Konzept und Implementierung zur Observability für microservicebasierte Anwendungen

BACHELOR THESIS

Daniel Fabrikantow

Eingereicht am 02. Dezember 2021

Friedrich-Alexander-Universität Erlangen-Nürnberg
Technische Fakultät, Department Informatik
Professur für Open-Source-Software

Supervisor:
Georg Schwarz, M.Sc.
Prof. Dr. Dirk Riehle, M.B.A

Versicherung

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

_____

Erlangen, 02. Dezember 2021

License

This work is licensed under the Creative Commons Attribution 4.0 International license (CC BY 4.0), see https://creativecommons.org/licenses/by/4.0/

_____

Erlangen, 02. Dezember 2021

Abstract

‚Microservices‘ are nowadays a well-known and popular architecture pattern. Many world-famous tech companies have chosen this form over others. In addition to its many advantages, the decoupling and splitting of tasks into smaller services also brings a number of challenges. A central negative point with regard to the development of these services is the difficulty of troubleshooting and keeping track of the overall application.

In this thesis, some software tools for monitoring and aggregating log information are presented. In addition, a combination of such programs is selected for the development of a concept and to show the exemplary implementation of these tools in an ongoing open source project.

Zusammenfassung

‚Microservices‘ sind in der heutigen Zeit ein bekanntes und beliebtes Architekturmuster. Viele weltbekannte Tech-Unternehmen haben sich für diese entschieden. Die Entkopplung und die Aufteilung der Aufgaben in kleinere Services bringen neben daraus resultierenden Vorteilen auch Herausforderungen mit sich. Einen zentralen Negativpunkt hinsichtlich der Entwicklung dieser Dienste stellen die erschwerte Fehlersuche sowie die Schwierigkeit dar, den Überblick über die Anwendung als Gesamtes zu behalten.

In dieser Arbeit werden Softwaretools zur Überwachung und zur Aggregation von Log-Informationen vorgestellt. Darüber hinaus wird eine Kombination von Programmen gewählt, um ein Konzept zu entwickeln und eine beispielhafte Implementierung dieser Werkzeuge in ein bereits laufendes Open-Source-Projekt zu präsentieren.

Inhaltsverzeichnis

1 Einleitung 1

1.1 Ziel der Arbeit 2

1.2 Aufbau der Arbeit 2

2 Grundlagen 3

2.1 Microservices im Vergleich zur monolithischen Architektur 3

2.2 Was ist Observability? 4

3 Vergleich bekannter Technologien 6

3.1 Logging 7

3.1.1 Prometheus 7

3.1.2 Logstash 8

3.1.3 Fluentd 9

3.2 Monitoring 9

3.2.1 Grafana 9

3.2.2 Kibana 10

3.2.3 OpenSearch 11

3.3 Wahl der Softwarekette 12

4 Anforderungen 13

4.1 Funktionale Anforderungen 13

4.1.1 Logging 13

4.1.2 Monitoring 14

4.2 Nichtfunktionale Anforderungen 14

5 Konzeption 16
5.1 Logstash 16
5.2 Filebeat 18
5.3 OpenSearch und OpenSearch Dashboards 19
5.4 Zielarchitektur 20
6 Umsetzung 22
6.1 Vorbereitung 22
6.2 Integration von Logstash 24
6.3 Bisheriges Logging in ODS 26
6.4 Zu ergänzende Anbindung an das ODS 27
7 Auswertung 30
7.1 Funktionale Anforderungen 30
7.2 Nichtfunktionale Anforderungen 32
7.3 Zusammenfassung 33
8 Ausblick 35
Literaturverzeichnis 36
Anhang 38

Abkürzungsverzeichnis

AGPL3 GNU Affero General Public License Version 3.0

API Application Programming Interface

AWS Amazon Web Services

ELK Elasticsearch, Logstash, Kibana

GPL3 GNU General Public License Version 3.0

HTTP Hypertext Transfer Protocol

JSON JavaScript Object Notation

ODS Open Data Service

OSS Open-Source-Software

REST Representational State Transfer

SSPL Server Side Public License Version 1.0

SQL Structured Query Language

TCP Transmission Control Protocol

UDP User Datagram Protocol

VM Virtuelle Maschine

YAML YAML Ain’t Markup Language

1 Einleitung

In der Informatik herrscht ein stetiger Wandel durch neue Technologien und Möglichkeiten, die Effizienz und die Produktivität zu steigern, sowie durch eine schnelle Anpassung an neue Nutzeranforderungen. Es entwickeln sich damit regelmäßig neue Trends bei Software- und Architekturkonzepten, die sich in der Industrie durchsetzen und im Vergleich zur altbekannten Technik Vorteile bieten. So konnte sich in den letzten Jahren eine Art Microservice-Bewegung ausweiten. Immer mehr Unternehmen wie Netflix, Amazon oder Ebay entscheiden sich für eine microservicebasierte Architektur (Richardson, 2021, S. Known uses). Bei der richtigen Anwendung bietet dieses Architekturmuster einen großen Nutzen durch Skalierbarkeit oder unabhängige Bereitstellung (Pacheco, 2018, S. 7).

Allerdings werden mit jeder neuen Lösung auch neue Herausforderungen geschaffen, die es zu bewältigen gilt. Das Entwickeln mehrerer eigenständiger Services, die miteinander kollaborieren, kann die Fehlersuche während der Entwicklung erschweren und somit die eigentliche Entwicklungszeit verlängern. Denn das Überwachen mehrerer untereinander kommunizierender Systeme – mit teils unterschiedlich genutzten Sprachen – kann schnell anspruchsvoll werden, da ein konventionelles Debugging hier nicht mehr möglich ist.

Diesen Arbeitsaufwand gilt es mithilfe eines zentralen Dienstes in einer Microservice-Umgebung zu vereinfachen. Der Dienst hilft dabei, den Überblick über das Gesamtsystem zu behalten, den Zustand des Systems zu überwachen und Fehler bis zu ihrem Ursprung zurückzuverfolgen. Diese Punkte können auch unter dem Begriff ‚Observability' zusammengefasst werden.

Ein solcher Service wird in dieser Arbeit konzeptionell vorgestellt sowie anhand des Open-Source-Projekts Open Data Service (ODS) beispielhaft umgesetzt. Bei ODS handelt es sich um „eine Open-Source-Software, die an der Professur für Open-Source-Software der Friedrich-Alexander-Universität Erlangen-Nürnberg entwickelt wird. Das Projekt stellt ein einfach zu konfigurierendes System bereit, das die Aufgaben des Datenabrufs, der Datenaufbereitung und das Bereitstellen der Daten übernehmen kann" (Werner, 2019, S. 1).

1.1 Ziel der Arbeit

Der weitreichende Markt in der Softwareentwicklung, vor allem im Bereich von microservicebasierten Open-Source Projekten, verfügt bereits über eine Reihe von nützlichen Werkzeugen zum Aggregieren von Log-Informationen und das Anzeigen dieser mithilfe von Dashboards. Ziel dieser Bachelorarbeit ist es, einen Überblick über den heutigen Stand einiger Open-Source-Softwares in diesem Bereich zu geben, diese zu vergleichen sowie sich anhand bestimmter Kriterien auf eines dieser Programme festzulegen und als weiteren Service beispielhaft an den ODS anzubinden. Das zentralisierte Logging und das Monitoring sollen dem ODS zugutekommen und das Team dabei unterstützen, Fehler in der Anwendung schnell zu finden und einen möglichen Ausfall zeitnah beheben zu können.

1.2 Aufbau der Arbeit

Nach dieser Einleitung zusammen mit einer kurzen Beschreibung des Ziels dieser Bachelorarbeit werden in Kapitel 2 zunächst die Grundlagen erläutert, um ein besseres Verständnis elementarer Begriffe für den weiteren Verlauf zu gewährleisten.

Nach diesem Überblick über relevante Kernbegriffe werden anschließend in Kapitel 3 verschiedene Open-Source-Softwares präsentiert. Zudem werden Kriterien erläutert, die zur Auswahl von AWS OpenSearch führten und für den weiteren Verlauf dieser Arbeit entscheidend waren.

In Kapitel 4 werden die Anforderungen an das Konzept sowie die Umsetzung der Anbindung an den ODS festgelegt.

Das Designkonzept der verwendeten Programme sowie die Zielarchitektur des ODS werden in Kapitel 5 näher beschrieben.

In Kapitel 6 werden die beispielhafte Umsetzung und die Implementierung dieses Konzepts erläutert. Hier wird neben der technischen Anbindung an ODS auch auf das zugehörige Dashboard eingegangen.

Die Überprüfung der in Kapitel 4 festgelegten Anforderungen und eine abschließende Zusammenfassung finden in Kapitel 7 statt.

Kapitel 8 bietet einen Ausblick auf mögliche Erweiterungen und Schwächen des ODS hinsichtlich Logging und Monitoring.

2 Grundlagen

2.1 Microservices im Vergleich zur monolithischen Architektur

Zum besseren Verständnis der Problemstellung wird zunächst das Architekturkonzept von Microservices näher erläutert. Hierzu ist es hilfreich, eine dazu gegensätzliche Architekturform, den Monolithen, zu betrachten und mit den Microservices zu vergleichen.

Wie der Name vermuten lässt, basiert der traditionelle Ansatz auf der Vorstellung eines großen Blocks aus Stein, der ein einzelnes zusammenhängendes Programm darstellt. Somit sind alle Funktionen, Daten, sämtliche Logik und das User Unterface an eine Einheit gebunden und benötigen keine API zur Kommunikation untereinander (Demchenko, 2020, s. What is a monolithic architecture). Innerhalb eines Monolithen kann der Code einem Organisationsgerüst folgen, wie dem MVC (Model-View-Controller), dem HMVC (Hierarchical Model View Controller) oder der MTV (Model Template View). Damit kann bedenkenlos eine Web-Applikation gestartet werden (Pacheco, 2018, S. 6). Bei einer Wartung oder einer Erweiterung der Applikation treten hingegen mit zunehmender Komplexität Probleme auf. Einige schwerwiegende sind z. B.:

- Das Versagen einer einzelnen Komponente kann zu einem Programmabsturz führen.
- Aufbau und Bereitstellung nur des gesamten Systems sind möglich.
- Jede Änderung am Code beeinflusst das gesamte Netzwerk (feste Verkopplung einzelner Komponenten).

Mit Hinblick auf diese Nachteile eines Monolithen wurde das Architekturkonzept von Microservices entwickelt. Im Gegensatz zur monolithischen Architektur sind Microservices eigenständige und „voneinander unabhängig deploybare Services“, wie Fowler und Lewis in ihrem Fachartikel zu Microservices schreiben (Fowler & Lewis, 2015). Diese Dienste können gemeinsam eine verteilte Anwendung repräsentieren und

laufen jeweils in eigenen Prozessen, die meist durch RESTful APIs miteinander kommunizieren. Am häufigsten werden hierfür HTTP-Protokolle verwendet (Fowler & Lewis, 2015). Den größten Nutzen dieses Architekturkonzepts bieten, neben den gegenteiligen Aspekten der oben genannten Probleme des Monolithen, folgende Punkte (Richardson, 2021, s. Benefits):

- Neue Technologien können mit jeweils passender Programmiersprache als weiterer Service an die Anwendung angebunden werden.
- Einzelne Dienste sind vergleichsweise klein und damit verständlicher und schneller zu entwickeln.

