Technische Hochschule
Brandenburg
University of
Applied Sciences

# Implementierung eines Stacks zum Erzeugen, Verarbeiten und Visualisieren von Telemetriedaten für ein Microservice basiertes System

**Bachelorarbeit**

zur Erlangung des Grades Bachelor of Science
des Fachbereichs Informatik und Medien der
Technischen Hochschule Brandenburg

vorgelegt von:
Franz Stefan Kurt Dalitz

Betreuer: Prof. Dr.-Ing. Sven Buchholz
Zweitbetreuer: Dipl.-Wirt.-Inf. Jan Vogt

Brandenburg an der Havel, den 27. August 2023

# Kurzfassung

Einsicht in die Abläufe und Leistung heutiger Software und verteilter Systeme zu erlangen, wird durch stetig steigende Komplexität zunehmend erschwert. Die Firma Peterson Technologies GmbH implementierte zur Linderung dieser Problematik ein internes Telemetriesystem. Ziel dieser Arbeit ist es, das System auf Wunsch der Firma durch ein neues zu ersetzen, um so die Überwachung zu verbessern und aktuellere Software zu verwenden. Nach einer Untersuchung des vorherigen Systems werden Prioritäten festgelegt, welche die spätere Nutzung, Instandhaltung und Erweiterung des neuen Systems erleichtern sollen. Darauf folgen die Planung seiner Architektur, die Auswahl und Implementierung der Software und schließlich das Erzeugen einer einfachen Dokumentation. Der resultierende Telemetriestack ist funktionstüchtig und erfüllt die Vorstellungen der Firma. Dennoch weist er einige Probleme, wie beispielsweise eine hohe CPU Belastung auf, welche zusammen mit Möglichkeiten der Verbesserung und Erweiterung abschließend betrachtet werden.

## Schlüsselwörter

Telemetrie, Distributed Tracing, Microservices, Grafana, OpenTelemetry

# Inhaltsverzeichnis

**Abbildungsverzeichnis** ..... IV
**1 Einleitung und Grundlagen** ..... 1
1.1 Ziel und Motivation ..... 1
1.2 Peterson Technologies GmbH ..... 1
1.3 Grundlagen ..... 2
1.3.1 Microservice Softwarearchitektur ..... 2
1.3.2 Containerisierung und Orchestrierung ..... 3
1.3.3 Observability und Telemetrie ..... 3
1.3.4 Architektur von Telemetriestacks ..... 5
**2 Geschichte der Software-Telemetrie** ..... 5
**3 Konzeption** ..... 9
3.1 Vorgehensweise ..... 9
3.2 Ausgangszustand ..... 10
3.3 Architektur und Softwareauswahl ..... 12
**4 Implementierung** ..... 19
4.1 Instrumentierung ..... 19
4.1.1 Java Services ..... 19
4.1.2 Node.js Services ..... 20
4.2 Pipelines ..... 23
4.2.1 Tracelog-Agent ..... 23
4.2.2 Metrics-Agent ..... 25
4.2.3 External-Agent ..... 27
4.3 Backends ..... 28
4.4 Grafana ..... 30
4.4.1 Konfiguration ..... 30
4.4.2 Visualisierung und Alarme ..... 31
4.5 Dokumentation ..... 35
**5 Evaluation** ..... 35
**6 Zusammenfassung und Ausblick** ..... 37
6.1 Zusammenfassung ..... 37
6.2 Ausblick ..... 38
**Literaturverzeichnis** ..... 40
**Ehrenwörtliche Erklärung** ..... 48

# Abbildungsverzeichnis

Abbildung 1: Log Nachricht eines Node Exporter Dienstes (Eigene Bildschirmaufnahme).... 4
Abbildung 2: Metriken einer Open Policy Agent Instanz (Eigene Bildschirmaufnahme).... 4
Abbildung 3: Beispielhafte Darstellung eines Traces (Eigene Darstellung)....4
Abbildung 4: Ausgangszustand (Eigene Darstellung).... 12
Abbildung 5: Architektur des neuen Telemetriestacks (Eigene Darstellung)....19
Abbildung 6: Architektur des Tracelog-Agenten (Eigene Bildschirmaufnahme)....25
Abbildung 7: Architektur des Metrics-Agenten (Eigene Darstellung).... 27
Abbildung 8: Architektur der External-Agenten (Eigene Bildschirmaufnahme).... 28
Abbildung 9: Ausschnitt des “Service Details” Dashboards (Eigene Bildschirmaufnahme).... 33
Abbildung 10: Open Policy Agent Dashboard (Eigene Bildschirmaufnahme)....34
Abbildung 11: Hohe CPU-Auslastung des Tracelog-Agenten (Eigene Bildschirmaufnahme).... 36

# 1 Einleitung und Grundlagen

## 1.1 Ziel und Motivation

Diese Bachelorarbeit wird in Kooperation mit der Firma Peterson Technologies GmbH erarbeitet. Das Ziel ist, einen neuen Telemetriestack zu entwickeln, welcher den bereits in der Firma vorhandenen ersetzen soll. Der Stack soll dabei alle Fähigkeiten des vorherigen Systems behalten und dieses lediglich um Tracing-Funktionalität erweitern. Außerdem wird diese Gelegenheit genutzt, um die Architektur des Systems zu überarbeiten. Dabei soll versucht werden, möglichst wenig Eingriffe in den Code der Anwendungen der Firma vorzunehmen. Da die Peterson Technologies GmbH ihre Anwendungen mit der Microservice-Architektur entwickelt, besteht ihre Software aus vielen einzelnen Teilen und Eingriffe müssten für jeden dieser Teile durchgeführt werden. Dadurch wäre der Arbeitsaufwand exponentiell größer. Das resultierende System soll leicht zu bedienen sein, was unter anderem durch die Wahl von Software mit gleichartigen Möglichkeiten der Konfiguration realisiert werden kann. Wie "schwer" es ist, das System zu verwenden, beschreibt hier der Aufwand, der mit dem Starten, Stoppen, Aktualisieren, Pflegen und Erweitern einzelner Systemkomponenten und des gesamten Systems verbunden ist. Es soll außerdem nach der Implementierung des neuen Systems eine grundlegende Dokumentation verfasst werden, die sich mit der Handhabung des Systems und seinen Komponenten auseinandersetzt.

## 1.2 Peterson Technologies GmbH

Die Firma Peterson wurde im Jahr 1920 von der Peterson Familie zur Inspektion von Korn, das in den Flüssen und Kanälen der Niederlande gehandelt wurde, gegründet. Peterson erweiterten kontinuierlich die von ihnen angebotenen Dienstleistungen, unter anderem in der Öl- und Gasexploration und Produktion. Control Union, eine Firma, die ursprünglich im landwirtschaftlichen Sektor aktiv war, erwarb später das vollständige Eigentum an Peterson, wodurch die Firma Peterson and Control Union entstand (Peterson and Control Union, o. J.-a). Diese Firma bietet heute Dienste in Bereichen wie Logistik, Qualität und Zertifizierungen an und hat über 4000 Mitarbeiter aus mehr als 70 Ländern (Peterson and Control Union, o. J.-b). ORCA Geo Services wurde als Tochterfirma von Peterson and Control Union gegründet. Ihr Ziel war es, Geodaten in Zertifizierungsprozesse zu integrieren. Anfänglich überwachte und optimierte sie mit Hilfsmitteln wie Satellitenbildern und GPS-Daten

Tee- und Fruchtplantagen, unter anderem auf den Philippinen und in Vietnam. Etwa ein Jahr nach ihrer Gründung bekamen ORCA Geo Services von ihrem Mutterunternehmen den Auftrag, ein neues Zertifizierungssystem zu entwickeln. Kurz darauf entstand ein weiteres Projekt: die Entwicklung eines "Traceability-Systems", mit dem Lieferketten zertifizierter Produkte nachverfolgt werden können. An beiden Systemen wird noch heute aktiv gearbeitet. Im Jahr 2002 benannte sich die Firma zu Peterson Technologies GmbH um, da sie hauptsächlich Technologie für Peterson and Control Union entwickelte und somit der Aspekt der Geodaten immer weiter in den Hintergrund rückte. Heute arbeiten ungefähr 30 Personen in verschiedenen Ländern wie Spanien, Argentinien und Deutschland für die Peterson Technologies GmbH.

## 1.3 Grundlagen

### 1.3.1 Microservice Softwarearchitektur

Über viele Jahre hinweg wurden Anwendungen immer umfangreicher, wodurch es immer schwerer wurde, sie instand zu halten und funktionale Änderungen an ihnen vorzunehmen (Thönes, 2015). Die Softwarearchitektur von Anwendungen, deren gesamte Logik durch einen einzigen Prozess ausgeführt wird, nennt sich "monolithisch" (Ponce et al., 2019). Das Aktualisieren oder Hinzufügen von Bibliotheken kann ein monolithisches System destabilisieren, eine einzige Veränderung erfordert den Neustart der gesamten Anwendung und es muss durchgehend die gleiche Technologie verwendet werden (Dragoni et al., 2017). In der Microservice-Architektur hingegen besteht eine Anwendung aus mehreren deutlich kleineren "Services", die über APIs miteinander kommunizieren (Fowler & Lewis, 2015). Dadurch kann die Entwicklung, Ausführung und Wartung der einzelnen Microservices separiert werden. Die Architektur eignet sich besonders für "Cloud-Native Applications", also Anwendungen, die in einer Cloud ausgeführt werden und häufig für die gleichzeitige globale Nutzung durch tausende Kunden ausgelegt sind (Gannon et al., 2017). Microservices eignen sich dazu aufgrund ihrer Eigenschaften wie Portabilität, Flexibilität und Robustheit und weil sie Anwendungen skalieren, also ihre Leistung abhängig von ihrer Belastung anpassen können (Dragoni et al., 2018).

### 1.3.2 Containerisierung und Orchestrierung

Ursprünglich wurden virtuelle Maschinen in Cloud-Infrastrukturen zur Zuweisung und zum Management von Hardware verwendet (Pahl, 2015). Nun wird vermehrt

Containerisierung eingesetzt, eine Virtualisierungsmethode, bei der Anwendungen zusammen mit all ihren Abhängigkeiten zu einer Einheit zusammengefasst werden, die man als Container bezeichnet (Potdar et al., 2020). Das Ziel der Containerisierung ist es, kleine, portable Anwendungen zu erzeugen, die auf vielen Servern gleichzeitig ausgeführt und miteinander verbunden werden können (Pahl, 2015). Die Software Docker kann anhand von Dockerfile-Dateien Bilder von Containern erzeugen, die unabhängig von ihrer Umgebung sind (Potdar et al., 2020). Um mehrere Container gleichzeitig zu kontrollieren, werden Orchestrierungswerkzeuge verwendet. Docker Compose dient der Definition und Ausführung von Multi-Container Anwendungen durch YAML-Dateien (Pan et al., 2019). Technologien wie Docker Swarm und Kubernetes ermöglichen die Verwaltung containerisierter Anwendungen in Clustern, also Gruppen, mehrerer Maschinen. Dadurch bieten sie eine erhöhte Sicherheit vor Ausfällen (Al Jawarneh et al., 2019). Beide Technologien arbeiten dabei mit einer "Master Node" (Maschine), die alle anderen Nodes und Prozesse kontrolliert (Al Jawarneh et al., 2019).

### 1.3.3 Observability und Telemetrie

Im Bereich der Softwareentwicklung bedeutet Observability Einsicht in die komplizierten und unsichtbaren Systeme der heutigen Zeit zu erlangen (Cantrill, 2006). Dies ist in Microservice-Architekturen aufgrund ihrer Komplexität besonders schwer zu erreichen (Bento et al., 2021). Thakur und Chandak (2022) definieren Observability als die Fähigkeit, den momentanen Zustand eines Systems anhand von Daten, die es selbst erzeugt, zum Beispiel Telemetriedaten wie Logs, Metriken und Traces bestimmen zu können. Telemetrie, das Messen von Werten aus der Ferne, wird auch in vielen anderen Bereichen der Wissenschaft eingesetzt, wie zum Beispiel in der Luft- und Raumfahrt, der Medizin und der Meteorologie (Kiourti et al., 2014; Shi et al., 2018; Sidqi et al., 2018).

