.de/online/php/php-monitoring-579907445.html (besucht am 16. 04. 2020).

[25] *OpenCensus.* 6.03.2020. URL: https://opencensus.io/stats/ (besucht am 17. 04. 2020).

[26] *OpenTelemetry.* URL: https://opentelemetry.io/about/ (besucht am 17. 04. 2020).

[27] *§ 15 TMG - Einzelnorm.* 19.09.2020. URL: https://www.gesetze-im-internet.de/tmg/__15.html (besucht am 19. 09. 2020).

[28] *IP-Adressen und andere Nutzungsdaten - Häufig gestellte Fragen - ULD.* URL: https://www.datenschutzzentrum.de/artikel/575-IP-Adressen-und-andere-Nutzungsdaten-Haeufig-gestellte-Fragen.html (besucht am 19. 09. 2020).

[29] *JSON.* 28.01.2020. URL: https://www.json.org/json-de.html (besucht am 14. 09. 2020).

[30] brianlic-msft. *Windows10, Version 1809, grundlegende Diagnoseereignisse und -felder (Windows10) - Windows Privacy.* URL: https://docs.microsoft.com/de-de/windows/privacy/basic-level-windows-diagnostic-events-and-fields-1809 (besucht am 31. 05. 2020).

[31] Sjoera Nas, Floor Terra, Jill Baehring. "DPIA Office 365 Online and mobile Office apps". In: (). (Besucht am 01. 06. 2020).

[32] *Projekt OpenTelemetry: Dynatrace, Google und Microsoft arbeiten an Datentransparenz | storageconsortium.de.* URL: https://storageconsortium.de/content/content/projekt-opentelemetry-dynatrace-google-und-microsoft-arbeiten-datentransparenz (besucht am 18. 04. 2020).

[33] *OpenCensus.* 20.07.2020. URL: https://opencensus.io/ (besucht am 20. 08. 2020).

[34] *OpenCensus.* URL: https://github.com/census-instrumentation (besucht am 20. 08. 2020).

[35] *OpenTelemetry.* URL: https://opentelemetry.io/docs/ (besucht am 20. 08. 2020).

[36] *OpenTelemetry.* URL: https://github.com/open-telemetry (besucht am 20. 08. 2020).

[37] John Watson and Lavanya Chockalingam und pwpadmin. *What Is The OpenTelemetry Specification And How Does It Work?* 2020. URL: https://blog.newrelic.com/product-news/what-is-opentelemetry/ (besucht am 09. 08. 2020).

[38] Alex Sokolov. *What's the difference between SDK and API?* URL: https://dev.to/alexsklv/what-s-the-difference-between-sdk-and-api-22b6 (besucht am 12. 08. 2020).

[39] *open-telemetry/opentelemetry-specification*. URL: https://github.com/open-telemetry/opentelemetry-specification/blob/master/specification/trace/api.md (besucht am 09. 08. 2020).

[40] *open-telemetry/opentelemetry-specification*. URL: https://github.com/open-telemetry/opentelemetry-specification/blob/master/specification/metrics/api.md (besucht am 09. 08. 2020).

[41] *Introduction*. URL: https://www.jaegertracing.io/docs/1.18/ (besucht am 20. 08. 2020).

[42] Bapi Chakraborty und Shijimol Ambi Karthikeyan. *Understanding Azure Monitoring: Includes IaaS and PaaS Scenarios*. 1st ed. 2019. 2019. ISBN: 978-1-4842-5130-0. DOI: 10.1007/978-1-4842-5130-0.

[43] bwren. *Azure Monitor-Übersicht - Azure Monitor*. URL: https://docs.microsoft.com/de-de/azure/azure-monitor/overview (besucht am 21. 08. 2020).

[44] *Microsoft Azure Services Monitoring and Tracing - Instana*. 2019. URL: https://www.instana.com/blog/microsoft-azure-services-monitoring-and-tracing/ (besucht am 21. 08. 2020).

[45] *open-telemetry/opentelemetry-java*. URL: https://github.com/open-telemetry/opentelemetry-java/tree/master/examples/http (besucht am 19. 09. 2020).

[46] *spring-projects/spring-petclinic*. URL: https://github.com/spring-projects/spring-petclinic (besucht am 20. 08. 2020).

EIDESSTATTLICHE ERKLÄRUNG

Hiermit versichere ich, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe. Alle Ausführungen, die anderen veröffentlichten oder nicht veröffentlichten Schriften wörtlich oder sinngemäß entnommen wurden, habe ich kenntlich gemacht.

Die Arbeit hat in gleicher oder ähnlicher Fassung noch keiner anderen Prüfungsbehörde vorgelegen.

Heilbronn, 21.09.2020 M. Graf

Ort, Datum Unterschrift