Auch wenn die Teilung verschiedener Komponenten in einzelne Prozesse Vorteile mit sich bringt, entstehen dadurch auch Nachteile. In erster Linie wird der Ressourcenverbrauch erhöht. Um eine Isolation der Instanzen gewährleisten zu können, müssen diese in teils eigenen virtuellen Maschinen (VM) gestartet werden (Richardson, 2021, s. Drawbacks). Ein ebenso einschneidender und zentraler Aspekt dieser Arbeit ist, dass einzelne Services jederzeit ausfallen können. Ursachen für Fehlverhalten müssen schnell erkannt und beseitigt werden (Fowler & Lewis, 2015, s. Design for failure). Jeder dieser Services kann in unterschiedlichen Sprachen geschrieben sein, um seinem Zweck besser dienen zu können, und verfügt möglicherweise über seine eigene Versionskontrolle. Hier besteht die Notwendigkeit eines anspruchsvollen und zentralisierten Loggings und Monitorings der gesamten Microservice-Anwendung.

In diesem Zusammenhang ist allerdings zu erwähnen, dass Microservices die monolithische Architektur nicht grundsätzlich ersetzen sollen, denn während sich Monolithen für einige Problematiken gut eignen, kommen Microservices für andere Problemstellungen eher in Frage. Vielmehr ist das Wissen darüber erforderlich, wie und wann die beste Lösung oder die Kombination von Lösungen an die sich entwickelnden Anforderungen eines Projekts angepasst werden müssen (Figueroa, 2019, s. There is no "versus").

2.2 Was ist Observability?

Vor allem in microservicebasierten Anwendungen, in denen Teams meist nach einzelnen Diensten aufgeteilt werden, kann die Observability dazu beitragen, spezifische Fragen bezüglich funktionsübergreifender Vorgänge in verteilten Systemen besser zu verstehen und zu beantworten. Funktioniert ein Service zu langsam oder ist er defekt, können Warnungen diesbezüglich mithilfe einer Observability-Lösung sofort erhalten und die Probleme präventiv gelöst werden. Dadurch kann eine Leistungsverbesserung erzielt werden, noch bevor die Problematik sich negativ auf die Nutzer auswirken kann. Aus diesem Grund schreiben Kasun

Indrasiri und Prabath Siriwardena in ihrem Buch ‚Microservices for the Enterprise': „Observability is the measure of how well internal states of a system can be inferred from knowledge of its external outputs. It is one of the most important aspects, which needs to be baked into any microservices design" (Indrasiri & Siriwardena, 2018).

Im Zuge dieser Erkenntnis schreiben sie in ihrem Buch über drei fundamentale Säulen: Logging, Metriken und Rückverfolgung. Mittels dieser drei Säulen lässt sich die Observability beschreiben und erreichen.

Beim *Logging* geht es um die Aufzeichnung von möglichst allen Ereignissen. Denn jede Transaktion, die über einen Microservice läuft, kann auch protokolliert werden, einschließlich seiner zugehörigen Metadaten, wie seinem Status oder seinem Zeitstempel (Indrasiri & Siriwardena, 2018).

Metriken können verschiedene Werte annehmen und sind ein Indikator dafür, wie gut (oder wie schlecht) ein Service zu arbeiten scheint. Soll ein Alerting-System genutzt werden, das heißt ein Warnmeldesystem, das bei selbst definierten Schwellenwerten einen Alarm auslöst, so wird dafür im Regelfall auf Metriken zurückgegriffen. Diese zeichnen Trends der einzelnen Dienste ab, womit proaktiv gehandelt werden kann. Typische Metriken sind beispielsweise die Anzahl an verarbeiteten Transaktionen pro Zeiteinheit, die Ausfallrate eines Microservices, die Speichernutzung, die Prozessorauslastung oder die durchschnittliche Latenz eines bestimmten Dienstes, der über die Logs mit Zeitstempel abgeleitet werden kann (Indrasiri & Siriwardena, 2018).

Als letzte grundlegende Säule wird die Ablaufverfolgung aufgeführt, die auch Tracing genannt wird. Auch hier wird die Rückverfolgung aus den Logs hergeleitet und dient zur weiteren Ansicht des Systems. Unter Berücksichtigung der Reihenfolge einzelner Ereignisse kann die Grundursache eines Fehlverhaltens schneller identifiziert werden. Dabei kann das Trace nicht nur über mehrere Services hinweg verzeichnet werden, sondern auch innerhalb eines einzelnen (Indrasiri & Siriwardena, 2018).

3 Vergleich bekannter Technologien

Beim Vergleich moderner Logging- und Monitoring-Werkzeuge mit Hinblick auf die Anbindung an den ODS ist die Lizenz der Open-Source-Software eines der wesentlichen Kriterien dafür, dass diese mit der eigenen Lizenz gut kombinierbar ist. Dabei verwendet der ODS eine sogenannte GNU Affero General Public License 3.0 (AGPL3).

Zusammenfassend ist AGPL3 eine Abwandlung der GNU General Public License 3.0 (GPL3) und ergänzt diese um eine weitere Bedingung: Webserver, die ein modifiziertes Programm ausführen, das mit anderen Nutzern interagiert, müssen eine Kopie des entsprechend veränderten Quellcodes zum Herunterladen bereitstellen (Free Software Foundation, 2015).

Die anknüpfenden Softwaretools sollen hingegen entweder die MIT License oder die Apache License 2.0 besitzen. Eine kurze tabellarische Zusammenfassung dieser beiden Lizenzen sowie von weiteren im Verlauf vorgestellten Technologien sind in Anhang 1 zu finden.

Weitere vorteilhafte Eigenschaften sind eine Volltextsuche, mit deren Hilfe nach konkreten Fehlern gesucht werden kann, eine ansprechende und leicht anpassbare Benutzeroberfläche sowie eine aktive Community und eine ausführliche Dokumentation. Um möglichst viele Bedingungen abzudecken, ist eine Kombination mehrerer Programme beinahe unumgänglich. Aus diesem Grund ist das abschließende Kriterium die nahtlose Kompatibilität der verwendeten Werkzeugkette.

Abbildung 3.1 und Abbildung 3.2 zeigen tabellarische Aufzählungen heutiger Open-Source-Softwares im Bereich von Logging und Monitoring. Neben der zugehörigen Lizenz sind weitere Metriken genannt, die eine lebhafte Community symbolisieren sollen. GitHub-Sterne können ein Indikator dafür sein, dass sich Repositorys an Popularität erfreuen und sich viele Nutzer ein bestimmtes Repository vormerken möchten. GitHub Commits können zusammen mit der neuesten Softwarefreigabe einen Anhaltspunkt für die aktive Entwicklung und die Aktualität des Programms darstellen. Schließlich ist die Anzahl an Mitwirkenden abgebildet, um die Größe eines

Projekts darzustellen. Auf die einzelnen Programme wird in den jeweiligen Punkten unter 3.1 Logging und 3.2 Monitoring genauer eingegangen.

	Lizenz	GitHub-Sterne	GitHub Commits	neueste Freigabe	Mitwirkende
Prometheus[1]	Apache-2.0[2]	39 155	9262	06.10.2021 (v2.30.3)	622
Logstash[3]	Apache-2.0	12 420	9876	14.10.2021 (v7.15.1)	466
Fluentd[4]	Apache-2.0	10 646	6164	29.09.2021 (v1.14.1)	220

Abbildung 3.1: Tabellarische Aufzählung von Open-Source-Softwares im Bereich des Loggings

	Lizenz	GitHub-Sterne	GitHub Commits	neueste Freigabe	Mitwirkende
Grafana[5]	AGPL3	44 350	31 569	11.10.2021 (v8.2.1)	1568
Kibana[6]	Doppellizenz[7]	16 622	47 201	14.10.2021 (v7.15.1)	715
OpenSearch[8]	Apache-2.0	3947	55 050	06.10.2021 (v1.1.0)	64

Abbildung 3.2: Tabellarische Aufzählung von Open-Source-Softwares im Bereich des Monitorings

3.1 Logging

3.1.1 Prometheus

Prometheus ist ein Open-Source-Projekt, das ursprünglich von SoundCloud entwickelt wurde. Es dient vor allem dem Monitoring und Echtzeit-Warnmeldungen von numerischen Metriken. Eingesetzt werden kann das Programm in maschinenzentrierten Systemen, es eignet sich aber vor allem für microserviceorientierten Umgebungen. Hierfür spricht insbesondere die

[1] https://github.com/prometheus/prometheus (Zugriff am 15.10.2021)
[2] Nur außerhalb des „x-pack“ Ordners und falls nicht anders gekennzeichnet zu Beginn einer Datei
[3] https://github.com/elastic/logstash (Zugriff am 15.10.2021)
[4] https://github.com/fluent/fluentd (Zugriff am 15.10.2021)
[5] https://github.com/grafana/grafana (Zugriff am 15.10.2021)
[6] https://github.com/elastic/kibana (Zugriff am 15.10.2021)
[7] SSPL und Elastic Lizenz, es sei denn, der Header gibt eine andere Lizenz an
[8] https://github.com/opensearch-project/OpenSearch (Zugriff am 15.10.2021)

Unterstützung der mehrdimensionalen Datenerfassung, die durch Metriknamen und Schlüssel-Wert-Paare identifiziert werden kann. Mithilfe von PromQL, einer flexiblen Abfragesprache, wird diese Dimensionalität zur Datenabfrage genutzt, was eine vereinfachte Filterung, Gruppierung und einfacheres Matching ermöglicht. Somit legt Prometheus großen Wert auf Zuverlässigkeit, d. h., Statistiken können jederzeit eingesehen werden, auch unter Ausfallbedingungen. Es existieren außerdem keine Abhängigkeiten vom verteilten Speicher. Die Zeitreihenerfassung erfolgt über ein Pullmodell über HTTP und die Push-Zeitreihen werden über ein Zwischengateway unterstützt. Nicht zu empfehlen ist Prometheus jedoch in Systemen, die eine 100-prozentige Genauigkeit voraussetzen, beispielsweise bei einer Abrechnung pro Anfrage oder in medizinischen Systemen. Hier sind die gesammelten Daten vermutlich nicht immer vollständig und detailliert genug sind, um die Ausfallsicherheit garantieren zu können. Zur grafischen Visualisierung besitzt Prometheus zwar ein eigenes vereinfachtes Dashboard, jedoch werden die Metriken meist mit Grafana kombiniert. Hierfür bietet Prometheus eine ausführliche Dokumentation an. Ebenso kann es über Elasticsearch auch an Kibana angebunden werden (Prometheus, 2021).

3.1.2 Logstash

Logstash ist eine Open-Source-Datenerfassungs-Engine mit Echtzeit-Pipelining-Funktionen. Ihr Fokus liegt auf Log-Daten. Hierbei kommen vor allem Plugins zum Einsatz, die bereits existieren (über 200) oder für individuelle Zwecke selbst erstellt werden können (Elastic, 2021). Dabei besitzt jede Pipeline drei grundlegende Elemente: Input-Plugin, Output-Plugin und Filter. Diese werden zur Erfassung von Daten, zur Umwandlung der Daten in eine gewünschte Form und zur Ausgabe an ein beliebiges Ziel benötigt. Es kann sich um verschiedene mit jeweils unterschiedlichen Formaten handeln, beispielsweise Log-Dateien, Cloud-Services oder Datenbanken. Durch diesen Aufbau lässt sich Logstash für weitere zukünftige Eingabequellen in einem Projekt horizontal erweitern. Allerdings ist das Umwandeln von Strings, wie Logs, in numerische Werte ressourcenintensiv, da die Umwandlung in Echtzeit geschieht. Persistente Warteschlangen sorgen dafür, dass eingelesene Daten mindestens einmal zuverlässig zugestellt werden (Luber & Litzel, 2020). Üblicherweise wird Logstash zusammen mit weiteren Tools von Elastic verwendet und bildet zusammen mit Elasticsearch und Kibana den sogenannten ELK-Stack. Dabei ist Elasticsearch als eine Suchmaschine zu verstehen, die zur Indexierung und Durchsuchung der Daten aus JSON-Dokumenten eine REST-API zur Verfügung stellt (Elastic, 2021). Kibana stellt die zugehörige Visualisierungsplattform bereit, die unter 3.2.2 näher erklärt wird. Allerdings können aufgrund der Plugins auch weitere Tools verwendet werden, wie Zabbix, Graphite oder Datadog zum Monitoring oder Watcher für das Alerting.