Logs sind strukturierte oder unstrukturierte Zeichenketten wie beispielsweise Stack-Traces, die sehr detailliert beschreiben, wie oder warum bestimmte Probleme in einem System auftreten (Karumuri et al., 2021). Der Begriff "Metriken" bezeichnet numerische Daten, die das generelle Verhalten einer Anwendung über die Zeit hinweg beschreiben, wie zum Beispiel CPU- und Arbeitsspeicherauslastung (Usman et al., 2022). Eine Metrik besteht aus mehreren Eigenschaften wie einem Zeitstempel, Namen, Wert und Labeln (Usman et al., 2022). Traces sind Informationen über die Pfade von Anfragen (Karumuri et al., 2021) und bestehen aus Bäumen von "Spans", die jeweils einzelne Prozessschritte darstellen (Thakur &

Chandak, 2022). Wenn Traces Prozess-, Netzwerk- oder Sicherheitsgrenzen überqueren, werden sie als “Distributed Traces” bezeichnet (Thakur & Chandak, 2022). In Microservice-Systemen kommt demnach Distributed Tracing zum Einsatz, da bei der Kommunikation zwischen den Services Prozessgrenzen überschritten werden. Die Abbildungen 1 bis 3 zeigen beispielhafte Darstellungen für die vorgestellten Arten von Telemetriedaten.

```
ts=2023-07-23T17:33:11.816Z caller=tls_config.go:274 level=info msg="Listening on" address=[::]:9100
```

*Abbildung 1: Log Nachricht eines Node Exporter Dienstes*
*(Quelle: Eigene Bildschirmaufnahme)*

```
# HELP go_cgo_go_to_c_calls_calls_total Count of calls made from Go to C by the current process.
# TYPE go_cgo_go_to_c_calls_calls_total counter
go_cgo_go_to_c_calls_calls_total 2
# HELP go_cpu_classes_gc_mark_assist_cpu_seconds_total Estimated total CPU time goroutines spent performing GC tasks
to assist the GC and prevent it from falling behind the application. This metric is an overestimate, and not directly
comparable to system CPU time measurements. Compare only with other /cpu/classes metrics.
# TYPE go_cpu_classes_gc_mark_assist_cpu_seconds_total counter
go_cpu_classes_gc_mark_assist_cpu_seconds_total 0.0734745
# HELP go_cpu_classes_gc_mark_dedicated_cpu_seconds_total Estimated total CPU time spent performing GC tasks on
processors (as defined by GOMAXPROCS) dedicated to those tasks. This includes time spent with the world stopped due to
the GC. This metric is an overestimate, and not directly comparable to system CPU time measurements. Compare only with
other /cpu/classes metrics.
# TYPE go_cpu_classes_gc_mark_dedicated_cpu_seconds_total counter
go_cpu_classes_gc_mark_dedicated_cpu_seconds_total 7.292481321
# HELP go_cpu_classes_gc_mark_idle_cpu_seconds_total Estimated total CPU time spent performing GC tasks on spare CPU
resources that the Go scheduler could not otherwise find a use for. This should be subtracted from the total GC CPU
time to obtain a measure of compulsory GC CPU time. This metric is an overestimate, and not directly comparable to
system CPU time measurements. Compare only with other /cpu/classes metrics.
# TYPE go_cpu_classes_gc_mark_idle_cpu_seconds_total counter
go_cpu_classes_gc_mark_idle_cpu_seconds_total 0.079386604
```

*Abbildung 2: Metriken einer Open Policy Agent Instanz*
*(Quelle: Eigene Bildschirmaufnahme)*

*Abbildung 3: Beispielhafte Darstellung eines Traces*
*(Quelle: Eigene Darstellung, generiert mit Mermaid in Markdown)*

### 1.3.4 Architektur von Telemetriestacks

Um Telemetriedaten effektiv nutzen zu können, müssen diese erzeugt, gespeichert, abgerufen und dargestellt werden. Der Prozess, Anwendungen so anzupassen, dass

sie Telemetriedaten generieren, nennt sich Instrumentierung (The OpenTelemetry Authors, o. J.). Eine Instrumentierung kann entweder manuell durch Eingriffe in den Quellcode einer Anwendung geschehen, oder automatisch, also ohne den Quellcode zu verändern (The OpenTelemetry Authors, o. J.). Bibliotheken zur Instrumentierung bieten oft Möglichkeiten, Daten direkt an Backends zu senden; die Nutzung von Telemetrie-Kollektoren wie dem OpenTelemetry Collector oder dem Grafana Agenten kann jedoch das Sammeln, Verarbeiten und Weiterleiten der Daten unterstützen (Grafana Labs, o. J.-b; The OpenTelemetry Authors, o. J.). Es können auch Agenten wie Grafana Promtail eingesetzt werden (Horalek et al., 2023), die bestimmte Arten von Telemetriedaten sammeln. Als Telemetrie-Backends werden Anwendungen eingesetzt, die das Speichern und Abrufen der Daten ermöglichen, wie zum Beispiel die Programme Prometheus und Elasticsearch (Al Maruf et al., 2022). Mithilfe von Software wie Grafana oder Kibana können die Daten dann aus den Backends geladen, grafisch dargestellt und analysiert werden (Al Maruf et al., 2022; Cruz et al., 2021). Um anhand selbst definierter Regeln Alarme auszulösen, können beispielsweise Grafana oder die Kombination von Prometheus mit einer Alertmanager Instanz verwendet werden (Al Maruf et al., 2022; Cruz et al., 2021).

# 2 Geschichte der Software-Telemetrie

Ursprünglich wurden Anwendungen durch einfache, manuell in den Quellcode eingearbeitete Konsolenausgaben überwacht (Kabinna et al., 2016). Als Teil des Sendmail Projekts wurde in den 1980er Jahren der Syslog Standard entwickelt, durch den Logs über Netzwerke versendet werden können. Der Standard fand breite Akzeptanz und erlaubte Entwicklern, das Logging ihrer Anwendungen auszulagern (Irwin & Loyola, 2022). Über viele Jahre hinweg wurde Syslog durch neue Projekte, wie Syslog-ng (One Identity LLC., o. J.), Rsyslog (Adiscon GmbH, o. J.) und journald (systemd, o. J.) erweitert. Heutzutage kommen verschiedene Log Standards und Formate, je nach Anwendungsbereich zum Einsatz, wie das Common Log Format (CLF), und Windows Event Logs (Dwyer & Truta, 2013; Sheeraz et al., 2023). Mit der Veröffentlichung der Log4j Bibliothek in 2001, welche bereits seit 1996 entwickelt wurde, begann die kontinuierliche Evolution der Logging-Bibliotheken (Gülcü, 2003; Kabinna et al., 2016). Diese ermöglichen durch Logger das Generieren einheitlicher Logs mit expliziten Schweregraden und deren Anreicherung mit zusätzlichen Informationen (Gülcü, 2003; Kabinna et al., 2016). Zum Sammeln, Empfangen und Aggregieren von Logs wurden Dienste wie Apache Flume und das von Facebook in

2008 veröffentlichte Scribe entwickelt, um so den Umgang mit den steigenden Mengen an Logdaten zu erleichtern (Bifet, 2013; Liu et al., 2014). Fluentd, ein in 2011 entwickelter Dienst zum Sammeln und Verarbeiten von Logs, wurde 2016 von der Cloud Native Computing Foundation (CNCF) als Projekt aufgenommen (The Linux Foundation, 2019a). Als Teil der Linux Foundation hilft die CNCF Cloud-nativen Open Source Projekten und ordnet ihre Reife einer der Kategorien "Sandbox", "Incubating" oder "Graduated" zu (The Linux Foundation, o. J.). 2019 verlieh CNCF Fluentd aufgrund des Erfolges der Software den Graduated Status (The Linux Foundation, 2019a). Nach der Veröffentlichung der Logstash Software, deren erste Versionshinweise auf der offiziellen Seite von Elastic in das Jahr 2014 zurückreichen (Elasticsearch B.V., o. J.), wurde der "ELK-Stack" zu einer populären Logging-Lösung (Horalek et al., 2023). Dieser Stack, dessen Komponenten alle von dem Elastic Unternehmen entwickelt werden, ist in der Lage, Logs mit Logstash zu sammeln, mit Elasticsearch zu speichern und mit Kibana zu analysieren und visualisieren (Horalek et al., 2023). Ein weiterer moderner Logging-Stack, der "PLG-Stack", sammelt mit dem Promtail Agenten Logs, speichert diese mit Loki und visualisiert sie mit Grafana (Horalek et al., 2023).

Seit den späten 1960er Jahren wurden Software-Metriken wie "LOC" (Lines of Code) und "Defects per KLOC" (Fehler pro eintausend Zeilen Code) zur Einschätzung von Software und der Leistung von Programmierern eingesetzt (Fenton & Neil, 1999). Diese frühen Metriken basierten auf Analysen des Quellcodes. Ab den 70er Jahren wurde versucht, sie zur Vorhersage von Softwarequalität und zur Einschätzung des Erfolges verschiedener Methoden und Werkzeuge einzusetzen (Fenton & Neil, 1999). Frühe Methoden der Überwachung von Prozessen waren der "ps" (Process Status) Befehl, vorgestellt in der dritten Edition des UNIX Handbuchs in 1973, das "/proc" Dateisystem welches Prozessinformationen speichert und der "top" (Table of Processes) Befehl, der späteren Task-Managern ähnelt (Killian, 1984; LeFebvre, 2004; Thompson & Ritchie, 1973). Mit dem Konzept der Vernetzung von Computern entstand das Interesse an der Überwachung von Netzwerkverkehr. Mit dem Simple Network Monitoring Protocol (SNMP) legte die Internet Engineering Task Force (IETF) 1989 einen ersten Standard fest, der dies ermöglichte (Case et al., 1989). 1995 veröffentlichte Tobias Oetiker den Multi Router Traffic Grapher (MRTG), ein Perl Script, welches mit SNMP die Verkehr-Zähler von Routern liest und die gesammelten Daten mit einem C Programm visualisiert (Oetiker, o. J.). Oetiker entwickelte dann auch das Round Robin Database tool (RRDtool), um Probleme mit der Performance von MRTG zu adressieren und das Werkzeug flexibler zu machen (Beverly, 2002).

Später entstanden mehrere Frameworks wie Cricket und Cacti, die anstrebten, RRDtool weiter zu verbessern und mit stärkeren Interfaces zu versehen (Beverly, 2002; The Cacti Group, Inc., o. J.). Im Juli 2005 wurde Collectd veröffentlicht, ein Daemon der Metriken von Betriebssystemen, Anwendungen, Logs und externen Geräten erzeugen und diese über Netzwerke veröffentlichen kann (*Main Page*, o. J.). Graphite, eine Monitoring-Anwendung, die Zeitreihendaten speichern und visualisieren kann, entwarf und programmierte Chris Davis im Jahr 2006 für die Firma Orbitz, welche das Projekt 2008 unter einer Open Source Lizenz veröffentlichte (*Graphite*, o. J.). Metriken können unter anderem von Collectd, StatsD oder Cronjobs an Graphite gesendet werden (*Graphite*, o. J.). StatsD ist ein Daemon, der 2008 von der Firma Etsy veröffentlicht wurde und auf einem UDP Port Nachrichten sammelt, aus ihnen Metriken extrahiert und diese an Graphite sendet (Henderson, o. J.; Malpass, 2011). Cronjobs sind Aufgaben, die mit dem "crontab" Befehl definiert und durch Cron periodisch ausgeführt werden (Keller, 1999). Skripte zum Erzeugen und Senden von Metriken können so automatisiert werden (*Graphite*, o. J.). Die Firma Soundcloud startete 2012 das Open Source Projekt Prometheus, da sie mit ihrer auf StatsD und Graphite basierenden Monitoring Lösung unzufrieden war (Volz & Rabenstein, 2015). Prometheus sammelt Metriken von instrumentierten Anwendungen und speichert diese in einer Zeitreihendatenbank, von der aus die Daten mit der Prometheus Query Language abgefragt werden können (Volz & Rabenstein, 2015). Im Jahr 2013 wurde InfluxDB veröffentlicht, eine Datenbank zum Speichern großer Mengen von Zeitreihendaten (Petre et al., 2019). Die Cloud Native Computing Foundation nahm Prometheus im Mai 2016 nach Kubernetes als zweites Projekt auf und verlieh ihm 2018 den Graduated Status (The Linux Foundation, 2016, 2018).