3.1.3 Fluentd

Fluentd ist eine Open-Source-Datenerfassungssoftware, ursprünglich entwickelt von Treasure Data. Hierbei wird auf einen Unified Logging Layer gesetzt, der die Datenquellen von Backend-Systemen entkoppelt und vereinheitlicht. Erreicht wird dies, indem Daten möglichst oft als JSON-Datei strukturiert werden, mit denen das Sammeln, Filtern, Puffern und Ausgeben von Protokollen über mehrere Quellen und Ziele hinweg ermöglicht wird. Somit setzt auch Fluentd auf ein Plugin-System, das durch die Community stetig erweitert wird, und bietet dadurch ebenso eine horizontale Erweiterung an (Fluentd Project, 2021). Wird der Unified Logging Layer als pushbasiertes System implementiert, so muss dieser Layer auch einen wiederholbaren Datentransfer unterstützen. Wird er hingegen pullbasiert implementiert, muss der Protokollverbraucher eine erfolgreiche Datenübertragung sicherstellen, zum Beispiel über Offsets (Tamura, 2014). Aufgrund der großen Auswahl an Plugins unterstützt Fluentd eine Vielzahl weiterer Tools. So können beispielsweise Elasticsearch als Suchmaschine, Kibana oder Grafana als Visualisierung und Nagios als Alerting-System genutzt werden.

3.2 Monitoring

3.2.1 Grafana

Grafana ist eine Open-Source-Visualisierungssoftware, die 2014 von Torkel Ödegaard umgesetzt wurde. Angefangen hat das Projekt aus einem Fork von Kibana und führte eine Unterstützung für Metriken ein, die zu dieser Zeit von Kibana noch nicht gewährleistet werden konnte. So wurde der Fokus hier vor allem auf die Analyse und die Visualisierung von Metriken wie CPU, Arbeitsspeicher oder I/O-Auslastung gesetzt. Zudem bietet Grafana eine Integration von Graphite, Prometheus, InfluxDB, PostgreSQL und Elasticsearch an, sowie zusätzliche Datenquellen als Plugins. Dabei hat jede Datenquelle einen eigenen Query-Editor, der an Features und Funktionen angepasst ist. Weiterhin besitzt Grafana eine eigene Alerting-Engine. Die Visualisierung selbst basiert auf einer Vielzahl von Elementen und Anpassungsmöglichkeiten, um Daten beliebig anzuzeigen (Grafana Labs, 2021). Eine beispielhafte Darstellung eines Grafana-Dashboards in Bezug auf die Kubernetes-Kapazität zeigt Abbildung 3.3.

Abbildung 3.3: Beispiel eines konfigurierten Grafana-Dashboards (Grafana Labs, 2021)

3.2.2 Kibana

Kibana stellt die Visualisierungsplattform des bekannten ELK-Stacks dar. Im Gegensatz zu Grafana liegt der Fokus hier vor allem auf dem Analysieren und dem Abfragen von Log-Daten. Allerdings können in Kibana heutzutage auch Metriken angezeigt werden. Eine Besonderheit von Kibana ist allerdings, dass es darauf ausgelegt ist, ausschließlich mit Elasticsearch zu arbeiten, und somit keine weiteren Arten von Datenquellen unterstützt (Elastic, 2021). Seit der Version 7.11.0, am 10. Februar erschienen, verfügt auch Kibana über ein integriertes Alerting-System (Marten & Smith, 2021). Im Bereich der Visualisierung bietet Kibana eine Vielzahl an Anpassungsoptionen, die durch die Community stetig erweitert werden. Abbildung 3.4 stellt exemplarisch ein Kibana-Dashboard dar.

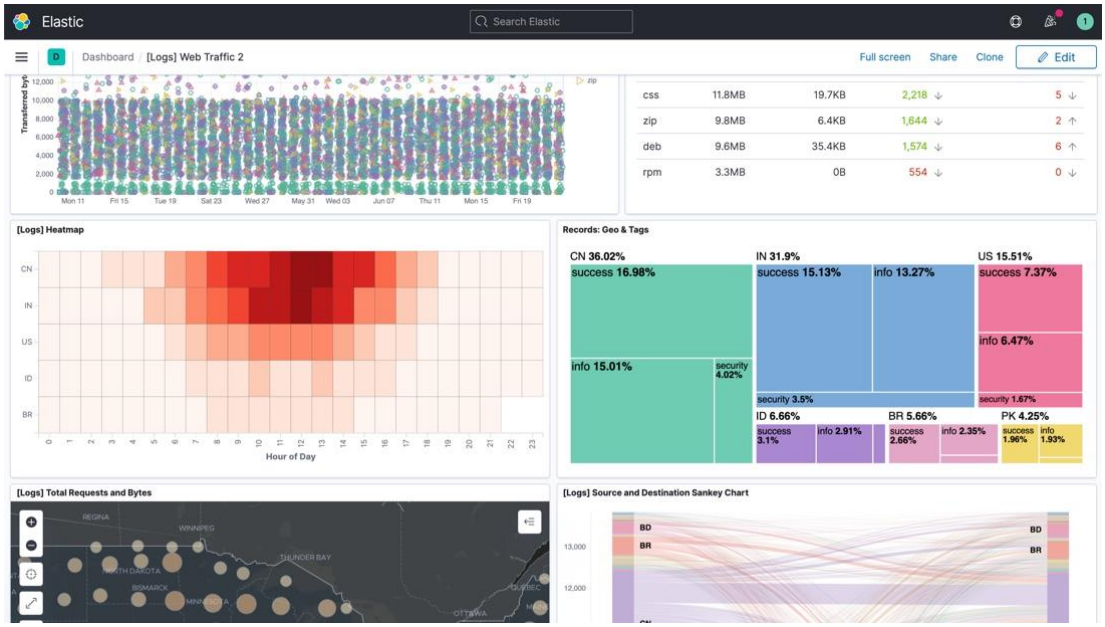

Abbildung 3.4: Beispiel eines konfigurierten Kibana-Dashboards (Elastic, 2021)

3.2.3 OpenSearch

OpenSearch und OpenSearch Dashboards sind aus einem Fork von Elasticsearch beziehungsweise Kibana entstanden, der von Amazon Web Services (AWS) durchgeführt wurde, weil sich Elastic Anfang 2021 dazu entschlossen hatte, die Software Elasticsearch und Kibana unter neuer Lizenz zu betreiben. Somit basiert der Fork von Elasticsearch 7.10.2 als auch von Kibana 7.10.2, denn diese war die letzte Fassung unter der Apache-Lizenz 2.0. Dadurch existieren in beiden Versionen ähnliche Features. Sie werden jedoch unabhängig voneinander entwickelt, wie in Abbildung 3.2 zu erkennen ist. Die stetige Entwicklung und Wartung wurde von AWS zugesichert. Zu den bereits vorhandenen neuen Funktionen zählen unter anderem die Trace[9]-Analyse, die Warnfunktion und eine SQL-Abfragesyntax. Außerdem bietet OpenSearch ebenso ein Plugin[10] an, um Logstash anzubinden. Hierdurch wird unter der Apache-Lizenz 2.0 ELK-Stack erreicht (Amazon Web Services, 2021). Abbildung 3.5 zeigt einen Screenshot aus OpenSearch Dashboards mit offiziell zur Verfügung gestellten Beispieldaten (hier: Web Traffic).

[9] Ein Trace stellt ein „Protokoll über den Ablauf eines Programms“ dar (Dudenredaktion, 2021)
[10] Ein Plugin stellt ein Zusatzmodul oder ein Zusatzprogramm, welches eine Anwendung um weitere Funktionen ergänzt

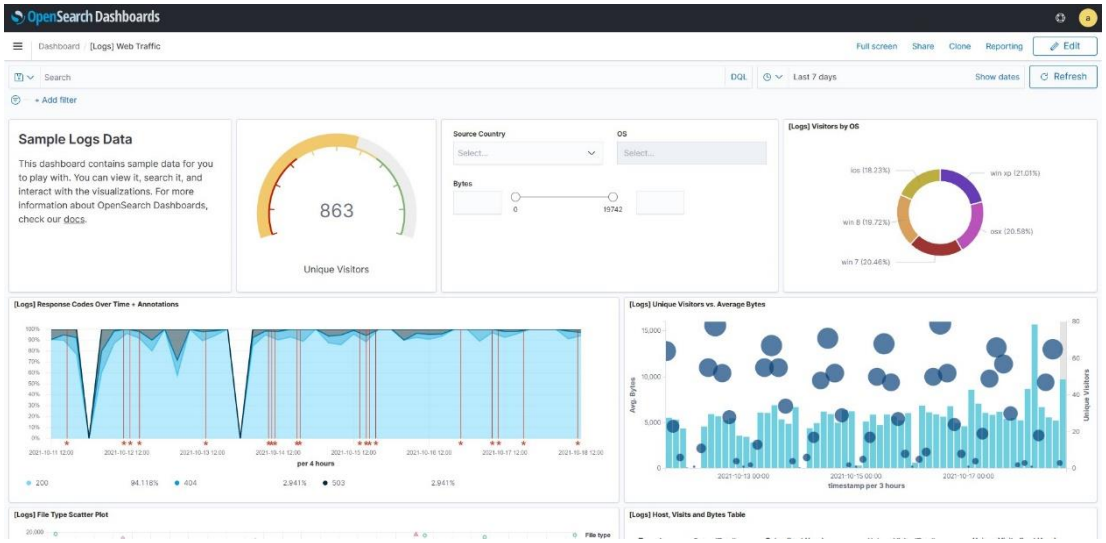

Abbildung 3.5: Beispiel eines konfigurierten OpenSearch-Dashboards

3.3 Wahl der Softwarekette

Für den weiteren Verlauf dieser Arbeit wurde die Softwarekette bestehend aus Logstash, OpenSearch und dem zugehörigen OpenSearch Dashboards gewählt. Hierfür spricht, abgesehen von den mit dem ODS kompatiblen Apache-2.0-Lizenzen, auch die trotz des kürzlich entstandenen Forks aktive Community. Die Basisfunktionen sind bereits in Version 7.10.2 von Elasticsearch und Kibana zahlreich vorhanden, zudem wird OpenSearch durch AWS fortdauernd aktualisiert. Mithilfe eines bereits verfügbaren Plugins können auch die Vorteile von Logstash genutzt werden. So entsteht eine perfekt zusammenarbeitende Programmkette, die beispielsweise eine Volltextsuche ermöglicht. Das Dashboard kann durch die große Anzahl an zur Verfügung stehenden Widgets nach Belieben angepasst werden und besitzt eine leicht zugängliche Benutzeroberfläche. Im folgenden Kapitel werden die einzelnen Anforderungen an diese Werkzeugkette konkretisiert.

4 Anforderungen

4.1 Funktionale Anforderungen

4.1.1 Logging

A1: Bestandteile einer Log-Information

Log-Informationen sollen Statuscode, Fehlermeldung, Servicenamen, Benutzer-ID, IP-Adresse, Correlation ID, Methodennamen, Call Stack und einen Zeitstempel beinhalten.

A2: Anzeige der Log-Informationen auf der Weboberfläche

Die Log-Informationen (siehe A1) können vollständig auf der Weboberfläche des Dashboards gefunden und eingesehen werden.

A3: Definition eines standardisierten Log-Formats

Unterschiedliche Dienste einer Microservice-Anwendung besitzen meist ungleiche Log-Formate. Diese sollen vereinheitlicht auf dem Dashboard ausgegeben werden.

A4: Farbliche Hervorhebung im Fehlerfall

Im Fehlerfall eines beliebigen Services wird diese Log-Information farblich auf dem Dashboard angezeigt.

A5: Rückverfolgung einzelner Anfragen

Einzelne Anfragen, die mehrere Services durchlaufen, sollen vollständig und nachvollziehbar wiedergegeben werden, sodass ihr Pfad zurückverfolgt werden kann.

A6: Durchsuchbarkeit der Logs

Log-Informationen können mithilfe einer Volltextsuche auf dem Dashboard durchsucht werden.

A7: Filtereinstellungen

Das Dashboard bietet die Möglichkeit, die vorhandenen Log-Informationen nach Kriterien zu filtern. Filterkriterien sind der Status-Code und die verschiedenen vorhandenen Microservices.

A8: Vorgabe eines Service-Templates

Ein wiederverwendbares Skript soll bereitgestellt werden, um zukünftig neue Microservices in das Logging-System zu integrieren.

4.1.2 Monitoring

A9: Anzeige eines zentralen Monitoring-Dashboards

Das Dashboard soll einen zentralen Ort besitzen, auf dem die einzelnen Services angezeigt werden.

A10: Health-Checks

Ergänzend zu Anforderung 9 sollen zu jedem Dienst in Echtzeit Health-Checks angezeigt und bei einem Ausfall farblich hervorgehoben werden.

A11: Anzeige von Metriken

Darüber hinaus sollen auf dem Dashboard Metriken zur Anwendung selbst angezeigt werden. Diese Metriken sollen CPU-Auslastung, Speichernutzung, eingehende Anfragen und Fehlerraten erfassen.

4.2 Nichtfunktionale Anforderungen

A12: Usability

Die Benutzeroberfläche soll verständlich, übersichtlich und benutzerfreundlich sein.

A13: Fehlertoleranz

Noch unbekannte Log-Formate sollen nicht zu einem Ausfall oder einer Blockade führen.

A14: Verfügbarkeit

Die Services sollen möglichst ausfallsicher sein. Trotz einer Wartung soll weiterhin die Möglichkeit bestehen, Log-Informationen zu aggregieren und anzuzeigen.

5 Konzeption

Dieses Kapitel beschreibt die Funktionsweise und den generellen Aufbau der verwendeten Programme, um die Observability aus Abschnitt 2.2 und die daraus resultierende Zielarchitektur für den ODS zu erreichen.

5.1 Logstash

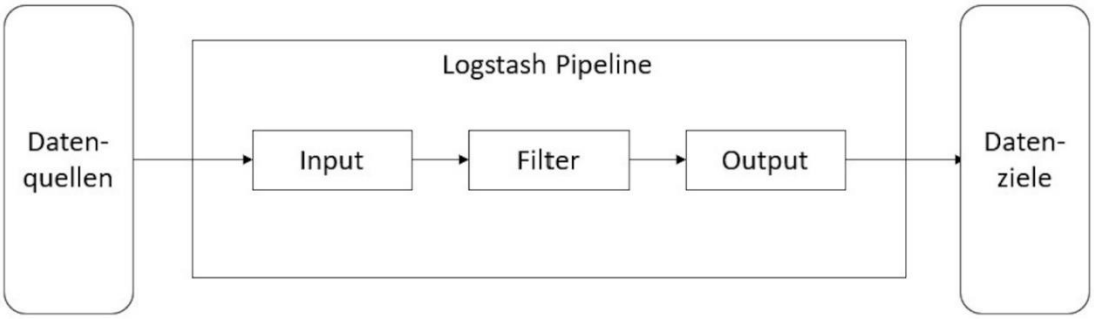

Abbildung 5.1: Allgemeiner Aufbau von Logstash

Wie in Kapitel 3.1.2 erwähnt, handelt es sich bei Logstash um eine Echtzeit-Datenerfassungs-Engine, die dabei helfen soll, Observability für den ODS umzusetzen. Denn einer der wesentlichen Prozesse hierbei ist das Konvertieren von unstrukturierten Log-Informationen in eine einheitliche Protokollierung, die die Suche und die Analyse von relevanten Daten erleichtert. Denn die meisten Analysemaschinen wie OpenSearch, die in Abschnitt 5.3 näher erläutert wird, sind davon abhängig, dass Daten einer Struktur folgen.

Dabei bestehen Log-Informationen im Wesentlichen aus zwei Hauptbestandteilen: zum einen aus einem Zeitstempel, der den Zeitpunkt angibt, zu dem ein Ereignis eingetreten ist, und zum anderen aus den Daten, die eine oder mehrere Informationen über ein konkretes Ereignis enthalten.

Abbildung 5.1 zeigt den prinzipiellen Aufbau von Logstash. In der offiziellen OpenSearch-Dokumentation wird die Zusammensetzung näher beschrieben:

Protokollierte Ereignisse werden aus einer oder mehreren Datenquellen an Logstash gesendet. Bereits in diesem Prozess entsteht eine vorteilhafte Entkopplung der Ereignisverarbeitung in den einzelnen Services der Anwendung. Nach der Übermittlung werden keine weiteren Informationen benötigt, was anschließend mit den Ereignissen geschieht (Amazon Web Services & Django Software Foundation, 2021, s. Clients and Tools, Logstash).

Daraufhin durchläuft jedes dieser Ereignisse eine vorkonfigurierte Pipeline, die aus drei Teilen besteht.

Beim *Input* können Ereignisse, wie Protokolle, von mehreren Quellen gleichzeitig empfangen werden, da jede Eingabe in einem eigenen Thread läuft, um eine gegenseitige Blockade durch gleichzeitig eingehende Ereignisse zu vermeiden. Hier unterstützt Logstash eine Reihe von Eingabe-Plugins, u. a. für TCP/UDP, Dateien, Syslogs, stdin und HTTP. Danach werden die empfangenen Ereignisse an eine Arbeitswarteschlange weitergeleitet. Dabei wendet jeder Arbeiter die vorgegebenen Filter innerhalb seines eigenen Threads an, um auch bei der Verarbeitung der Ereignisse eine Nebenläufigkeit zu garantieren.

Um einen optimalen Durchsatz zu schaffen, verarbeitet ein solcher Pipeline-Arbeiter die Ereignisse aus der Arbeitswarteschlange in sogenannten Batches. Mithilfe von zwei konfigurierbaren Parametern können die Größe und die Verzögerung angepasst werden. Der hieraus gezogene Vorteil ist, dass nicht jedes Ereignis einzeln mit jedem Filter bearbeitet wird, sondern alle Ereignisse gesammelt in einer gewünschten Größe. Der zweite Parameter ist ein zeitlich limitierender Faktor und sorgt, falls diese Größe in einer gewissen Zeit nicht erreicht wird, dafür, dass die bis zu dieser Vorgabe gesammelten Ereignisse dennoch zusammen verarbeitet werden (Amazon Web Services & Django Software Foundation, 2021, s. Clients and Tools, Logstash, Logstash execution model).

Weiterhin lässt sich die Anzahl der Arbeiter einstellen. Allerdings führt eine feste Anzahl in den meisten Fällen zu Leistungseinbrüchen, sofern mithilfe unterschiedlicher Maschinen getestet und gearbeitet wird. Die standardmäßige Voreinstellung ist somit meist die beste Wahl. Diese ermittelt die Anzahl anhand der Prozessorkerne der Instanz (Amazon Web Services & Django Software Foundation, 2021, s. Clients and Tools, Logstash, Logstash execution model).

Um strukturierte Daten schließlich zu formen, wird der *Filter* verwendet. Dieser Teil der Logstash-Pipeline besteht aus einem oder mehreren Filter-Plugins, die dabei helfen, die Informationen aus dem Protokoll zu analysieren und gegebenenfalls zu bereichern. Dabei ist zu beachten, dass diese Filter jeweils sequenziell ausgeführt werden, sodass festgelegt wird, in welcher Reihenfolge die Daten transformiert

werden. Die am häufigsten verwendeten Filter sind *mutate* und *grok*. Ersterer verändert die Datentypen eines Feldes und Letzterer formatiert unstrukturierte Daten in Felder, indem er eine Zeile mit einem angegebenen Muster abgleicht, das auf einem regulären Ausdruck basiert, und sie dann bestimmten Feldern zuordnet (Amazon Web Services & Django Software Foundation, 2021, s. Clients and Tools, Logstash).

Gefilterte Ereignisse werden abschließend mittels des *Outputs* an eines oder mehrere Datenziele gesendet. Auch hierfür bietet Logstash eine große Anzahl von Ausgabe-Plugins an, wie OpenSearch, TCP/UDP, E-Mails, stdout und HTTP (Amazon Web Services & Django Software Foundation, 2021, s. Clients and Tools, Logstash).

5.2 Filebeat

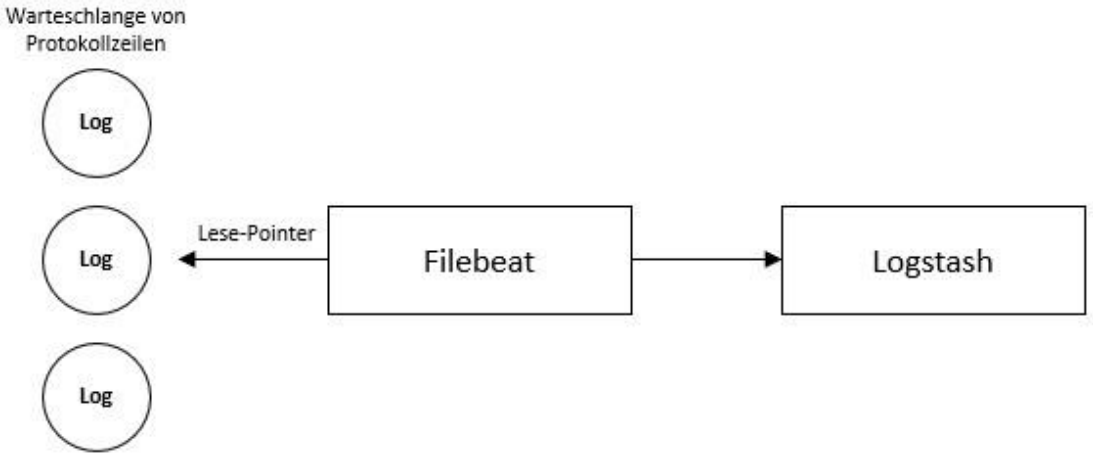

Abbildung 5.2: Funktionsweise von Filebeat anhand einer Logdatei

In Ergänzung zu Logstash wird in den meisten Fällen zusätzlich ein Datenversender genutzt, der mehrere Protokolldateien beobachtet und neue Datenzeilen zur Verarbeitung an Logstash übermittelt, wenn diese hinzukommen. Die Erweiterung durch Filebeat eignet sich vor allem, da Logstash lediglich Zeile für Zeile liest, diese Daten transformiert und anschließend an OpenSearch sendet. Dieser Prozess könnte bei mehreren Microservices zu einer deutlich langsameren Verarbeitung führen. Aus diesem Grund wurde Filebeat, das damit den ELK-Stack vervollständigt. Allerdings ist ebenfalls eine Version unter der Apache-Lizenz 2.0 erhältlich, die somit auch mit dem OpenSearch-Stapel kombinierbar ist.

Zusätzlich nutzt Filebeat ein Protokoll, das bei einer vollen Pipeline von Logstash reagiert. Sollte die Verarbeitungsgeschwindigkeit der vorliegenden Daten nicht ausreichen, so wirkt Filebeat dem mit einer reduzierten Lesegeschwindigkeit entgegen. Bei freien Kapazitäten kehrt Filebeat wieder zur Normalgeschwindigkeit zurück (Elastic, 2021, s. Schluss mit überlasteten Pipelines).