Bereits im Jahr 1945 entwickelte Alan Turing zum Verknüpfen von Unterprozeduren das Konzept eines Stacks, den er "Reversion Storage" nannte (Henriksson, 2011). Diese grundlegende Struktur ermöglichte Programmierern bereits von Anfang an das Debuggen von Programmen durch das Nachverfolgen von Funktionsaufrufen (Lester, 1971). Eine Auflistung aktiver Unterprozeduren in umgekehrter Reihenfolge ihres Aufrufs wurde als "Call-Stack Trace" bezeichnet (Copperman, 1992). Mit Remote Procedure Calls (RPC) entstand das Konzept von Programmen, die über Netzwerke Befehle und Parameter versenden, um Prozesse in fremden Umgebungen auszuführen und dann mit deren Ergebnissen weiterzuarbeiten (Birrell & Nelson, 1984). Dadurch wurde die Umsetzung des Prinzips der Call-Stack Traces erschwert, da Teile von Programmen extern ausgeführt werden konnten. Bald entwickelten sich

erste Tracing-Frameworks wie X-Trace mit dem Ziel, das Nachvollziehen der komplexen Prozesse verteilter Systeme zu erleichtern (Fonseca et al., 2007). 2010 veröffentlichte Google in einem Paper das Prinzip ihrer internen Tracing-Lösung Dapper und ihre Erfahrungen mit der Software nach 2 Jahren in aktivem Gebrauch (B. H. Sigelman et al., 2010). Sie sprachen bereits von Spans, die einzelne Arbeitsschritte darstellen und die IDs der Traces und ihrer Eltern-Spans speichern, um so Trace-Bäume zu konstruieren (B. H. Sigelman et al., 2010). In einem Blog Post kündigte Twitter 2012 die Veröffentlichung ihrer Software Zipkin, die aus einer Implementierung des Papers von Google entstand, unter der APLv2 Lizenz an (Aniszczyk, 2012). Sie stellte mit Instrumentierungen durch die Finagle Bibliothek und Apache Scribe und Cassandra als Backend (Aniszczyk, 2012) ein vollständiges Tracing-System dar. Die Firma Uber begann 2015 an einem stark von Dapper und Zipkin inspirierten RPC-Protokoll zu arbeiten, mit speziellen Feldern, die das Tracing verteilter Systeme erleichtern sollten (Shkuro, 2017). Um das Protokoll zu testen, verwendete sie das UI von Zipkin und einen Backend-Prototypen mit Riak und Solr, aufgrund mangelnder Erfahrung mit Scribe und Cassandra (Shkuro, 2017). Der Prototyp skalierte nicht gut mit ihrem Netzwerkverkehr, weshalb Uber anfing an der Jaeger Software zu arbeiten und somit allmählich eigene Lösungen für alle Systemkomponenten zu entwickeln (Shkuro, 2017). Gegen Ende des Jahres 2015 entstand das OpenTracing Projekt, welches als einheitlicher und anbieterunabhängiger Tracing-Standard für Anwendungen und deren Instrumentierung dienen sollte (B. Sigelman, 2016; Woods, 2016). Es wurde 2016 von CNCF als drittes Projekt aufgenommen (Woods, 2016). 2017 wurde Jaeger zu einem Open Source Projekt und wurde noch im selben Jahr von der CNCF angenommen, die dem Projekt 2019 den Graduated Status verlieh (The Jaeger Authors, 2017; The Linux Foundation, 2019b; Woods, 2017). Google veröffentlichte 2018 OpenCensus, eine anbieterunabhängige Open Source Bibliothek für Tracing und Metrik-Sammlung (Shah, 2018). Durch die Existenz von OpenTracing und OpenCensus, zweier Projekte mit ähnlichen Zielen, die unterschiedliche Architekturen hatten und nicht versuchten, gegenseitige Kompatibilität zu erreichen, entstand große Unsicherheit für Entwickler (B. Sigelman & McLean, 2019). Im März 2019 kündigten die beiden Projekte daher an, eine Fusionierung mit maximaler Abwärtskompatibilität zu planen (B. Sigelman, 2019). In einem Blog Post einen Monat später veröffentlichten sie ihren neuen Namen, OpenTelemetry, und einen Entwicklungszeitplan (Young et al., 2019). CNCF akzeptierte 2021 OpenTelemetry und archivierte 2022 OpenTracing (The Linux Foundation, 2021, 2022).

Aus dem Bedürfnis heraus, schnell und unkompliziert Einblicke in Anwendungen zu erlangen, entstanden noch vor der Veröffentlichung des Dapper Papers Firmen wie Splunk (Splunk Inc., o. J.), die Application Performance Management (APM) Werkzeuge als Software-As-A-Service (SaaS) Dienste anbieten. Das Prinzip von SaaS basiert darauf, Software zu verwenden, die nicht selbst, sondern von dritten Anbietern gehostet wird (Dubey & Wagle, 2007). APMs sammeln Daten über die Leistung von Systemen und analysieren und visualisieren sie (Ahmed et al., 2016). Zu ihnen gehören beispielsweise Dynatrace (Dynatrace LLC., o. J.), New Relic (New Relic, Inc., o. J.), und Datadog (Datadog, o. J.).

# 3 Konzeption

## 3.1 Vorgehensweise

Zunächst wird das bereits bestehende System untersucht. Um die beschriebenen Ziele zu erreichen, wird dann die Architektur des neuen Telemetriesystems geplant und passende Software für jede Stack-Komponente ausgewählt. Dabei sollen auch die schon in der Einleitung beschriebenen Wünsche und Kriterien, wie zum Beispiel möglichst wenig Eingriffe in den Code der Firmeneigenen Anwendungen vorzunehmen, respektiert werden. Außerdem wird die vorhandene Infrastruktur beachtet, welche somit Entscheidungen über Architektur und Softwareauswahl beeinflussen kann. Zum Treffen dieser Entscheidungen werden außerdem Gespräche mit den Entwicklern der Peterson Technologies GmbH geführt, die in dieser Arbeit nicht dokumentiert werden.

Nach dem Planungsprozess folgt die Implementierung des neuen Systems. Das heißt, alle Komponenten müssen programmiert, an den richtigen Stellen installiert und konfiguriert werden. Wenn alle Systemkomponenten funktionstüchtig sind, werden alle systemunabhängigen Konfigurationen und Daten übertragen. Dabei handelt es sich um die Visualisierungen und Alarme des vorherigen Systems. Durch weitere Gespräche mit den Entwicklern der Firma wird festgestellt, welche der Visualisierungen übernommen, angepasst oder verworfen werden sollen.

Der letzte Schritt der Implementierung besteht darin, die Dokumentation zu verfassen.

## 3.2 Ausgangszustand

Die Entwickler der Peterson Technologies GmbH arbeiten an mehreren Softwareprojekten für Peterson and Control Union gleichzeitig und verwenden für alle Projekte die Microservice-Architektur. Ein großer Teil der Services, aus denen ihre Anwendungen bestehen, sind in der Programmiersprache Java verfasst. Dazu verwendet die Firma das Build Tool Maven und das Open Source Tool Spring Boot. Einige Services basieren auf der Open Source Laufzeitumgebung Node.js, sind in der Sprache Typescript geschrieben und verwenden das Node.js Framework Express.js. Ein kleiner Teil aller Services, der überwiegend aus Node.js Services besteht, ist containerisiert und wird mit Docker ausgeführt.

Die Entwicklung und das Ausführen der Software ist auf verschiedene Umgebungen verteilt. Es werden Umgebungen für die reine Entwicklung, zum Testen und zum spontanen Beheben schwerwiegender Fehler eingesetzt. Das aktive System, auf dem die Kunden der Firma ihre Software verwenden, ist auch als eigene Umgebung definiert.

Jeder Microservice wird mit einem eigenen GitLab Repository verwaltet. Die GitLab Instanz, in der sich die Repositorys befinden, ist privat und läuft auf einem firmeninternen Server. Dort wird der Code aller Anwendungen ohne umgebungsabhängige Informationen und Konfigurationen gespeichert. Über "Issues" (Listen von Aufgaben) wird in diesen Repositorys auch die Arbeit an der Software koordiniert. Mithilfe von GitLab-Pipelines werden Services getestet, gebaut und in beliebigen Umgebungen ausgeführt.

Die Java Services der Peterson Technologies GmbH erzeugen ihre Metriken mit der Micrometer-Integration von Spring Boot. In den Services wird dann ein Endpunkt geöffnet, über den die Metriken im Prometheus Format abgerufen werden können. Eine zentrale Prometheus Instanz sammelt über diese Endpunkte die Metriken aller Services aus allen Umgebungen. Die Prometheus Instanz speichert diese Metriken dann direkt lokal auf dem gleichen Server, auf dem sie selbst läuft. Logs erzeugen die Java Services mit der Logback Bibliothek. Diese werden dann im JSON-Dateiformat ausgeleitet und an dem Ort gespeichert, an dem der jeweilige Service ausgeführt wurde. Eine zentrale Grafana Promtail Instanz sammelt die Logs und sendet diese an eine Loki Instanz weiter, welche sie dann lokal speichert und zum Abrufen bereitstellt.

Für ihre Node.js basierten Services hat die Firma bis jetzt noch keine Metrik-Instrumentierung implementiert. Auch ihre Logging-Implementierung ist nur sehr grundlegend. Es wurde bisher keine explizite Logging Bibliothek verwendet, sondern lediglich normale Konsolenausgaben.

Für die Software, die im System der Firma eingesetzt, aber nicht von ihr entwickelt wird und die keine eigenen Metrik-Endpunkte hat, werden sogenannte “Exporter” eingesetzt, die das Sammeln von Metriken ermöglichen. Dadurch kann auch die Leistung dieser externen Software überwacht werden. Zu den verwendeten Exportern zählen beispielsweise die Apache-, Redis- und Elasticsearch-Exporter.

Alle gesammelten Daten werden mit Grafana abgerufen und visualisiert. Die Grafana Nutzerdaten der Firmenmitglieder und die Visualisierungen werden in einer Postgres Instanz gespeichert.

Die Prometheus Instanz wertet basierend auf den gesammelten Metriken Regeln aus, die von den Entwicklern definiert wurden. Sollten diese Regeln verletzt werden, beginnt Prometheus Alarme auszusenden. Eine Alertmanager Instanz, dedupliziert die so erzeugten Alarme, sorgt also dafür, dass jeder Alarm nur einmal ausgeführt wird (Prometheus Authors, o. J.). Wenn ein Alarm für eine festgelegte Dauer ausgelöst ist, überschreitet er einen Schwellwert, woraufhin der Alertmanager den Alarm als ausformulierte Nachricht an definierte Benachrichtigungs-Endpunkte sendet. In diesem Fall sind das die E-Mail und Microsoft Teams Kanäle der Firma. In Abbildung 4 ist das vorherige Telemetriesystem vereinfacht dargestellt.

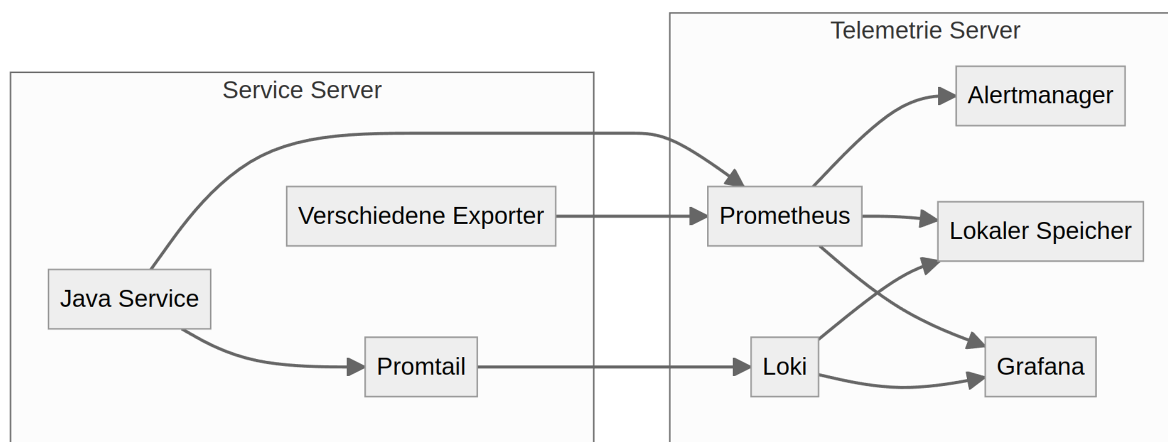


*Abbildung 4: Ausgangszustand*
*(Quelle: Eigene Darstellung, generiert mit Mermaid in Markdown)*

## 3.3 Architektur und Softwareauswahl

Bei der Planung der Architektur und Auswahl der Software wird auf mehrere Faktoren geachtet. Zu diesen Faktoren zählen, Probleme wenn möglich auf gleiche Arten und Weisen zu lösen, Software von möglichst wenig unterschiedlichen Anbietern zu verwenden und gleiche Sprachen und Konfigurationsarten zu verwenden. Durch diese Prioritäten wird versucht, ein möglichst homogenes und stabiles Umfeld für das System zu erzeugen.