5.3 OpenSearch und OpenSearch Dashboards

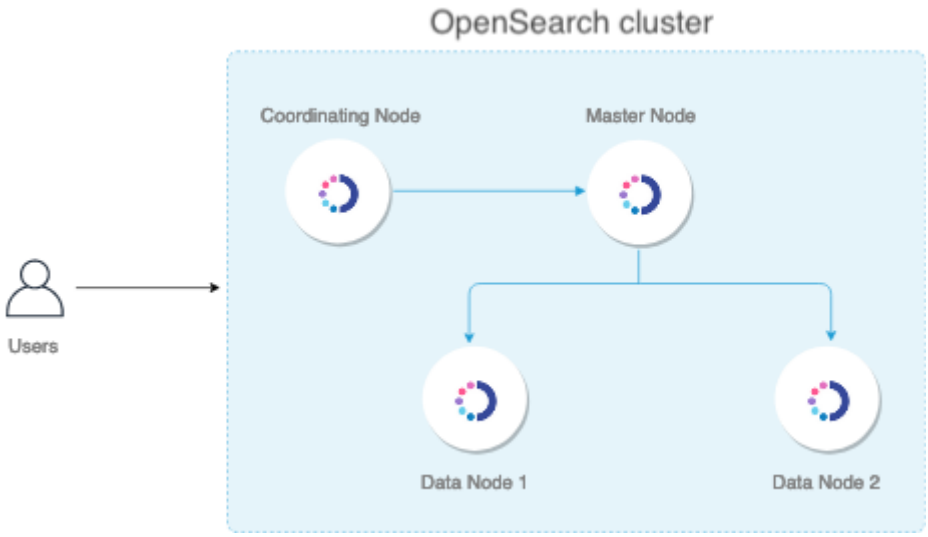

Abbildung 5.3: Standard-Clusterbildung von OpenSearch (Amazon Web Services & Django Software Foundation, 2021, s. Cluster formation)

OpenSearch ist, wie vor seinem Fork aus Elasticsearch, eine verteilte Such- und Analysemaschine, allerdings mit neuem Namen und neuen Funktionen, jedoch mit alter Apache-Lizenz 2.0. Beim Aufbau von OpenSearch wird der Fokus in der offiziellen Dokumentation zunächst auf die sogenannten *Cluster* gesetzt. Abbildung 5.3 zeigt eine grundlegende Clusterbildung. Es wird deutlich, dass jedes Cluster aus mindestens einem Knoten besteht.

Diese Knoten können verschiedenen Typen zugeordnet werden, die jeweils eine andere Aufgabe abdecken. So sind die *Master*-Knoten für den Gesamtbetrieb zuständig. Sie überprüfen den Zustand aller Knoten und verwalten die Indizes. Die *Coordinating*-Knoten nehmen Clientanforderungen an, sammeln und aggregieren die Ergebnisse zu einem Endergebnis und senden dieses zurück. Der Typ *Data* ist der eigentliche Arbeiter-Knoten, der alle datenbezogenen Aktionen durchführt. Bei dem letzten Knoten handelt es sich um *Ingest*, der die Daten transformiert, bevor diese im Index gespeichert werden. Standardmäßig fungiert jeder Knoten gleichzeitig als all diese Typen (Amazon Web Services & Django Software Foundation, 2021, s. Cluster formation).

Neben zahlreichen Plugins und Ergänzungen an OpenSearch sind im Hinblick auf den ODS und die Erfüllung der Anforderungen vor allem das Konzept der Volltextsuche sowie die Überwachung wesentlicher Metriken von Bedeutung. Daher beschränkt sich diese Arbeit auf die Plugins, die zur Erfüllung dieser Konzepte dienen.

Um Daten durchsuchen zu können, muss zuvor ein *Index* angelegt werden. Suchmaschinen, wie OpenSearch, organisieren die Daten intern mittels eines solchen Index, wodurch ein schnelles Wiederauffinden ermöglicht wird. Innerhalb eines Index erfolgt die Identifikation einzelner Dokumente mithilfe einer eindeutigen ID. Durch diese Funktionsweise kann eine schnelle Volltextsuche durchgeführt werden (Amazon Web Services & Django Software Foundation, 2021, s. Index data).

Im Bereich der Analyse und der Aggregation von Metriken zeigen OpenSearch und OpenSearch Dashboards hingegen ihre Schwächen im Vergleich zur Kombination von Prometheus und Grafana. Denn während OpenSearch den Fokus auf die Analyse und die schnelle Durchsuchung legt, wird bei den beiden Vergleichsprogrammen mehr Wert auf die Datenerfassung und die Visualisierung von numerischen Metriken gelegt. Allerdings können diese, aufgrund ihrer Lizenz, nicht mit in die Anwendung des ODS aufgenommen werden. Jedoch schafft ein weiteres Plugin Abhilfe. Der *Performance Analyzer* bietet eine REST-API an, mit der Leistungsmetriken aus dem erstellten Cluster abgefragt und weiter mithilfe von *PerfTop*, der Standard-Befehlszeilenschnittstelle, visuell dargestellt werden können (Amazon Web Services & Django Software Foundation, 2021, s. Monitoring Plugins, Performance Analyzer).

5.4 Zielarchitektur

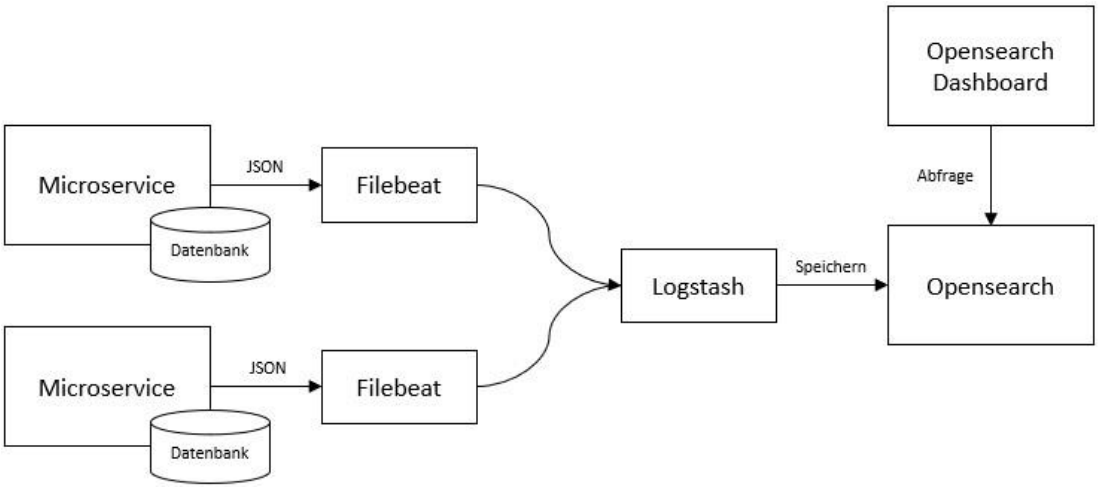

Abbildung 5.4: Genereller Aufbau des OpenSearch-Stapels innerhalb von ODS

Die bisher vorgestellten Programme sollen dazu dienen, die Architektur des ODS dahingegen zu erweitern, dass die drei Säulen der Observability aus Kapitel 2.2 abgedeckt werden. Damit soll ein Monitoring-System geschaffen werden, das die Entwickler dabei unterstützt, Fehlverhalten der Services schnell zu erkennen.

Im ODS befinden sich zum Zeitpunkt dieser Arbeit insgesamt sieben Microservices. Das Ziel wird es sein jeden dieser Dienste, sowohl im Frontend als auch im Backend, vollständig so umzuschreiben, dass jede Debug-Information in einer JSON-Datei ausgegeben wird. Dabei dient jeder einzelne Service auch als Index für OpenSearch, sodass hiernach vereinfacht gegliedert und gesucht werden kann. Hierfür soll eine statische Klasse pro verwendete Sprache eingeführt werden, die als Schnittstelle eine Funktion bereitstellt, mit der das Logging für die Microservices des ODS einheitlich ermöglicht werden sollen. Als Parameter sollen die Informationen aus Anforderung A1 übernommen werden.

Nachdem dieser Schritt umgesetzt wird und sämtliche notwendigen Informationen protokolliert werden, folgt die indirekte Weiterleitung dieser Protokolldatei an Filebeat. Die einzelnen Microservices sollen lediglich die JSON-Datei um Informationen erweitern und Filebeat mittels eines Lese-Pointers auf Veränderungen überwachen. Jede neue Zeile reiht sich somit in diesem Schritt in eine interne Warteschlange ein, die solange verzögert wird, bis sie an Logstash übermittelt werden kann.

Ist eine Weiterleitung an Logstash erfolgt, werden die Protokolle hier akzeptiert, nach ihrer Verwendung transformiert und schließlich an OpenSearch übertragen. OpenSearch speichert jede bearbeitete Log-Information in einem Index ab, wodurch eine Analyse und die Durchsuchung vereinfacht werden sollen. Zuletzt ermöglicht OpenSearch Dashboards in dieser Softwarekette die Visualisierung mithilfe einer Weboberfläche. Hier können Daten verwaltet und mittels verschiedener Widgets dargestellt werden. Der prinzipielle Aufbau des OpenSearch-Stapels innerhalb von ODS wird in Abbildung 5.4 dargestellt.

6 Umsetzung

In diesem Abschnitt geht es um die beispielhafte Implementierung der Softwarekette aus dem vorherigen Kapitel, die durchgeführt wird, um die Zielarchitektur und das zugehörige Konzept der Observability zu erreichen. Dabei wird der Prozess der Entwicklung schrittweise dokumentiert und das Vorgehen erläutert.

6.1 Vorbereitung

Eine gute Vorbereitung bringt großen Nutzen und sorgt für einen vereinfachten Einstieg in die Implementierung. Im Falle von ODS wird diese Microservice-Anwendung mittels *Docker* bereitgestellt.

Jaroslaw Krochmalski bietet in seinem Buch *Developing with Docker* eine kurze und passende Beschreibung von Docker. Laut ihm kann es als eine Art Werkzeug für Entwickler betrachtet werden, das bei Problemen wie dem Installieren, dem Bereitstellen, dem Verwalten und dem Ausführen der Software hilft. Denn die Grundidee von Docker besteht darin, eine Anwendung mit all ihren Abhängigkeiten und dem eigentlichen Code in einem vollständigen Dateisystem zusammenzufassen, wodurch garantiert wird, dass das Programm gleichmäßig ausgeführt wird und läuft, unabhängig von der Umgebung, in der es letztendlich bereitgestellt wird. Dieses isolierte Dateisystem wird Docker-Container genannt. Ein Container ist keine virtuelle Maschine im gängigen Sinne, sondern steht für die Virtualisierung des Betriebssystems. Das bedeutet, ein Docker-Bild läuft innerhalb desselben Betriebssystemkernels statt auf einem unabhängigen Gastbetriebssystem, wie bei einer üblichen virtuellen Maschine. Ein Docker-Container hat sein eigenes Dateisystem und Umgebungsvariablen. Zudem sind die Container nicht nur vom zugrunde liegenden Betriebssystem, sondern auch voneinander isoliert. (Krochmalski, 2016, S. 7, 8).

Damit OpenSearch in Docker lokal in Betrieb genommen werden kann, braucht es in Docker zunächst eine Voreinstellung von mindestens vier Gigabyte Arbeitsspeicher. Abbildung 6.1 zeigt, wie diese Einstellung für Linux beziehungsweise für das Windows-Subsystem für Linux (WSL) durchgeführt werden kann.

```
1   # Linux
2   sudo sysctl -w vm.max_map_count=262144
3   sudo sysctl -p
4
5   # Windows (WSL)
6   wsl -d docker-desktop
7   sysctl -w vm.max_map_count=262144
```

Abbildung 6.1: Docker-Einstellung für vier Gigabyte Arbeitsspeicher

Nachdem die Voraussetzung erfüllt ist, um OpenSearch generell starten zu können, wird als nächstes eine YAML-Datei benötigt. Diese wird von Docker-Compose verwendet, um die Dienste der Anwendung zu konfigurieren. Compose stellt hierbei das Werkzeug zum Definieren und Ausführen von Docker-Anwendungen mit mehreren Containern dar. Anschließend werden mit einem einzigen Befehl alle Services aus der Konfiguration erstellt und gestartet (Docker Inc., 2021).