Wie in der Geschichte der Telemetrie erwähnt, schlossen sich OpenCensus und OpenTracing zu dem Projekt OpenTelemetry zusammen, welches nun den Standard für Telemetrie-Formate und Instrumentierungen darstellt. Dort wo es möglich ist, werden von OpenTelemetry entwickelte Lösungen verwendet, um nicht Gefahr zu laufen, in der Zukunft Probleme mit Inkompatibilitäten zu bekommen. Die Peterson Technologies GmbH hat bereits vor dieser Arbeit sowohl Grafana als auch Grafana Loki in ihrem Telemetriesystem verwendet. Seit der Implementierung des vorherigen Systems haben Grafana Labs weitere Softwarelösungen für Komponenten von Telemetriesystemen entwickelt und veröffentlicht. Software von Grafana Labs wird im Sinne der Priorität eines homogenen Umfeldes und um auf den Erfahrungen der Entwickler der Peterson Technologies GmbH aufzubauen, anderen Optionen vorgezogen.

Im neuen Telemetriesystem sollen alle drei Hauptarten von Telemetriedaten implementiert werden, also Metriken, Logs und Traces. Die Java Services der Firma

erzeugen bereits durch Micrometer Metriken und durch Logback Logs. Es gibt zwar auch andere Möglichkeiten Java Anwendungen für diese Telemetriesignale zu instrumentieren, wie zum Beispiel Dropwizard für Metriken und Log4J für Logs, doch es bestehen keine ersichtlichen Gründe, diesen Teil des Systems zu überarbeiten, da er einwandfrei funktioniert. Die Auswahl einer Tracing Instrumentierung stellt die erste wichtige Entscheidung dar. Anwendungen wie Jaeger und Zipkin hatten ursprünglich eigene Bibliotheken und SDKs zur manuellen Instrumentierung, zum Beispiel die Brave Bibliothek für Zipkin (*Zipkin*, o. J.). Einige Zeit lang wurde auch aktiv mit OpenCensus und OpenTracing instrumentiert. Es ist auch nach wie vor möglich Implementierungen auf Basis dieser Optionen zu entwickeln, doch OpenTelemetry stellt Instrumentierungsmethoden für Java zur Verfügung, von denen aus den beschriebenen Gründen Gebrauch gemacht wird. Das OpenTelemetry Format wird mittlerweile auch von Jaeger unterstützt (The Jaeger Authors, o. J.-a) und es werden Exporter für sowohl Jaeger als auch Zipkin angeboten (*OpenTelemetry SDK Autoconfigure*, o. J.).

Es gibt zwei verschiedene Arten der Instrumentierung durch OpenTelemetry. Die manuelle und die automatische Instrumentierung. Die manuelle Variante basiert auf direkten Eingriffen in den Quellcode einer Anwendung. Dabei werden per Hand Traces in allen Funktionen von Interesse erzeugt. Automatische Instrumentierungen ermöglichen Traces zu generieren, ohne jegliche Eingriffe in die Anwendungen vornehmen zu müssen. Von OpenTelemetry gibt es zur automatischen Instrumentierung von Java Anwendungen einen Java Agenten. Dieser injiziert dynamisch, also zur Laufzeit, Bytecode in eine Anwendung, um Traces für populäre Bibliotheken wie GraphQL, Hibernate und Logback zu erzeugen (*Supported Libraries, Frameworks, Application Servers, and JVMs*, o. J.). Besonders interessant sind für die Peterson Technologies GmbH das Nachverfolgen von Datenbankzugriffen und das Erfassen der Übergänge zwischen ihren Microservices. Beides kann durch die automatische Instrumentierung überwacht werden. Deshalb wird im neuen System die automatische Instrumentierung der manuellen vorgezogen. Dadurch, dass somit keine Eingriffe in den Quellcode nötig sind, wird das System weniger anfällig für zukünftige Änderungen in Instrumentierungsbibliotheken.

Auch für Node.js Anwendungen steht von OpenTelemetry eine automatische Instrumentierung für Traces zur Verfügung (The OpenTelemetry Authors, o. J.). Metriken wurden bisher in den Node.js Services noch nicht instrumentiert. Um auch diese Services vollständig in das neue System zu integrieren, müssen jedoch alle

Telemetriesignale implementiert werden. Eine einfache Metrik Instrumentierung, die nur grundlegende Metriken erzeugt und diese über einen Endpunkt abrufbar macht, reicht dafür vollkommen aus. Zwei Module, die dies ermöglichen, sind die "sdk-metrics" und "prom-client" Module. Aufgrund des Strebens nach einem homogenen System wäre es sinnvoll, die Lösung von OpenTelemetry, die sdk-metrics, zu verwenden. Allerdings bietet dieses Modul keine Möglichkeit der automatischen Instrumentierung (*OpenTelemetry Metrics SDK*, 2023). Das prom-client Modul hingegen erzeugt automatisch die von Prometheus als Standard definierten Metriken (*Prometheus Client for Node.Js*, 2023), weshalb stattdessen dieses Modul verwendet wird. Es wäre theoretisch möglich, die generierten Metriken direkt an die Backends zu versenden (*OpenTelemetry Collector Metrics Exporter for Node with Grpc*, 2023); die Pipelines des Systems werden jedoch manuell alle Metriken einsammeln, um dabei die Up-Metrik zu generieren, mit deren Hilfe später die Zustände der Services überwacht werden.

Da die Peterson Technologies GmbH plant, in der Zukunft alle ihre Anwendungen zu containerisieren und da einige Services bereits mit Docker ausgeführt werden, bietet es sich an, jetzt schon den von Google entwickelten cAdvisor (Container Advisor) in das System einzuarbeiten. Dieser Service erzeugt automatisch Metriken von Containern, die in derselben Umgebung laufen wie er selbst (*google/cadvisor*, o. J.). Die Metriken können dann über einen Endpunkt in cAdvisor abgerufen werden.

Die einfachen Konsolenausgaben, welche die bisherige Logging Implementierung der Node.js Services darstellten, müssen für die vollständige Integration in das Telemetriesystem zu vollwertigen Logs transformiert werden. Es existieren verschiedene Logging Module für Node.js, wie Winston und pino (*pino*, o. J.; *winston*, o. J.), die zum Erzeugen und Emittieren von Logs genutzt werden können. Auch hier gibt es eine Lösung von OpenTelemetry, das "sdk-logs" Modul, welches so wie das sdk-metrics Modul keine automatische Instrumentierung anbietet (*OpenTelemetry Logs SDK*, 2023). In diesem Fall stellt die fehlende automatische Instrumentierung jedoch kein Problem dar, da die Nachrichten bereits erzeugt werden und daher nur umgewandelt werden müssen. Dementsprechend wird an dieser Stelle die OpenTelemetry Lösung gewählt. Mithilfe des "exporter-logs-otlp-grpc" Moduls können die generierten Logs dann direkt an den Pipeline-Endpunkt versendet werden (*OpenTelemetry Collector Logs Exporter for Node with Grpc*, 2023).

Wie in der Grundlagen-Sektion angesprochen, ist es theoretisch möglich, ein Telemetriesystem ohne Pipeline zu entwickeln, bei dem Daten direkt an Backends gesendet werden. Allerdings kommen die Instrumentierten Daten von mehreren Orten (wegen der Verteilung der Services auf mehrere Server) und in verschiedenen Formaten (durch unterschiedliche Instrumentierungen). Deshalb werden die Daten durch Pipelines gesammelt und verarbeitet. Außerdem wird das System in der Produktionsumgebung von vielen Nutzern verwendet, wodurch eine große Menge an Telemetriedaten erzeugt wird. Der OpenTelemetry Java Agent könnte diese Daten theoretisch selbst puffern, doch dann läge diese Aufgabe bei den Instrumentierten Anwendungen statt im Telemetriesystem, was sie unnötig belasten würde. Stattdessen können die Pipelines das Puffern der Daten übernehmen, um die Microservices zu entlasten.

Als Pipelines könnten beispielsweise der OpenTelemetry Collector oder der Grafana Agent verwendet werden. Beide sind sehr flexibel und bieten Empfang, Puffern, Transformieren und Weiterleiten von Metriken, Logs und Traces in verschiedenen Formaten an (Grafana Labs, o. J.-b; The OpenTelemetry Authors, o. J.). Der Grafana Agent nutzt selbst einen eingebetteten OpenTelemetry Collector und kann daher zuverlässig mit Daten im OpenTelemetry Format arbeiten (Grafana Labs, o. J.-b). Der Agent nutzt außerdem weitere Integrationen, zum Beispiel für verschiedene Metrik Exporter (Grafana Labs, o. J.-b). Dadurch, dass der Grafana Agent von Grafana Labs entwickelt wird, garantiert er im Sinne eines homogenen Systems, mit anderen Komponenten von Grafana Labs reibungslos zusammenzuarbeiten.

Es gibt zwei verschiedene Arten, den Grafana Agenten auszuführen, den Standard Modus und den Flow Modus. Im Standard Modus wird der Agent über eine YAML-Datei konfiguriert. Der Flow Modus ist der neuere der beiden Modi und verwendet zur Konfiguration eine eigene, von Terraform inspirierte Sprache namens River (Grafana Labs, o. J.-a). Mit River werden programmatisch einzelne Komponenten definiert und dann zu einer Pipeline verbunden. Dadurch ist der Agent im Flow Modus extrem flexibel, weshalb er im neuen Telemetriesystem eingesetzt wird.

Als Tracing Backends stehen beispielsweise Jaeger und Zipkin zur Verfügung. Auch für Metriken existieren mehrere Optionen, wie InfluxDB, Graphite und Prometheus. Grafana Labs haben jedoch die Backends Loki, Mimir und Tempo für Logs, Metriken und Traces entwickelt, welche aus dem Streben nach einem homogenen System eingesetzt werden. Diese Backends werden alle über YAML-Dateien konfiguriert,

wobei sich ihre Konfigurationsmöglichkeiten ähneln (Grafana Labs, o. J.-d, o. J.-e, o. J.-f). Wie auch der Grafana Agent können die Grafana Backends in verschiedenen Modi ausgeführt werden, wodurch jedoch nicht die Art der Konfiguration, sondern das Verhalten der Anwendungen beeinflusst wird. Intern besteht jedes der Backends aus mehreren Komponenten. Im monolithischen Modus ausgeführt, laufen alle Unterkomponenten in einer einzigen Instanz. Im Microservices Modus werden sie als einzelne Instanzen gestartet. Dann kann die Menge der Instanzen bestimmter Komponenten erhöht oder verringert werden, um die Belastung optimal zu verteilen. Tempo und Loki bieten zusätzlich jeweils einen skalierbaren Modus an, der bei Tempo mehreren Instanzen erlaubt zusammen zu arbeiten und bei Loki das Lesen und Schreiben in einzelne Instanzen separiert (Grafana Labs, o. J.-d, o. J.-f). Um den Konfigurationsaufwand vorerst so gering wie möglich zu halten, wird für alle Backends der monolithische Modus verwendet.