Eine solche YAML-Datei für das ODS ist in Anhang 2 dargestellt. Zu Beginn wurde die Beispiel-Datei aus der offiziellen Dokumentation verwendet, die insgesamt nur zwei Knoten beinhaltet. Hier wurden zur Erklärung einzelner Komponenten zusätzliche Kommentare ergänzt. Auch eine weitere Umgebungsvariable zur Integration von Logstash wurde eingefügt. Diese Datei war für die erste Implementierung ausreichend, besonders da die Knoten standardmäßig als alle Typen gleichzeitig fungieren. In Zukunft können weitere Knoten ergänzt sowie die Typen der Knoten verändert werden. Die Typänderung wurde als Kommentar (Zeile 9–12) in der beigefügten YAML-Datei ergänzt. Hier ist anzumerken, dass ein Knoten als koordinierender Knoten dient, wenn alle drei Typdefinitionen eines Knotens auf *false* gesetzt sind.

Abschließend konnte OpenSearch mittels des Befehls *docker-compose up* im Terminal gestartet und die Lauffähigkeit überprüft werden. Abbildung 6.2 zeigt einen erfolgreichen Health-Check des Services. Das zugehörige Dashboard wird in einem beliebigen Browser unter *localhost:5601* erreicht und angezeigt.

```
[opensearch@26864f5700a1 ~]$ curl -k -u "admin:admin" -X GET 'https://localhost:9200/_cluster/health?pretty'
{
  "cluster_name" : "opensearch-cluster",
  "status" : "green",
  "timed_out" : false,
  "number_of_nodes" : 2,
  "number_of_data_nodes" : 2,
  "discovered_master" : true,
  "active_primary_shards" : 3,
  "active_shards" : 6,
  "relocating_shards" : 0,
  "initializing_shards" : 0,
  "unassigned_shards" : 0,
  "delayed_unassigned_shards" : 0,
  "number_of_pending_tasks" : 0,
  "number_of_in_flight_fetch" : 0,
  "task_max_waiting_in_queue_millis" : 0,
  "active_shards_percent_as_number" : 100.0
}
```

Abbildung 6.2: Ergebnis eines erfolgreichen Health-Checks von OpenSearch

6.2 Integration von Logstash

Nachdem OpenSearch und OpenSearch Dashboards erfolgreich in Betrieb genommen wurden, fehlten jetzt noch die eigentlichen Protokolldaten, die durch OpenSearch aggregiert und visualisiert werden sollten. Die notwendige Infrastruktur für einen Datenfluss sowie die Verarbeitung dieser Daten, um eine einheitliche Struktur zu schaffen, wurden durch Logstash ermöglicht.

Für die Integration weiterer Open-Source-Software musste zusätzlich die Kompatibilität beachtet werden. Hierfür wird in der offiziellen Dokumentation eine Matrix angeboten, die eine Limitierung für Logstash aufweist, denn die aktuelle Version[11] kann nicht genutzt werden (Amazon Web Services & Django Software Foundation, 2021, s. Clients and Tools, Agents and ingestion tools). Aus diesem Grund wurde im Anschluss die OSS-Version 7.13.4 verwendet. Damit allerdings ein reibungsloses Zusammenspiel mit OpenSearch funktioniert, musste eine weitere Umgebungsvariable für die Kompatibilität in das Cluster eingeführt werden. Diese ist in Anhang 2 in Zeile 16 zu finden. Zudem kann diese YAML-Datei erweitert werden, um Logstash einzubinden.

[11] Logstash OSS Version 7.15.2 (Stand 15.11.2021)

```
68    opensearch-logstash:
69      image: opensearchproject/logstash-oss-with-opensearch-output-plugin:7.13.4
70      container_name: opensearch-logstash
71      volumes:
72        - logstash:/usr/share/logstash
73      networks:
74        - opensearch-net
75
76  volumes:
77    opensearch-data1: # Auskommentieren, um persistente Daten zu gewährleisten (s. oben)
78    opensearch-data2:
79    logstash:
```

Abbildung 6.3: Erweiterung von *docker-compose.yml* um Logstash

Nachdem die Installation erfolgt war, konnte die Pipeline von Logstash konfiguriert werden. In Kapitel 5.1 wurde bereits der generelle Aufbau einer Pipeline dargestellt und erläutert. Dazu wurde nun nach dem Konzept eine mögliche Konfiguration der Pipeline erstellt, mit dem die Daten aus einer JSON-Datei ausgelesen und das gesamte Protokoll an OpenSearch übertragen wird.

```
1   input {
2     file {
3       path => "/tmp/test-input.json"
4       type => "json"
5       start_position => "beginning"
6     }
7   }
8
9   filter {
10    json {
11      source => "message"
12    }
13  }
14
15  output {
16    opensearch {
17      hosts => ["https://opensearch-node1:9200","https://opensearch-node2:9200"]
18      index => "opensearch-logstash-docker-%{+YYYY.MM.dd}"
19      user => "admin"
20      password => "admin"
21      ssl => true
22      ssl_certificate_verification => false
23    }
24    file {
25      path => "/tmp/test-output.txt"
26    }
27  }
```

Abbildung 6.4: Mögliche Logstash-Pipeline-Konfiguration

Da hier noch kein Fokus auf korrekte Zertifikats-Handhabung gelegt wurde, wurde hier die Verifikation auf *false* gesetzt. Außerdem sorgt *start_position* zusammen mit dem Argument *config.reload.automatic=true* aus Abbildung 6.4 für ein automatisches erneutes Laden der Pipeline sowie der Eingabedatei, unabhängig von Filebeat. Später soll dieses Vorgehen durch Filebeat ersetzt werden, sodass dieser Prozess zuverlässiger und vereinfacht erweiterbar wird und die mögliche Überlastung von Logstash präventiv durch die Lesegeschwindigkeit verhindert werden kann. Darüber hinaus wurde hier vorerst zu Testzwecken eine Beispieldatei im JSON-Format als Eingabe genutzt. Neben der Ausgabe nach OpenSearch wurde diese noch zusätzlich als Textdatei ausgegeben. Anzumerken ist, dass Logstash nur absolute und keine relativen Pfade akzeptiert.

Die Eingabe aus Abbildung 6.4 verwendet die Information

```
{ "message": "Dies ist eine Testnachricht", "status": "Erfolg" }
```

im JSON-Format. Die hieraus resultierende Ausgabe kann in der nächsten Abbildung betrachtet werden. Obwohl das Ergebnis in einer Zeile ausgegeben wurde, wird hier zur Übersicht in mehrere Zeilen getrennt. Ebenso ist der Index in OpenSearch Dashboards unter dem Reiter *Stack Management* und weiter *Index Pattern* zu sehen.


```
{
  "@version":"1",
  "status":"Erfolg",
  "@timestamp":"2021-11-27T23:32:16.316Z",
  "message":"Dies ist eine Testnachricht",
  "host":"cae6e1b2a86a"
}
```


Abbildung 6.5: Ausgabe der *test-output.txt*-Datei

6.3 Bisheriges Logging in ODS

Zur Erfüllung des Konzepts der Observability kann es hilfreich sein, das bisherige Logging jedes Services des ODS zu betrachten. Beim Blick auf das Repository werden in GitHub die im Projekt verwendeten Sprachen zusammengefasst. Die vier hauptsächlich verwendeten Sprachen sind: TypeScript, JavaScript, Java und Vue. Bei genauerer Betrachtung des Codes wird deutlich, dass überwiegend über *console.log* beziehungsweise *console.debug* protokolliert wird. Im Java-Teil wird auf die *Simple Logging Facade for Java* (SLF4J) gesetzt. Diese Fassade ist der populärste Versuch, eine Bibliothek, die das eigentliche Logging implementiert und durchführt, unter eine andere, einfachere Client-API zu setzen. Diese Logging-Bibliothek hat sich in der heutigen Zeit bewährt und etabliert (Ullenboom, 2014, s. 19.1.4 Die Simple Logging Facade).

```typescript
// Use the f = () => {} syntax to access this
consumeEvent = async (msg: ConsumeMessage | null): Promise<void> => {
  if (msg == null) {
    console.debug(
      'Received empty event when listening on datasource executions - doing nothing',
    );
    return;
  }
```

```java
import lombok.extern.slf4j.Slf4j;

public DataImport.MetaData trigger(Long id, RuntimeParameters runtimeParameters)
  throws DatasourceNotFoundException, AdapterException, IOException {
  try {
    return executeImport(id, runtimeParameters);
  } catch (Exception e) {
    log.error("Failed to execute import", e);
    publishImportFailure(id, e);
    throw e;
  }
}
```

Abbildung 6.6: Logging-Beispiele aus dem ODS in TypeScript[12] und Java[13]

6.4 Zu ergänzende Anbindung an das ODS

Zum Ende der Bearbeitungszeit dieser Bachelorarbeit konnte die direkte Anbindung an das ODS nicht fertiggestellt werden. Dies lag an der zu Beginn verwendeten, aber nicht kompatiblen, Softwarekette von Prometheus und Grafana. Aufgrund des fehlenden Features einer optimierten Volltextsuche und vor allem wegen einer nicht geeigneten Lizenz wurde ein Wechsel zu OpenSearch durchgeführt. Trotz des in diesem Jahr durchgeführten Forks und vorhandener Dokumentation wären mehr und ausführlichere Beispiele wünschenswert gewesen. Jedoch wird diese Werkzeugkette fortlaufend entwickelt und vermutlich weiter ergänzt. Aus diesem Grund soll im Folgenden der Weg zur Zielarchitektur beschrieben werden, um die Observability für das Projekt ODS abschließend zu ermöglichen.

Bis zu diesem Punkt sind alle benötigten Programme start- und lauffähig und benötigen letztlich nur noch die Daten aus dem ODS-Projekt. Wie bereits in Kapitel 5.4 beschrieben, sollten alle Ereignisse jedes Microservices das alte Logging-Verfahren ersetzen, sodass das Ereignis selbst zusammen mit seinen Metadaten und

[12] https://github.com/jvalue/ods/blob/main/pipeline/src/api/amqp/datasourceExecutionConsumer.ts

[13] https://github.com/jvalue/ods/blob/main/adapter/src/main/java/org/jvalue/ods/adapterservice/datasource/DatasourceManager.java

der Fehlermeldung in eine eigene JSON-Datei eingepflegt werden kann, statt wie bisher über die Konsole ausgegeben zu werden. Hierfür wäre es sinnvoll, eine statische Funktion zu implementieren, die als Schnittstelle dient und die einzelnen Klassen importieren und nutzen kann. Dies würde das Logging für zukünftige Erweiterungen vereinfachen und vor allem vereinheitlichen. Eine solche Funktion sollte für jede verwendete Sprache zentral definiert werden.