Wie bereits beschrieben speicherten im vorherigen Telemetriesystem die Loki und Prometheus Instanzen ihre Daten direkt lokal. Einige Backends wie Mimir und Tempo sind allerdings darauf ausgelegt, Daten mit Object Storage zu speichern (Grafana Labs, o. J.-e, o. J.-f). Die Nutzung von Object Storage bietet sich für Telemetriedaten ohnehin an, da sie geeignet ist, große Mengen ungeordneter Daten zu speichern (Google LLC, o. J.). Object Storage kann über Cloud Anbieter implementiert werden, beispielsweise über Google Cloud Storage oder Amazon Web Services AWS. Das bietet den Vorteil, den Speicher nicht selbst pflegen zu müssen. Allerdings sind diese Dienste kostenpflichtig (Amazon Web Services, Inc., o. J.; *Cloud Storage Pricing*, o. J.) und ihre Nutzung macht ein System abhängig von Dritten. Daher wird der Speicher für das neue System manuell aufgesetzt. Dazu können beispielsweise Programme wie MinIO, SeaweedFS oder Garage verwendet werden (MinIO, Inc., o. J.; *Quick Start*, o. J.; *SeaweedFS*, o. J.). Die Auswahl einer in jeder Hinsicht perfekten Object Storage Software wird im Rahmen dieser Arbeit nicht priorisiert. Zu Testzwecken wird MinIO verwendet, da die Entwickler der Software mit einfacher Konfiguration und hoher Performance werben (MinIO, Inc., o. J.). Sollte MinIO sich nach dieser Arbeit als nicht langfristig geeignet erweisen, bestehen einige Ausweichmöglichkeiten (wie SeaweedFS oder Garage), zwischen denen eine neue Auswahl getroffen werden kann.

Manche Backends wie Jaeger und Zipkin bieten eigene Visualisierungswerkzeuge an (The Jaeger Authors, o. J.-b; *Zipkin*, o. J.). Das Telemetriesystem wird aber neben Traces noch Metriken und Logs erzeugen, die von einer reinen Trace-Visualisierung

nicht verarbeitet werden können und in jeweils eigenen Backends gespeichert sind. Daher sollte eine eigenständige Software zum Abrufen und Visualisieren der Daten verwendet werden. Das von Grafana Labs entwickelte Grafana war wie bereits erwähnt schon vor dieser Arbeit im Telemetriesystem in Gebrauch und wird deshalb weiterhin verwendet werden. Unter anderem auch, weil die vorherigen Visualisierungen in das neue System übertragen werden sollen. Auf Wunsch der Geschäftsleitung der Peterson Technologies GmbH wird außerdem weiterhin eine Postgres Instanz zum Speichern der Nutzerdaten und Visualisierungen von Grafana verwendet.

Grafana Mimir hat Ruler und Alertmanager Komponenten, die zusammen genutzt werden könnten, um die Alarme des vorherigen Systems zu übernehmen. Es ist jedoch theoretisch nicht nötig, von diesen Komponenten Gebrauch zu machen, da seit der Entwicklung des letzten Telemetriesystems das in Grafana integrierte Alarmsystem kontinuierlich verbessert wurde. Dieses wird im neuen Telemetriesystem eingesetzt und bietet den Vorteil, dass dabei mehrere Datenquellen gleichzeitig zum Modellieren von Regeln eingesetzt werden können. Grafana übernimmt dann sowohl das Überprüfen und Auslösen der Regeln wie es die Aufgabe von Prometheus war, als auch das Deduplizieren der Alarme und Versenden von Benachrichtigungen (Grafana Labs, o. J.-c).

Es muss auch entschieden werden, auf welche Art und Weise alle gewählten Komponenten des Stacks ausgeführt werden. Sie könnten beispielsweise wie im vorherigen System direkt als Binärdateien ausgeführt werden. Alternativ ist es auch möglich, den Stack mit Hilfe von Container-Verwaltungssystemen wie Docker oder Kubernetes zu orchestrieren, was das gleichzeitige Kontrollieren seiner Komponenten erleichtert. Da die Peterson Technologies GmbH Docker bereits zum Ausführen einiger Programme verwendet, kommt im neuen System Docker zum Einsatz. Dadurch ist garantiert, dass einige Entwickler sich schon mit der Software auskennen, wodurch der Übergang zwischen den Systemen erleichtert wird. Bisher verwendete die Firma Bilder von Containern, welche direkt mit dem "docker run" Befehl ausgeführt wurden. Dabei mussten die jeweiligen Befehle entweder dokumentiert oder in Dateien an den Ausführungsorten gespeichert werden, da sonst die genauen Konfigurationen verloren gingen. Weil das neue System so leicht wie möglich zu bedienen sein soll, wird stattdessen Docker Compose verwendet. Durch das Zusammenfassen mehrerer Service-Konfigurationen in einer einzigen

Datei, können dann alle definierten Services gemeinsam kontrolliert, also beispielsweise gestartet und gestoppt werden (Docker Inc., o. J.-c).

Um die Dokumentation für das System zu entwickeln, wäre es unter anderem möglich, eine firmeninterne Website mit einem Wiki-Framework aufzusetzen. GitLab bietet jedoch auch Wiki Funktionalität und verwendet das “GitLab flavored Markdown”. Dieses basiert auf dem Commonmark Format, welches es um einige Funktionen, zum Beispiel um Diagramme und Flowcharts erweitert (*GitLab Documentation*, o. J.). Weil die Peterson Technologies GmbH ihre Anwendungen teilweise bereits mit GitLab Wikis dokumentiert, wird das neue System diese auch zu Dokumentationszwecken verwenden.

Abbildung 5 visualisiert die getroffene Softwareauswahl. Die Instrumentierung basiert auf dem Java Agenten von OpenTelemetry und einem selbst geschriebenen Node.js Modul. Mit Instanzen von cAdvisor werden zusätzliche Metriken für Containerisierte Anwendungen erzeugt. Der Grafana Agent wird als Pipeline eingesetzt, um die erzeugten Daten zu sammeln, zu verarbeiten und weiterzuleiten. Als Backends für die Telemetriedaten werden die Grafana Labs Programme Loki, Mimir und Tempo verwendet, welche jeweils ihre Daten mit Object Storage durch MinIO speichern. Zur Visualisierung kommt eine Grafana Instanz zum Einsatz, welche ihre Nutzerdaten und Visualisierungen mit Postgres speichert. Grafana übernimmt auch das Auswerten von Alarmen und das Aussenden von Benachrichtigungen. Das System wird in einem GitLab Wiki dokumentiert.

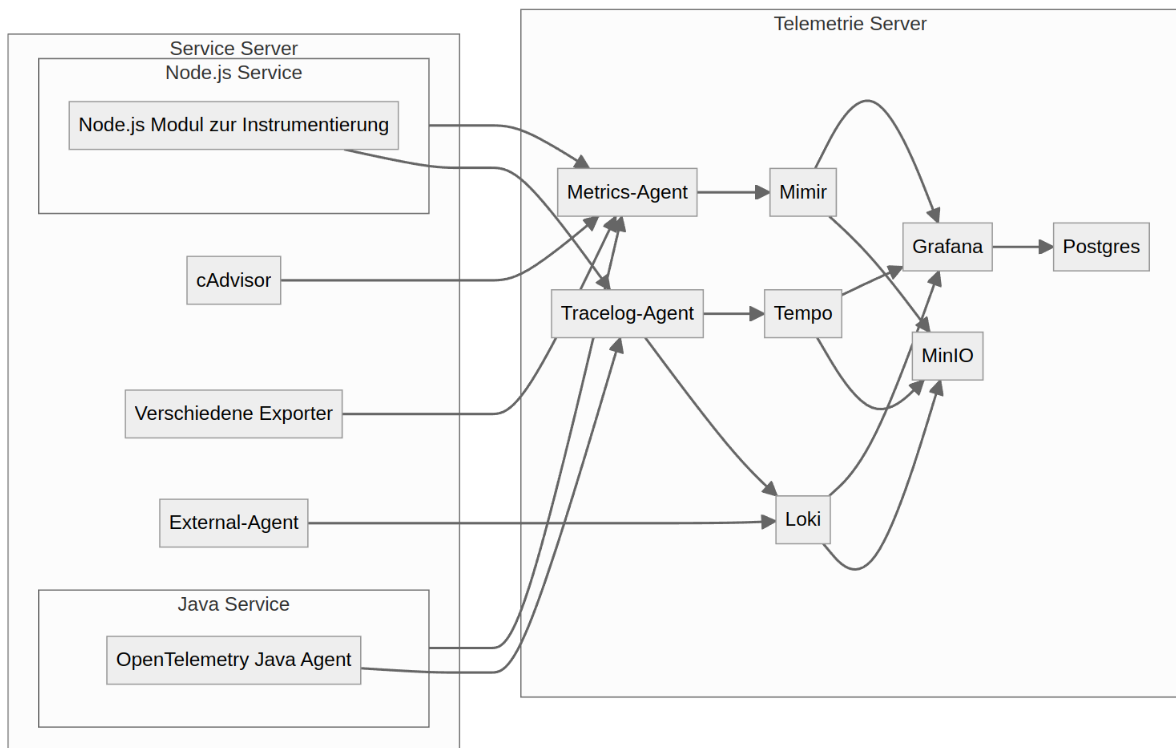


*Abbildung 5: Architektur des neuen Telemetriestacks*
*(Quelle: Eigene Darstellung, generiert mit Mermaid in Markdown)*

# 4 Implementierung

## 4.1 Instrumentierung

### 4.1.1 Java Services

Den Java Agenten veröffentlicht OpenTelemetry als JAR-Datei auf GitHub (*OpenTelemetry Instrumentation for Java*, o. J.). Diese Datei wird heruntergeladen und auf jedem Server, auf dem sich Java Services befinden, zentral gespeichert. Mit zentral ist gemeint, dass der Agent nicht in umgebungsspezifischen Ordnerstrukturen gespeichert wird, sondern unabhängig und nahe an der Wurzel des Servers. Dort wird auch ein Bash-Skript zum Aktualisieren des Agenten hinterlegt, welches zunächst die Verbindung zu GitHub prüft, dann die vorhandene Version des Agenten löscht und die neueste aus dem offiziellen Repository herunterlädt. Die Java Services der Peterson Technologies GmbH werden per Hand durch die Entwickler der Firma oder durch Pipelines ausgeführt. Die Ausführungsbefehle der Services sind dazu in eigenen Dateien gespeichert, inklusive umgebungsspezifischer Einstellungen und Variablen. Auch der Agent benötigt verschiedene Konfigurationen, die über

Umgebungsvariablen angegeben werden (*OpenTelemetry SDK Autoconfigure*, o. J.). Um die Services zusammen mit dem Agenten zu starten, wird ihren Ausführungsbefehlen die "-javaagent" Flagge hinzugefügt, zusammen mit dem Pfad zur JAR-Datei des Agenten. Die Konfigurationen für die Agenten werden den Befehlen durch "-D" Flaggen angehängt, um die gewünschten Exporter, Endpunkte und Service Informationen festzulegen. Die Exporter-Konfigurationen definieren, welche Telemetriedaten in welchem Format exportiert werden sollen. Traces können beispielsweise im OTLP, Jaeger oder Zipkin Format exportiert werden. Da weder Jaeger noch Zipkin verwendet werden, wird der OTLP-Exporter für alle Signale eingesetzt. Der Metrik-Exporter kann dabei ignoriert werden, da ein Grafana Agent diese später manuell sammelt. Als Endpunkt für alle versendeten Daten wird die spätere Adresse des OTLP-Endpunktes der Grafana Agenten angegeben. Mit der "OTEL_RESOURCE_ATTRIBUTES" Variable können die exportierten Daten mit zusätzlichen Informationen angereichert werden. Diese Funktionalität wird genutzt, um später die genaue Herkunft der Daten bestimmen zu können. Dazu wird der Name der Services, das System, zu dem sie gehören, und ihre Umgebung mit der Variable angegeben. Damit die Instrumentierung in Kraft tritt, werden alle Services einmal heruntergefahren und mit den angepassten Befehlen neu gestartet. In einem Einzelfall traten dabei Probleme mit dem Apache Server der Firma auf, welche noch durch die System Administratoren behoben werden müssen.

### 4.1.2 Node.js Services

Um die Instrumentierung der Node.js Services zu implementieren, wird ein eigenes npm Paket entwickelt, welches Metriken, Logs und Traces erzeugt. Da alle Node.js Services der Peterson Technologies GmbH auf dem npm Paket "Express" basieren, nimmt die Instrumentierung eine Express Anwendung als Argument.

Die Metrik Implementierung soll vor allem dazu dienen, einen abrufbaren Endpunkt zur Verfügung zu stellen. Das prom-client Modul bietet die "collectDefaultMetrics" Funktion an, die sowohl allgemeine als auch Node.js spezifische Metriken wie "event loop lag" und "active handles" generiert (*Prometheus Client for Node.Js*, 2023). Zum internen Sammeln der Metriken benötigt die "collectDefaultMetrics" Funktion ein Metrik-Register, welches ihr als Argument gegeben wird. Diese Register sind im prom-client Modul selbst definiert und können durch einfache Konstruktor-Aufrufe erzeugt werden. Um die gesammelten Metriken über einen Endpunkt abrufbar zu machen, wird der Express Anwendung ein neuer Pfad hinzugefügt. Dieser kann später über Umgebungsvariablen konfiguriert werden und nimmt einen Standardwert

von “/metrics” an. Wenn er aufgerufen wird, gibt die Instrumentierung die im internen Register gespeicherten Metriken im Prometheus Format zurück. Die Standard Prometheus-Metriken bieten keinen tiefen Einblick in Anwendungen und sind in diesem Fall außerdem ungenau, weil die Container, in welche die Node.js Services eingebettet sind, auch Ressourcen verbrauchen. Deshalb werden ergänzend cAdvisor Instanzen eingesetzt, um akkurate Metriken bezüglich der Leistung der Container sowie Anfrageraten und Netzwerkverkehr zu erzeugen. Diese Metriken werden dann direkt über die cAdvisor Instanzen abgerufen. Das Labeling der Metriken der Java Services erfolgt später beim Abrufen ihrer Endpunkte durch die Agenten. Bei den Node.js Services ist dies jedoch nur bei der Hälfte der Metriken möglich, die auch tatsächlich über die Endpunkte der Services abgerufen wird. Eine cAdvisor Instanz kann Metriken für mehrere Container gleichzeitig erzeugen und der Pipeline ist nur bekannt, dass sie einen cAdvisor-Endpunkt anfragt, nicht für welche oder wie viele Container die Instanz Metriken erzeugt. Deshalb werden den Containern der Node.js Services zusätzliche Label angehängt, welche den Metriken durch das Konfigurieren einer Whitelist in cAdvisor weitergegeben werden. So kann später eindeutig die Herkunft der cAdvisor-Metriken bestimmt werden. Um die Label eindeutig mit der Firma zu assoziieren, bekommen sie das Präfix “com.petersoncontrolunion”. Zum automatischen Generieren und Exportieren von Traces wird zuerst die “NodeSDK” von OpenTelemetry (*OpenTelemetry SDK for Node.Js*, 2023) initialisiert. Dieser wird der “OTLPTraceExporter” (*OpenTelemetry Collector Exporter for Node with Grpc*, 2023) hinzugefügt, der dafür zuständig ist, die Traces im OTLP Format zu versenden. Traces werden durch die automatische Instrumentierung von OpenTelemetry (*OpenTelemetry Meta Packages for Node*, 2023) generiert. Zusätzliche Informationen über die Services werden wieder mit der “OTEL_RESOURCE_ATTRIBUTES” Umgebungsvariable angegeben, die mit Hilfe des “dotenv” Moduls aus “.env”-Dateien ausgelesen werden. Damit die Trace Instrumentierung in Kraft tritt, wird dann die SDK mit ihrer “start” Funktion aktiviert.

Um die Konsolenausgaben der Node.js Services zu Logs zu transformieren, werden die Standard “console” Methoden von Javascript, also “trace”, “log”, “debug”, “info”, “warn” und “error” überschrieben. Dazu werden zunächst die originalen Formen dieser Funktionen mit ihren Namen als Schüssel in einem Objekt hinterlegt. Dann wird über die Schlüssel iteriert und in jedem Schleifendurchgang die entsprechende Konsolenfunktion neu deklariert. Die überschriebenen Funktionen führen erst ihre originalen Versionen aus. So wird sichergestellt, dass die Konsolenausgaben trotz der Transformation weiterhin wie gewohnt zu Debugging Zwecken verwendet werden

können. Danach wird der Schweregrad der Logs bestimmt, welcher im Normalfall äquivalent zur originalen Konsolenfunktion ist. Eine Ausgabe der "console.error" Funktion bekommt beispielsweise den Schweregrad "error". Bei der "console.log" Funktion ist dies jedoch nicht möglich, da OpenTelemetry keinen "log" Schweregrad definiert (The OpenTelemetry Authors, o. J.). Weil Konsolenausgaben mit dieser Funktion nicht immer Warnungen oder Fehlermeldungen sind, wird ihnen das "info" Level zugewiesen; wenn jedoch das Wort "error" in einer Nachricht enthalten ist, erhält sie stattdessen den Schweregrad "error". Im nächsten Schritt wird anhand einer selbst definierten Umgebungsvariable namens "LOG_LEVEL" in Kombination mit dem evaluierten Grad eines Logs bestimmt, ob es tatsächlich ausgesandt werden soll. Dazu berechnet eine separate Funktion numerische Werte anhand eines Arrays, das die Namen aller Schweregrade enthält. Der "trace" Grad wird beispielsweise als 0 bewertet und "info" als 1. In den Konsolenfunktionen werden dann die so berechneten Werte mit dem Wert von "LOG_LEVEL" verglichen. Nur wenn die Nachricht einen gleichen oder höheren Wert als die Umgebungsvariable hat, soll sie ausgesandt werden. So kann ein Schweregrad als Minimum festgelegt werden, unter dem die Nachrichten zwar auf der Konsole, jedoch nicht als Logs ausgegeben werden. Wenn beispielsweise als "LOG_LEVEL" "info" angegeben wird, werden Nachrichten mit dem "debug" Grad nicht versendet, Nachrichten der Grade "info" bis "error" hingegen schon. Im letzten Schritt wird das Log konstruiert und mit der "emit" Funktion eines Loggers des "sdk-logs" (*OpenTelemetry Logs SDK*, 2023) Moduls emittiert. Die Logs bestehen aus ihrem Schweregrad, der als "severityText" angegeben wird und aus der Nachricht mit der die Konsolenfunktion aufgerufen wurde. Zum Erzeugen des Loggers wird zunächst ein "LoggerProvider" initialisiert, der die Ressourcen der Tracing SDK kopiert, um den Logs dieselben Zusatzinformationen wie den Traces anzuhängen. Dem Logger Provider wird außerdem der "OTLPLogExporter" (*OpenTelemetry Collector Logs Exporter for Node with Grpc*, 2023) hinzugefügt. Dadurch können Logger, die von dem Provider erzeugt werden, emittierte Logs mit dem OTLP Protokoll an einen Endpunkt senden, der mit einer Umgebungsvariable festgelegt wird.

Der Code für das fertige Modul wird in einem eigenen GitLab Repository gespeichert. Eine CI/CD-Pipeline baut aus dem Modul ein Paket und speichert es in dem npm Paket-Register des Repositorys. Nach dem Installieren dieses Paketes in einem Node.js Projekt ist es dann möglich, Anwendungen mit nur einer Zeile an zusätzlichem Code zu instrumentieren. Um das Paket in den Services der Firma lokal verwenden zu können, müssen die Entwickler sich zuerst mit einem ausreichend

berechtigten “Access Token” in dem Paket-Register authentifizieren. Auch andere CI/CD-Pipelines, die für das Bauen und Ausführen der instrumentierten Firmen-Services zuständig sind, müssen sich bei dem Paket-Register authentifizieren. Durch die “Token Access” Konfiguration in den “CI/CD” Einstellungen des Repositorys, in dem sich das Instrumentierungs-Paket befindet, wird anderen Pipelines die Authentifizierung ermöglicht.

## 4.2 Pipelines

Es werden drei verschiedene Arten von Grafana Agenten konfiguriert, welche als Einheit die Pipeline des Systems bilden. Diese Aufteilung erfolgt aufgrund ihrer stark unterschiedlichen Aufgabenbereiche. Über Umgebungsvariablen werden sie angewiesen, im Flow-Modus zu starten und ihren internen API- und UI-Server offenzulegen, der sonst nicht von außerhalb der Container zu erreichen wäre.

### 4.2.1 Tracelog-Agent

Der erste Agent, der intern als “Tracelog-Agent” bezeichnet wird, agiert als Endpunkt für alle Traces und Logs, die von den Services aller Systeme und Umgebungen generiert und ausgesandt werden. Die empfangenen Daten werden anschließend von ihm verarbeitet und an die jeweiligen Backends weitergeleitet. Innerhalb des Agenten wird zuerst ein OTLP-gRPC-Endpunkt definiert, über den die Daten aufgenommen werden. Damit der Endpunkt von außerhalb des Containers erreichbar ist, wird sein Port in der Compose-Konfiguration des Agenten exponiert. Vor ihrer separaten Weiterverarbeitung werden die empfangenen Traces und Logs gepuffert. Danach wird eine Verarbeitungskomponente eingesetzt, mit dem primären Ziel, den Logs die korrekten Label zuzuweisen. Innerhalb dieser Komponente werden durch Regex Ausdrücke die Schweregrade der Logs erkannt und dann in einer Internen Map gespeichert. Werte in dieser Map sind allen weiteren Operationen innerhalb der Komponente zugänglich. Außerdem wird durch eine Ersetzung dafür gesorgt, dass die erkannten Schweregrade ausschließlich aus Kleinbuchstaben bestehen, um das spätere Filtern der Daten zu erleichtern. Mit JMESPath Ausdrücken werden die von den Services mitgesandten Informationen, also ihre Namen, Systeme und Umgebungen, aus den Logs extrahiert. Auch diese Werte speichert die interne Map der Komponente. Alle extrahierten Informationen werden dann den Logs als Label hinzugefügt. Im nächsten Schritt der Pipeline folgt eine weitere Verarbeitungskomponente, die ausschließlich dafür zuständig ist, Logs von geringem Interesse zur Reduktion des Speicheraufwands herauszufiltern. Momentan entfernt

sie lediglich Logs, deren Schweregrad "trace", "debug" oder "info" ist, da Logs mit dem "warn" oder "error" Grad der Firma wichtiger sind. Nach der Filterung werden die Logs an das Loki Backend weitergeleitet.

Auch die Container aller Bestandteile des Telemetriesystems geben über ihre Konsolen Logs aus und auch diese werden gesammelt, um Probleme des Systems nachverfolgen zu können. Mit einer Discovery Komponente macht der Tracelog-Agent alle Container, die in derselben Docker-Umgebung wie er selbst laufen, zu für ihn sichtbaren Zielen. Damit der Tracelog Agent auf das dazu benötigte "Docker Daemon Socket" zugreifen kann, wird dieses ihm in seiner Compose-Konfiguration verfügbar gemacht. Die Discovery Komponente speichert für jedes Ziel eine Liste von Werten, die es beschreiben. In diesen Listen beginnen die Namen aller Felder mit einem Präfix, das sie als "intern" markiert. Flow Komponenten entfernen Felder mit einer solchen Markierung automatisch (Grafana Labs, o. J.-a). Um die Namen der Container als Label zu persistieren wird das Präfix mit einer Relabeling Komponente entfernt. Eine Source Komponente nimmt dann alle angepassten Ziele als Quellen und sammelt ihre Logs. Die Schweregrade dieser Logs werden dann aus ihren Inhalten extrahiert und ihnen als Label angehängt. Mit einem LogQL Stream Selector Ausdruck werden Logs, bei denen kein Grad erkannt wird, aussortiert. LogQL ist eine von Grafana Labs entwickelte Sprache zum Filtern von Logs, die von der Prometheus Query Language inspiriert ist (Grafana Labs, o. J.-d). Ab diesem Punkt folgen die Docker-Container-Logs demselben Pipeline-Pfad wie jene, die über den Endpunkt empfangen werden. Das heißt, dass auch von ihnen ausschließlich jene mit Schweregraden von mindestens "warn" an das Loki Backend weitergeleitet werden.