Nach Erreichen dieses Meilensteins fehlt die korrekte Pfadangabe von Filebeat zu jeder dieser eingepflegten JSON-Dateien. Zudem sollten in Logstash mehrere Pipelines angelegt werden, sodass jedes JSON-Protokoll in OpenSearch als eigener Index erstellt wird. Zwar können in einer Pipeline mehrere Ausgaben definiert werden, jedoch werden diese sequenziell abgearbeitet. Hierdurch würde jedes Protokoll in jedem Index gespeichert werden, womit der Vorteil einer Indexierung verloren gehen würde. Als Eingabe in Logstash kann hier das Plugin *beats* verwendet werden, wobei sichergestellt sein muss, dass Logstash auf den Filebeat-Port 5044 zugreifen kann.

```
1  input {
2      beats {
3          port => 5044
4      }
5  }
```

Abbildung 6.7: Filebeat-Eingabe-Plugin für Logstash

Um Filebeat mit Logstash zu verknüpfen, muss die Datei *filebeat.yml* im Konfigurationsordner angepasst werden. Neben der Definition der Eingabe und der Ausgabe sollte *setup.ilm.enabled* auf den Wert *false* gesetzt werden, denn die hier genutzte Version OSS-7.12.1 unterstützt zum momentanen Zeitpunkt noch keine Index-Lifecycle-Management-Richtlinien und würde mit einem wahren Wert somit nicht starten können. Genauso wie Logstash ist nämlich auch Filebeat in seiner Anbindung zu OpenSearch limitiert, die der Kompatibilitäts-Matrix entnommen werden kann (Amazon Web Services & Django Software Foundation, 2021, s. Clients and Tools, Agents and ingestion tools). Eine mögliche Konfiguration wird in Abbildung 6.8 gezeigt.

```
filebeat.inputs:
- type: json
  enabled: true
  paths:
    - /data/logs/json/*.json
filebeat.config.modules:
  path: ${path.config}/modules.d/*.yml
  reload.enabled: false
setup.template.settings:
  index.number_of_shards: 1
output.logstash:
  hosts: ["opensearch-logstash:5044"]
setup.ilm.enabled: false
setup.ilm.check_exists: false
```

Abbildung 6.8: Mögliche Konfiguration von *filebeat.yml*

Können aufgrund der unterschiedlichen verwendeten Sprachen in ODS keine einheitlichen Formate wiedergegeben werden, so kann mithilfe der Filter-Plugins in Logstash eine Strukturierung durchgeführt werden. Um den *grok*-Filter in Kapitel 5.1 zu testen, kann beispielsweise das Onlinetool *http://grokdebug.herokuapp.com/* genutzt werden. Hier kann ein vorher definierter Input auf ein beliebiges Muster überprüft werden.

Die Konfiguration von OpenSearch und OpenSearch Dashboards sollte ausreichend sein. Zuletzt kann lediglich ein Template für die auf dem Dashboard gewünschten Widgets sowie für häufig verwendete Suchanfragen erstellt werden, die sinnvoll für die Log-Informationen aus dem ODS wären.

Anzumerken ist, dass unter dem Reiter *Stack Management* das Indexmuster für ODS ausgewählt werden kann. Unter *Discover* kann der Filter passend zum Index eingestellt werden und die Logs werden angezeigt. Widgets können unter *Dashboard* betrachtet werden.

7 Auswertung

Die in Kapitel 4 definierten Anforderungen werden in diesem Kapitel auf ihre mögliche Erfüllung hin überprüft und es wird eine abschließende Zusammenfassung gegeben. Allerdings wird die Evaluierung hier aufgrund der in Kapitel 6.4 beschrieben Umstände dahingehend verändert, dass betrachtet wird, ob die Anforderungen mithilfe des OpenSearch-Stapels prinzipiell abgedeckt werden können.

7.1 Funktionale Anforderungen

A1: Bestandteile einer Log-Information

Die gewünschten Informationen sind abhängig von der jeweiligen Implementierung und können daher angepasst werden. Im Rahmen dieser Bachelorarbeit konnte dieser Teil der Implementierung nicht fertiggestellt werden. Wenn diese Informationen allerdings geloggt und in einer JSON-Datei gegliedert sind, wird OpenSearch diese verwenden und wiedergeben können.

A2: Anzeige der Log-Informationen auf der Weboberfläche

Die Weboberfläche wird mittels OpenSearch Dashboards dargestellt und ist mit den restlichen Services verknüpft. In Kapitel 6.2 wurde eine beispielhafte Log-Information über Logstash eingepflegt, die anschließend auf dem Dashboard angezeigt werden konnte.

A3: Definition eines standardisierten Log-Formats

Wie in Kapitel 6.4 beschrieben, wäre es sinnvoll, eine statische Funktion hierfür zu implementieren, die einen ersten Schritt in Richtung der Definition schafft. Sollte dies aufgrund der verschiedenen Sprachen nicht ausreichen, kann der Logstash-Filter *grok* hinzugezogen werden und Formate neu strukturieren.

A4: Farbliche Hervorhebung im Fehlerfall

Eine Möglichkeit für eine farbliche Hervorhebung im Text konnte innerhalb von OpenSearch nicht entdeckt werden. Somit entspricht die Version 1.1 von OpenSearch nicht der Anforderung.

A5: Rückverfolgung einzelner Anfragen

Diese Funktion konnte zwar nicht getestet werden, jedoch bietet OpenSearch ein Plugin als Lösung für dieses Problem an. Das Plugin Trace Analytics eignet sich dafür, OpenTelemtry-Daten zu sammeln und den Ereignisfluss zu visualisieren (Amazon Web Services & Django Software Foundation, 2021, s. Observability Plugins, Trace analytics).

A6: Durchsuchbarkeit der Logs

Ein großes Konzept und der Erfolg des ELK-Stacks ist unter anderem die Volltextsuche. Da OpenSearch hieraus entstanden ist, unterstützt auch OpenSearch die Volltextsuche mittels der angelegten Indizes.

A7: Filtereinstellungen

Das Dashboard bietet zahlreiche Optionen, um Filter hinzuzufügen und zu nutzen. So kann beispielsweise ein Feld für den Zeitstempel erzeugt und innerhalb eines Zeitraums grafisch dargestellt werden, sodass alle Logs eines gewünschten Services der letzten drei Stunden zu sehen sind.

A8: Vorgabe eines Service-Templates

Ein Template zur Anbindung neuer Services an das ODS ist nicht implementiert. Die Überlegungen, aus der Zielarchitektur eine neue Schnittstelle im Code anzubieten, um das Logging zu vereinheitlichen, könnten dieser Anforderung gerecht werden. Jedoch könnte dies aufgrund der unterschiedlichen Sprachen nicht ausreichen, sodass zusätzlich auf Logstash-Filter zurückgegriffen werden muss.

A9: Anzeige eines zentralen Monitoring-Dashboards

Ein zentraler Ort, dem sämtliche Log-Informationen entnommen werden können, ist gegeben. Sowohl visuelle Darstellungen als auch Log-Informationen in Textform können in OpenSearch Dashboards wiedergegeben werden.

A10: Health-Checks

Eine mögliche Überlegung könnte die Umsetzung dieses Features mithilfe neuer Endpunkte der einzelnen Microservices sein. Diese werden unter einem eigenen Index zusammengefasst und könnten so an OpenSearch übermittelt werden. Das Dashboard bietet zwar kein spezifisches Widget an, um einen Wahrheitswert zu generieren und anzuzeigen. Aber es könnte beispielsweise ein Zeitachsen-Widget verwendet werden, das den zeitlichen Verlauf der Erreichbarkeit darstellt. Sobald innerhalb eines Zeitfensters keine Log-Information von diesem Health-Endpunkt erfasst wird, kann ein Alert gesendet werden.

A11: Anzeige von Metriken

Hierfür bietet OpenSearch zwar den Performance Analyzer als Plugin an, allerdings muss auf PerfTop als visuelles Werkzeug zurückgegriffen werden. Eine Anzeige direkt über das Dashboard ist nicht möglich.

7.2 Nichtfunktionale Anforderungen

A12: Usability

Der Aufbau von OpenSearch selbst kann nicht angepasst werden. Allerdings wird von der Community beziehungsweise von AWS stetig das Angebot an Widgets erweitert. Außerdem können Vorlagen für Filter oder Suchanfragen angelegt werden, was zur Individualisierung beiträgt. Auch die Tatsache, dass OpenSearch aus dem bereits etablierten ELK-Stack entstanden ist, spricht für die Benutzerfreundlichkeit. Fehlermeldungen werden dem Nutzer bei falsch verwendeten Funktionen sofort mitgeteilt und erklärt.

A13: Fehlertoleranz

Durch die Schnittstelle und die damit angebotene Funktion, die das Logging übernimmt, wird aufgrund des Compilers bereits die Logging-Form erzwungen. Unbekannter Statuscode wird dennoch von OpenSearch akzeptiert und wiedergegeben. Bei ungültigem JSON-Format wird die Nachricht von Logstash nicht verworfen, sodass damit weiter gearbeitet werden kann beziehungsweise ein Alert versendet wird.

A14: Verfügbarkeit

Die Cluster-Implementierung in dieser Arbeit beinhaltet zwei Knoten, die jeweils jeden Typen gleichzeitig repräsentieren. Laut der offiziellen Dokumentation in Bezug auf die Clusterformation sollten Knoten vorhanden sein, die master-geeignet sind. Dies wird hier erfüllt. Geografische Zonen werden im ODS nicht genutzt, somit muss nicht ausdrücklich beachtet werden, dass eigenständige Daten-Knoten erzeugt werden. Auch sind keine plötzlich großen Mengen an Daten geplant, die gesammelt werden müssen. Somit besteht hier keine Notwendigkeit, explizite Aufnahme-Knoten zu definieren. Ebenso wird ODS keine suchintensiven Aufgaben durchführen, sodass auch kein explizit definierter Koordinierenden-Knoten gebraucht wird. Jedoch wird empfohlen, mindestens drei Master-Knoten zu verwenden, um plötzlich auftretende Ausfälle eines Knotens kompensieren zu können sowie die Möglichkeit zu haben, während einer Wartung weiterhin Logs verarbeiten zu können.

7.3 Zusammenfassung

Das Konzept zur Observability in einer microservicebasierten Umgebung wurde mithilfe der vorgestellten Programme erläutert, die anhand ihrer grundlegenden Funktionsweisen vorgestellt wurden. Dabei wurden eine beispielhafte Implementierung und die zugehörigen Arbeitsschritte so weit dokumentiert, wie es bis zur Einreichung dieser Arbeit möglich war.

Kapitel 2 bietet einen Überblick über die technische Umgebung sowie eines ihrer Kernprobleme. Hierzu wurde auch die für dieses Problem passende Lösung, die Observability, in ihrer prinzipiellen Theorie vorgestellt.

Im nächsten Abschnitt wurden bereits vorhandene und bekannte Open-Source-Programme vorgestellt und miteinander verglichen, die genau dem Ziel der Erfüllung des Konzepts der Observability dienen. Hier wurde sich nach einer Re-Evaluierung auf eine kompatible Softwarekette festgelegt und die hierfür entscheidenden Kriterien wurden verdeutlicht.

In Kapitel 4 wurden sämtliche funktionale und nichtfunktionale Anforderungen erfasst, die zur Observability nötig wären. Diese wurden abschließend in Kapitel 7 ausgewertet. Diese können größtenteils durch die festgelegten und kooperierenden Programme Filebeat, Logstash, OpenSearch und OpenSearch Dashboards erfüllt werden.

Diese Funktionsweisen der Werkzeuge wurden in Kapitel 5 näher vorgestellt, um einen konzeptionellen Zusammenhang zur Observability herzustellen. Danach wurde auf die Zielarchitektur eingegangen, die mithilfe dieser Kette schließlich erreicht werden soll.

Abschließend wurde in Kapitel 6 schrittweise die beispielhafte Implementierung des Konzepts dokumentiert. Neben dem bisherigen Logging des ODS wurde hier auch die Einrichtung erläutert. Hierbei ist anzumerken, dass zunächst Version 1.0 von OpenSearch genutzt wurde, anschließend Version 1.1. Kurz vor der Abgabe dieser Bachelorarbeit, am 23. November 2021, erschien allerdings bereits Version 1.2.

Infolge der Recherche und der Nutzung einer unbrauchbaren Softwarekette aufgrund der zu Beginn nicht in Betracht gezogenen Lizenz und dem daraus entstandenem Zeitdruck konnten nicht alle Anforderungen erfüllt werden, die an die Observability des ODS gestellt wurden. Trotzdem ist diese neue Softwarekette eine sinnvolle Ergänzung zum ODS. Beinahe alle gestellten Anforderungen können hierdurch zukünftig erfüllt und erweitert werden.

8 Ausblick

Die derzeitige Implementierung der Softwarekette aus Filebeat, Logstash, OpenSearch und OpenSearch Dashboards ist zwar unvollständig, nichtsdestotrotz sind diese Werkzeuge eine passende Ergänzung des ODS in Bezug auf die notwendige Observability. Dies liegt daran, dass auch der ODS als Microservice-Anwendung stetig wächst und neue Dienste angebunden werden, die ebenso überblickt werden müssen. Zudem ist OpenSearch erst im Jahre 2021 entstanden und wird kontinuierlich weiterentwickelt, unter Beibehaltung der Apache-Lizenz 2.0.

Im Hinblick auf eine zukünftige Bereitstellung und Wartung in der Produktion des Dienstes ist die Sicherheit noch nicht völlig ausreichend. OpenSearch bietet zur Gewährleistung von Authentifizierung und Zugangskontrolle ein eigenes Plugin an, das für einen hinreichenden Schutz sorgt und ergänzt werden sollte.

Die Clusterformation von OpenSearch kann zusätzlich dahingehend erweitert werden, größere Stabilität zu erlangen. So wird beispielsweise für Anwendungsfälle in der Produktion empfohlen, eine Konfiguration aus mindestens drei Master-Knoten zu wählen, um eine durchgängige Protokollierung von Log-Informationen sicherzustellen, selbst bei einem Ausfall oder einer Wartung.

Sinnvoll wäre auch ein Warnmeldesystem, das beispielsweise eine E-Mail versendet, sobald eine Bedingung erfüllt wird, beispielsweise eine Anzahl an Fehlern in einem definierten Zeitrahmen oder ein kritischer Wert der Speichernutzung. Eine solche frühe Warnung kann entscheidend sein, um einen möglichen Absturz eines Dienstes rechtzeitig zu verhindern. Auch diese Funktion unterstützt OpenSearch mit zwei weiteren Plugins: *Alerting* und *Anomaly detection*.

Literaturverzeichnis

Amazon Web Services. (2021). *Was ist Opensearch?* Abgerufen am 05. August 2021 von https://aws.amazon.com/de/opensearch-service/the-elk-stack/what-is-opensearch/

Amazon Web Services, & Django Software Foundation. (2021). *Documentation*. Von https://opensearch.org/docs/latest/opensearch/index/ abgerufen

Demchenko, M. (2020). *Microservices vs. Monolithic*. Abgerufen am 06. Oktober 2021 von https://ncube.com/blog/microservices-vs-monolithic-which-architecture-suits-best-for-your-project

Docker Inc. (2021). *Overview of Docker Compose*. Abgerufen am 20. November 2021 von https://docs.docker.com/compose/

Dudenredaktion. (2021). *Duden online*. Abgerufen am 20. Oktober 2021 von https://www.duden.de/rechtschreibung/Trace

Elastic. (2021). *Filebeat*. Abgerufen am 18. November 2021 von https://www.elastic.co/de/beats/filebeat

Elastic. (2021). *Kibana*. Abgerufen am 02. August 2021 von https://www.elastic.co/de/kibana/

Elastic. (2021). *Logstash Introduction*. Abgerufen am 02. August 2021 von https://www.elastic.co/guide/en/logstash/current/introduction.html

Elastic. (2021). *Was ist Elasticsearch?* Abgerufen am 02. August 2021 von https://www.elastic.co/de/what-is/elasticsearch

Figueroa, A. (2019). *Monolithic versus microservices, and all in between*. Abgerufen am 06. Oktober 2021 von https://medium.com/swlh/monolithic-vs-micro-services-and-all-in-between-7d496408ad02

Fluentd Project. (2021). *What is Fluentd?* Abgerufen am 03. August 2021 von https://www.fluentd.org/architecture

Fowler, M., & Lewis, J. (2015). *Microservices: Nur ein weiteres Konzept in der Softwarearchitektur oder mehr?* Abgerufen am 06. Oktober 2021 von https://www.sigs-datacom.de/uploads/tx_dmjournals/fowler_lewis_OTS_Architekturen_15.pdf

Free Software Foundation. (2015). *Why the Affero GPL*. Abgerufen am 2021. 10 07 von https://www.gnu.org/licenses/why-affero-gpl.en.html

GitHub, Inc. (2021). *Licensing a repository*. Abgerufen am 05. August 2021 von https://docs.github.com/en/repositories/managing-your-repositorys-settings-and-features/customizing-your-repository/licensing-a-repository

Grafana Labs. (2021). *Grafana*. Abgerufen am 02. August 2021 von https://grafana.com/grafana/

Indrasiri, K., & Siriwardena, P. (2018). *Microservices for the Enterprise.* Apress, Berkeley, CA.

Krochmalski, J. (2016). *Developing with Docker.* Packt Publishing, Limited.

Luber, S., & Litzel, N. (2020). *Was ist Logstash?* Abgerufen am 02. August 2021 von https://www.bigdata-insider.de/was-ist-logstash-a-939698/

Marten, K., & Smith, D. (2021). *What's new in Kibana 7.11.0*. Abgerufen am 02. August 2021 von https://www.elastic.co/de/blog/whats-new-kibana-7-11-0-alerting-generally-available

Pacheco, V. F. (2018). *Microservice Patterns and Best Practices.* Birmingham.

Prometheus. (2021). *Overview*. Abgerufen am 02. August 2021 von https://prometheus.io/docs/introduction/overview/

Richardson, C. (2021). *Pattern: Microservice Architecture*. Abgerufen am 04. Oktober 2021 von https://microservices.io/patterns/microservices.html

Tamura, K. (2014). *Unified Logging Layer: Turning Data into Action*. Abgerufen am 03. August 2021 von https://www.fluentd.org/blog/unified-logging-layer

Ullenboom, C. (2014). *Java SE 8-Standard-Bibliothek: Das Handbuch für Entwickler.* Galileo Computing.

Werner, K. L. (2019). Design und Implementierung eines Konzepts zur Live-Konfiguration. Friedrich-Alexander-Universität Erlangen-Nürnberg.

Anhang

GNU Affero General Public License 3.0

Permissions	Limitations	Conditions
Commercial use	Liability	License and copyright notice
Modification	Warranty	State changes
Distribution	Trademark use	Disclose source
Patent use		Network use is distribution
Private use		Same license

GNU General Public License 3.0

Permissions	Limitations	Conditions
Commercial use	Liability	License and copyright notice
Modification	Warranty	State changes
Distribution		Disclose source
Patent use		Same license
Private use		

Apache License 2.0

Permissions	Limitations	Conditions
Commercial use	Liability	License and copyright notice
Modification	Warranty	State changes
Distribution		Disclose source
Private use		Same license

MIT License

Permissions	Limitations	Conditions
Commercial use	Liability	License and copyright notice
Modification	Warranty	
Distribution		
Private use		

Erläuterungen der Permissions-Angaben

Commercial use	Das lizenzierte Material und die Derivate dürfen für kommerzielle Zwecke verwendet werden.
Modification	Das lizenzierte Material darf geändert werden.
Distribution	Das lizenzierte Material darf verteilt werden.
Patent use	Diese Lizenz bietet eine ausdrückliche Gewährung von Patentrechten von Mitwirkenden.
Private use	Das lizenzierte Material darf privat verwendet und modifiziert werden.

Erläuterungen der Limitations-Angaben

Trademark use	Diese Lizenz besagt ausdrücklich, dass sie *keine* Markenrechte gewährt, auch wenn Lizenzen ohne eine solche Angabe wahrscheinlich keine impliziten Markenrechte gewähren.
Liability	Diese Lizenz beinhaltet eine Haftungsbeschränkung.
Warranty	Diese Lizenz besagt ausdrücklich, dass sie *keine* Garantie bietet.

Erläuterungen der Conditions-Angaben

License and copyright notice	Dem lizenzierten Material muss eine Kopie des Lizenz- und Urheberrechtsvermerks beigefügt werden.
State changes	Änderungen am Lizenzmaterial sind zu dokumentieren.

Disclose source	Der Quellcode muss bei der Verteilung des lizenzierten Materials zur Verfügung gestellt werden.
Network use is distribution	Benutzer, die mit dem lizenzierten Material über das Netzwerk interagieren, erhalten das Recht, eine Kopie des Quellcodes zu bekommen.
Same license	Änderungen müssen bei der Verbreitung des lizenzierten Materials unter derselben Lizenz veröffentlicht werden. In einigen Fällen kann eine ähnliche oder verwandte Lizenz verwendet werden.

Anhang 1: Tabellarischer Überblick verschiedener Open-Source-Lizenzen (GitHub, Inc., 2021)

```yaml
version: '3'
services:
  opensearch-node1: # Name des 1. Knotens
    image: opensearchproject/opensearch:latest # Laden der letzten stabilen Version
    container_name: opensearch-node1 # Container-Name
    environment:
      - cluster.name=opensearch-cluster # Cluster-Name
      - node.name=opensearch-node1 # Name des 1. Knotens
      # Hier kann der Typ des Knotens definiert werden (standardmäßig sind alle true)
      # - node.master=true
      # - node.data=true
      # - node.ingest=true
      - discovery.seed_hosts=opensearch-node1,opensearch-node2 # Knoten zur Clustererkennung
      - cluster.initial_master_nodes=opensearch-node1,opensearch-node2 # Beide können als Master-Knoten fungieren
      - bootstrap.memory_lock=true # Deaktiviert zusammen mit Memlock-Einstellungen (s. unten) das Austauschen
      - compatibility.override_main_response_version=true # Versionskompatibilität für Logstash und Filebeat
      - "OPENSEARCH_JAVA_OPTS=-Xms512m -Xmx512m" # Minimale und maximale Java-Heap-Größe (beide 50% des System-RAM)
    ulimits:
      memlock: # siehe Zeile 15
        soft: -1
        hard: -1
      nofile:
        soft: 65536 # Maximale Anzahl geöffneter Dateien für den Opensearch-Benutzer
        hard: 65536 # Auf modernen Systemen auf mindestens 65536 einstellen
    volumes:
      - opensearch-data1:/usr/share/opensearch/data # Pfad innerhalb eines Containers
      # Keine persistenten Daten! Datenverlust durch lokalen Pfad gewährleisten:
      # - ./src/opensearch-data1:/usr/share/opensearch/data
    ports:
      - 9200:9200
      - 9600:9600 # Erforderlich für Performance Analyzer
    networks:
      - opensearch-net # Netzwerkeinstellung zur Portweiterleitung gesetzt
  opensearch-node2: # Name des 2. Knotens
    image: opensearchproject/opensearch:latest
    container_name: opensearch-node2
    environment:
      - cluster.name=opensearch-cluster
      - node.name=opensearch-node2
      - discovery.seed_hosts=opensearch-node1,opensearch-node2
      - cluster.initial_master_nodes=opensearch-node1,opensearch-node2
      - bootstrap.memory_lock=true
      - compatibility.override_main_response_version=true
      - "OPENSEARCH_JAVA_OPTS=-Xms512m -Xmx512m"
    ulimits:
      memlock:
        soft: -1
        hard: -1
      nofile:
        soft: 65536
        hard: 65536
    volumes:
      - opensearch-data2:/usr/share/opensearch/data
    networks:
      - opensearch-net
  opensearch-dashboards:
    image: opensearchproject/opensearch-dashboards:latest
    container_name: opensearch-dashboards
    ports:
      - 5601:5601
    expose:
      - "5601"
    environment:
      OPENSEARCH_HOSTS: '["https://opensearch-node1:9200","https://opensearch-node2:9200"]'
      # Falls als Umgebungsvariable definiert, muss diese ohne Leerzeichen angegeben sein
    networks:
      - opensearch-net
volumes:
  opensearch-data1: # Auskommentieren, um persistente Daten zu gewährleisten (s. oben)
  opensearch-data2:
networks:
  opensearch-net:
```

Anhang 2: Verwendete *docker-compose.yml*-Datei zum Starten von Opensearch