Weil die Traces ausschließlich durch automatische Instrumentierungen von OpenTelemetry erzeugt werden, müssen sie in keiner Weise angeglichen werden. Deshalb leitet eine Exporterkomponente sie nach dem Puffern direkt an das Tempo Backend weiter.

Abbildung 6 zeigt den vollständigen Aufbau des Tracelog-Agenten. Diese Darstellung wurde automatisch durch das Debugging Interface des Agenten generiert.

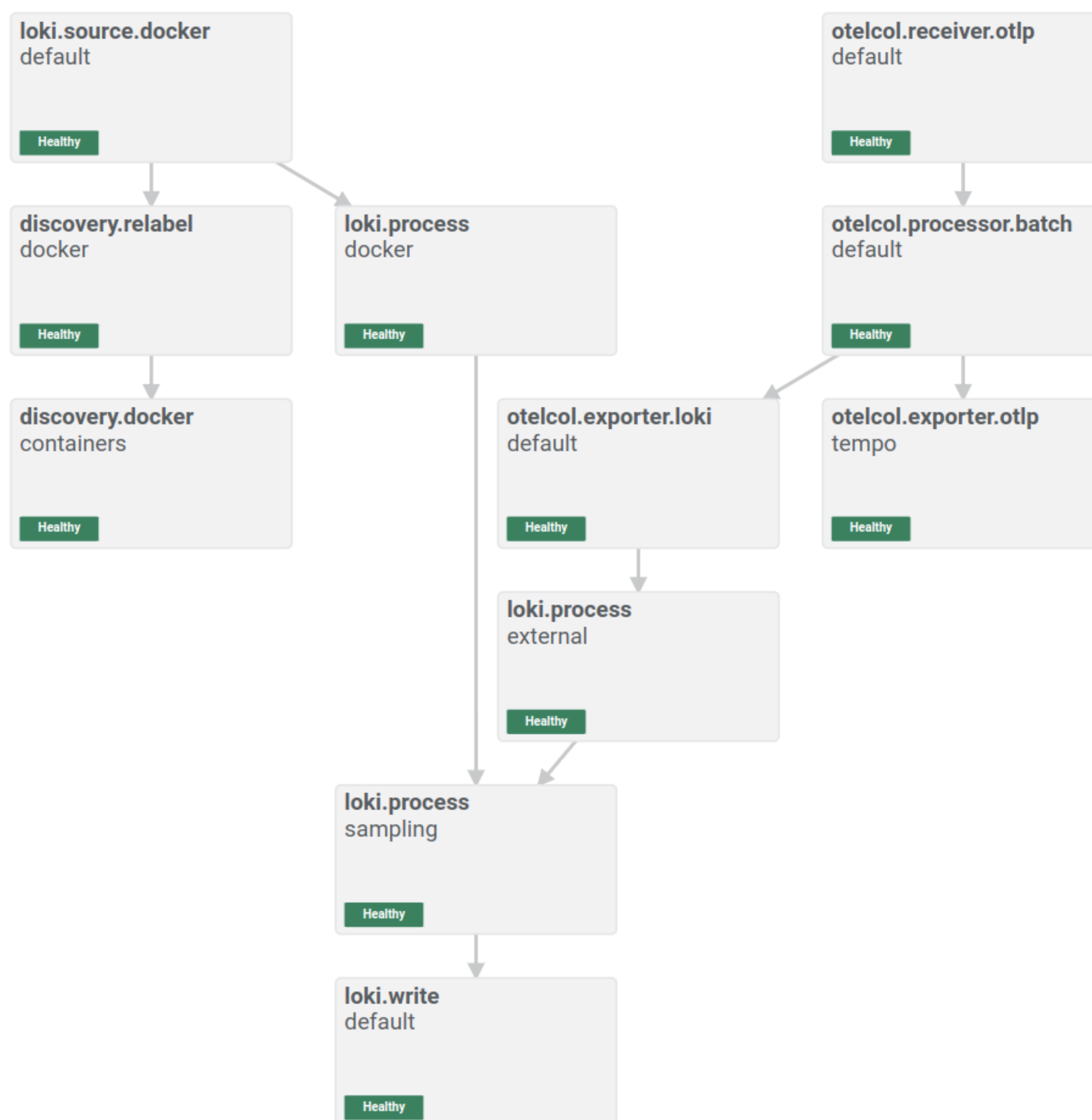


*Abbildung 6: Architektur des Tracelog-Agenten*
*(Quelle: Eigene Bildschirmaufnahme des Grafana Agent Flow UI)*

### 4.2.2 Metrics-Agent

Der intern als "Metrics-Agent" bezeichnete Grafana Agent ruft alle Endpunkte für Prometheus-Metriken auf, versieht die gesammelten Metriken mit Labeln und leitet sie an das Mimir Backend weiter. Damit der Agent die Abkürzungen für die firmeninternen Server mit ihren IP-Adressen assoziiert, werden diese in seiner Compose-Konfiguration als Hosts hinzugefügt. Für jeden Service ist eine Scrape Komponente definiert, welche die Metrik-Endpunkte aller seiner Instanzen aufruft. Wenn also ein Service gleichzeitig in mehreren Umgebungen ausgeführt wird, auch

wenn diese über mehrere Server verteilt sind, sammelt eine einzige Komponente seine Metriken. Dabei werden den Metriken direkt in der Definition der Ziele manuell Label zugewiesen. Auch die von den cAdvisor Instanzen generierten Metriken werden gesammelt, müssen aber verarbeitet werden, um die richtigen Label zu erhalten. Durch die Persistierung von cAdvisor haben Label ein unerwünschtes Präfix, welches mithilfe einer Relabeling Komponente entfernt wird. Diese Komponente sortiert außerdem jene Metriken aus, bei denen ein Regex Ausdruck keines der Firmen-Label erkennt. So ist sichergestellt, dass nur Metriken mit korrektem Labeling in die Backends gelangen. Abgesehen von den Endpunkten der Firmen-Services, werden auch jene der Exporter und Services des Telemetriesystems aufgerufen. Der Flow Agent bietet mittlerweile eingebettete Integrationen für einen Großteil der von der Peterson Technologies GmbH eingesetzten Metrik-Exporter an. Um die Nutzung solcher Integrationen zu testen, wird im Rahmen dieser Arbeit die Blackbox-Exporter Komponente implementiert. Die Blackbox Konfigurationsdatei des vorherigen Telemetriesystems kann unverändert weiterhin genutzt werden. Dazu wird sie dem Container in seiner Compose-Konfiguration zur Verfügung gestellt, im Agenten geladen und dann der Komponente weitergereicht. Als Ziele werden vorerst drei der damaligen Endpunkte abgefragt. Jedes der Ziele bekommt einen Namen, der sich aus dem System und der Umgebung des Endpunktes zusammensetzt. Eine Relabeling Komponente extrahiert dann System und Umgebung aus dem Namen und fügt sie den Metriken als Label hinzu. Die Relabeling Komponente benennt außerdem die Ziel- und Modul-Label der Blackbox-Integration um, da diese durch ein Präfix als intern markiert sind und sonst verloren gingen. Eine Scrape Komponente sammelt dann die Metriken der Blackbox. Alle Metriken werden an das Mimir-Backend weitergeleitet.

Abbildung 7 zeigt eine grobe Nachbildung der Architektur des Metrics-Agenten. Der gesamte Graph wäre zu groß, um hier dargestellt werden zu können und würde Einblick in das System der Peterson Technologies GmbH bieten.

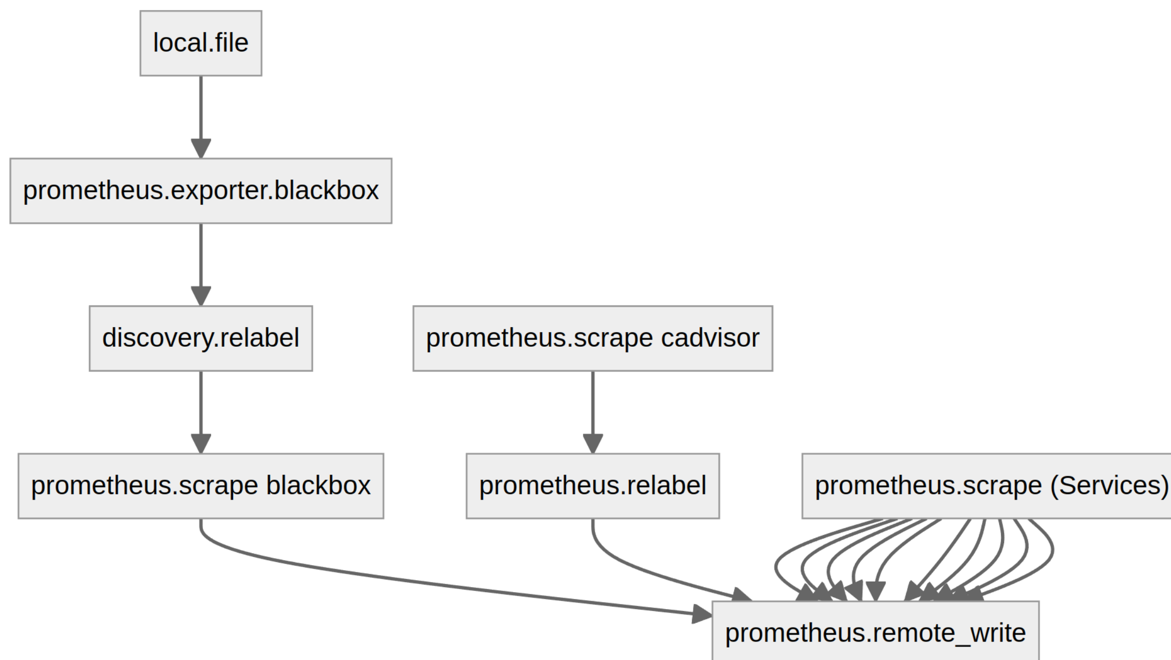


*Abbildung 7: Architektur des Metrics-Agenten*
*(Quelle: Eigene Darstellung, generiert mit Mermaid in Markdown)*

### 4.2.3 External-Agent

Die dritte Art von Grafana Agent, die für das System entwickelt wird, trägt den Namen "External-Agent" und sammelt Logs von Containern, die sich nicht in der Docker-Umgebung des Telemetrie-Servers befinden. Weil dazu Zugriff auf das "Docker Daemon Socket" notwendig ist, wird eine Instanz dieses Agenten auf jedem Server mit containerisierten Services ausgeführt. Es wäre zwar möglich, das Socket über Netzwerke erreichbar zu machen, doch davor warnen die Entwickler von Docker aus Sicherheitsgründen (Docker Inc., o. J.-a). Der External-Agent verwendet dieselbe Architektur wie der Tracelog-Agent zum Erkennen der Container und zum Sammeln ihrer Logs. Der einzige Unterschied besteht darin, dass im External-Agent keine separate Verarbeitungskomponente zum Filtern implementiert wird, da nicht mehrere Ströme von Logs separat verarbeitet werden. Abbildung 8 zeigt die Architektur der External-Agenten.

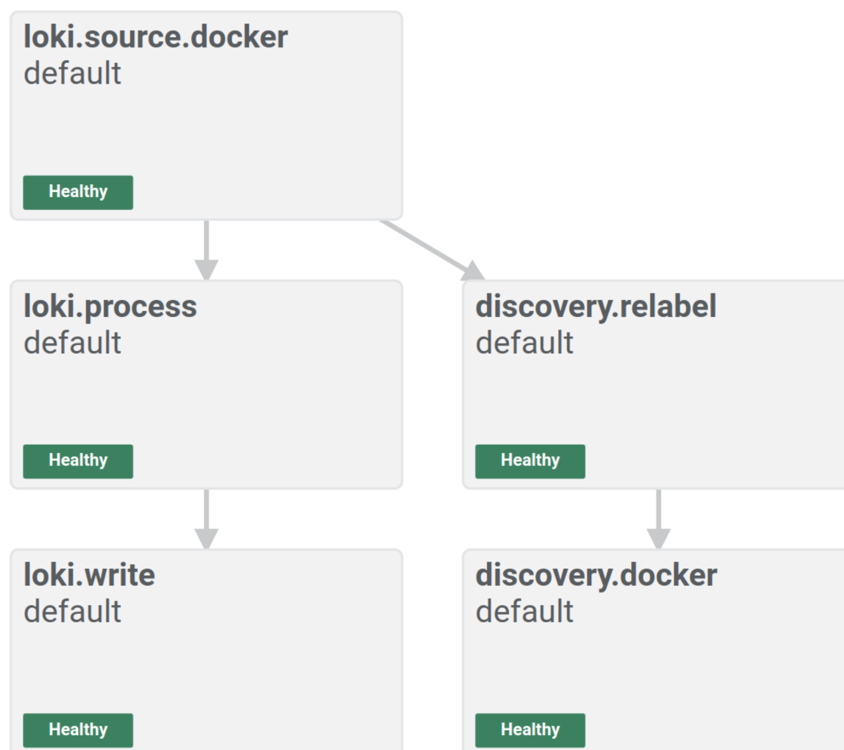


*Abbildung 8: Architektur der External-Agenten*
*(Quelle: Eigene Bildschirmaufnahme des Grafana Agent Flow UI)*

## 4.3 Backends

Die Konfigurationen der Backends (mit YAML-Dateien) sind so minimal wie möglich gehalten. Als Speicher wird die Adresse der MinIO Instanz inklusive benötigter Authentifizierungsdaten angegeben. Zur Speicherdefinition gehören auch die Namen der Buckets, in denen die Services ihre Daten ablegen sollen. Diese werden nach den Backends benannt, also "loki", "mimir" und "tempo". Die drei Backends bieten alle die Möglichkeit an, eine sogenannte "Retention" einzustellen. (Grafana Labs, o. J.-d, o. J.-e, o. J.-f). Damit ist ein Zeitintervall gemeint, nach dem die gespeicherten Daten automatisch wieder gelöscht werden. Bei Loki und Mimir ist eine Retention standardmäßig nicht aktiviert, was bedeutet, dass Logs und Metriken ohne weitere Konfiguration nicht bereinigt werden. Bei Tempo beträgt die Standard-Dauer 14 Tage. Auf Wunsch der Geschäftsführung der Peterson Technologies GmbH wird die Retention für Logs und Traces auf vier Wochen gesetzt und für Metriken auf ein Jahr.

Loki ist der einzige Backend Service, der in seiner Compose-Konfiguration einen Port nach außen exponieren muss. Das ist notwendig, damit er die Logs der

External-Agenten empfangen kann. Mimir und Tempo erhalten ihre Daten ausschließlich von den Tracelog- und Metrics-Agenten, die sich in derselben Docker-Umgebung befinden wie sie selbst.

Es ist in der Mimir Konfiguration vorgesehen, unterschiedliche Speicherorte für die “Blocks” und den momentanen Zustand des “Rulers” anzugeben (Grafana Labs, o. J.-e). Blocks enthalten Metriken und der Ruler ist eine optionale Mimir Komponente, die wie Prometheus Regeln auswertet. In dieser Arbeit wird die Ruler Komponente zwar nicht verwendet, doch für den Fall, dass sie in der Zukunft eingesetzt werden sollte, wird die Speichertrennung bereits vorgenommen. Dazu können entweder in der Object Storage mehrere Buckets für Mimir angelegt oder unterschiedliche Präfixe für die Daten vergeben werden. Vorerst bekommen die Daten des Rulers den “ruler”- und die Blocks den "blocks"-Präfix, zur Lösung des Konfliktes. Aufgrund von Fehlermeldungen bezüglich Überschreitungen der voreingestellten maximalen Menge an Labeln pro Metrik wurde dieses Limit auf 40 Label erhöht.

Tempo kann zusätzlich zum Speichern von Traces auch Metriken von Traces generieren. Dazu wird in der Tempo Konfiguration der “Metrics Generator” definiert und zum Speichern der Metriken der Endpunkt der Mimir Instanz angegeben. Die Trace-Metriken werden durch zwei verschiedene Prozessoren erzeugt. Der “Service Graphs” Prozessor analysiert Traces, um deren Kanten, also Verbindungen zwischen Services, zu finden (Grafana Labs, o. J.-f). Der “Span Metrics” Prozessor generiert “RED” (Request, Error, Duration) Metriken von Spans (Grafana Labs, o. J.-f). Dadurch können Anfrageraten und die Fehleranfälligkeit von Verbindungen untersucht werden.

Um die gespeicherten Daten zu persistieren, verwendet die MinIO Instanz ein Docker Volume. In ihrer Compose-Konfiguration erzeugt sie die Bucket Speicher, sofern diese noch nicht vorhanden sind und startet ein Webinterface. Damit das Interface außerhalb des Containers erreichbar ist, wird sein Port exponiert. Die Logindaten für MinIO, welche sowohl für den Zugriff durch die Backends als auch für das Login in das Webinterface dienen, werden über Umgebungsvariablen konfiguriert. Dort wird auch die Authentifizierungsmethode für das Abrufen der Metriken von MinIO festgelegt. Diese ist voreingestellt, ein JWT (JSON Web Token) zu verwenden, doch da sich der Telemetrie-Server in einem sicheren Perimeter befindet, wird vorerst keine Authentifizierung eingesetzt.

## 4.4 Grafana

### 4.4.1 Konfiguration

Die Grafana Instanz wird mit einer INI-Datei konfiguriert (Grafana Labs, o. J.-c), die in der Compose-Konfiguration in den Grafana Container geladen wird. Einstellungen, die als Dateien von Grafana eingelesen werden, gelten als provisioniert, was bedeutet, dass sie nicht über Grafanas Webinterface geändert werden können. In der INI-Datei werden unter anderem die URL der Web-UI, der Datenspeicher (über die Adresse und Login-Daten zu einer Postgres Instanz) und die Methode der Authentifizierung festgelegt. Der Port der Web-UI wird nach außen exponiert. In der Konfiguration der Postgres Instanz selbst werden lediglich ihre Login-Daten, der Name der Standard-Datenbank (hier "grafana") und ein Docker Volume, zur Persistierung der Daten, definiert. Die reguläre Login-Lösung von Grafana wird zusammen mit dem Login-Formular und der Möglichkeit für Nutzer, sich manuell Konten anzulegen, abgeschaltet. Zum Einloggen wird stattdessen "Generic OAuth" verwendet, mithilfe des Keycloak Authentifizierungs-Systems der Firma. Dazu werden die Adressen unter denen Keycloak im Firmennetz erreichbar ist und die Autorisierungsdaten von Grafana als Client konfiguriert. In Keycloak werden die Nutzer-Rollen von Grafana als "Realm Roles" unter gleichem Namen angelegt. Über die "role_attribute_path" Einstellung in der Grafana Konfiguration, werden die in Keycloak zugewiesenen Rollen automatisch auf alle Nutzer der Firma übertragen, die sich in Grafana anmelden. Zusätzlich wird festgelegt, dass sich nur Nutzer einloggen können, denen auch tatsächlich in Keycloak mindestens eine Grafana Rolle zugewiesen wurde. Damit Grafana Benachrichtigungen aussenden kann, wird das "SMTP" (Simple Mail Transfer Protocol) der Firma konfiguriert. Es wird derselbe Mail-Server eingesetzt, der bereits von der Alertmanager Instanz des vorherigen Systems verwendet wurde. Um seine lokale Adresse angeben zu können, wird in Grafanas Compose-Konfiguration das Host-Gateway als zusätzlicher Host namens "host.docker.internal" definiert. Dadurch kann "host.docker.internal" als Äquivalent zu "localhost" verwendet werden (Docker Inc., o. J.-b). Damit Grafana auf die gesammelten Telemetriedaten zugreifen kann, benötigt es alle Backends als "Datasources", also Datenquellen. Diese werden mit Namen, Typen und Adressen mit der "datasources.yaml"-Datei provisioniert. Zusätzlich werden über die Datenquellen die "Logs to Traces" und "Traces to Logs" Konzepte umgesetzt. Diese beschreiben die Verlinkung zwischen den Daten, um zu ermöglichen, über das Grafana Interface von Logs zu assoziierten Traces zu springen und andersherum (Grafana Labs,

o. J.-c). Das “Logs to Traces” Konzept wird durch die “Derived Fields” der Loki Datenquelle implementiert. Diese definieren das Erzeugen von Abfragen auf das Tempo Backend (Grafana Labs, o. J.-c), basierend auf Werten, die sich in den Logs befinden, wie beispielsweise die “traceID”. Das “Traces to Logs” Konzept wird durch die “tracesToLogs” Sektion in der Konfiguration der Tempo-Datenquelle integriert, in der die gewünschten Assoziationen zwischen Traces und Logs hergestellt werden. Außerdem wird in der Tempo-Datenquelle Mimir als Referenz für die Service-Graph Metriken angegeben, damit Grafana die Graphen darstellen kann.

### 4.4.2 Visualisierung und Alarme

Grafana speichert Dashboards, also Sammlungen von Visualisierungen, in einem eigenen JSON-Datenmodell (Grafana Labs, o. J.-c). Es ist möglich, das Modell eines Dashboards über das Grafana Interface anzeigen zu lassen und es ist auch möglich, Dashboards aus JSON-Dateien zu importieren. Alle Visualisierungen des vorherigen Systems könnten also theoretisch direkt in das neue System kopiert werden. Seit der Implementierung der Dashboards des vorherigen Systems wurden die Möglichkeiten der Visualisierung jedoch von Grafana Labs kontinuierlich aktualisiert und erweitert. Außerdem unterscheiden sich die Architekturen der beiden Systeme. Es werden jetzt andere (Mimir statt Prometheus) und zusätzliche (Tempo) Datenquellen verwendet. Teilweise haben die Daten andere Formate durch die Verarbeitungsschritte der Grafana Agenten. Des Weiteren verfügt das neue System insgesamt über mehr Daten, zum Beispiel durch die cAdvisor Instanzen und die Überwachung des Telemetriesystems. Aus diesen Gründen werden die Visualisierungen nicht unverändert kopiert, dienen aber dennoch als Grundlage für eine vollständige Überarbeitung. Im Rahmen der Überarbeitung werden Gespräche mit den Entwicklern der Firma darüber geführt, welche der ursprünglichen Dashboards übernommen und welche Änderungen an ihnen vorgenommen werden sollen. Während und nach Anpassungsschritten wird dabei auf die Rückmeldungen der Entwickler reagiert.

Beim Übertragen und Anpassen der Dashboards wird versucht, eine höhere visuelle Klarheit als bei den vorherigen Visualisierungen zu erzielen. Dazu wird in den neuen Dashboards vor allem auf Konsistenz geachtet, beispielsweise bei der Sortierung von Metriken nach dem Charakter ihrer Werte. Metriken wie die CPU-Auslastung von Services werden zum Beispiel absteigend sortiert, da ein hoher Wert für das System kritischer ist als ein niedriger. Zu den Anpassungen, die an der Struktur der Dashboards vorgenommen werden, zählt unter anderem die Nutzung der Optionen

zum Wiederholen von Zeilen und Feldern. Diese Optionen ermöglichen dynamisch Felder und Zeilen zu Dashboards hinzuzufügen oder zu entfernen, basierend auf den vorhandenen Daten. Dadurch müssen sie nicht für jeden Service und jede Umgebung einzeln definiert werden, sondern können nach einmaliger Konfiguration automatisch generiert werden. In allen Dashboards werden Filter für Systeme, Umgebungen und Services hinzugefügt, um gezielte Vergleiche zu erleichtern. Abbildung 9 zeigt einen Ausschnitt des "Service Details" Dashboards für den "analysis" Service. Es werden außerdem aufgrund der neuen Daten und Dienste, welche die Firma seit der Entwicklung der vorherigen Visualisierungen zu ihrem System hinzugefügt hat, einige vollständig neue Dashboards entwickelt. Dazu gehören unter anderem Dashboards für die Leistung von Open Policy Agenten (siehe Abbildung 10) und des Telemetriesystems. Einigen Dashboards, wie der Übersicht des Telemetriesystems, werden Links zur offiziellen Dokumentation ihrer Metriken hinzugefügt. Die Dashboards werden zum besseren Überblick in einer neuen Ordnerstruktur sortiert, in den Kategorien "Server", "Services", "Hilfsdienste" und "Telemetriesystem".

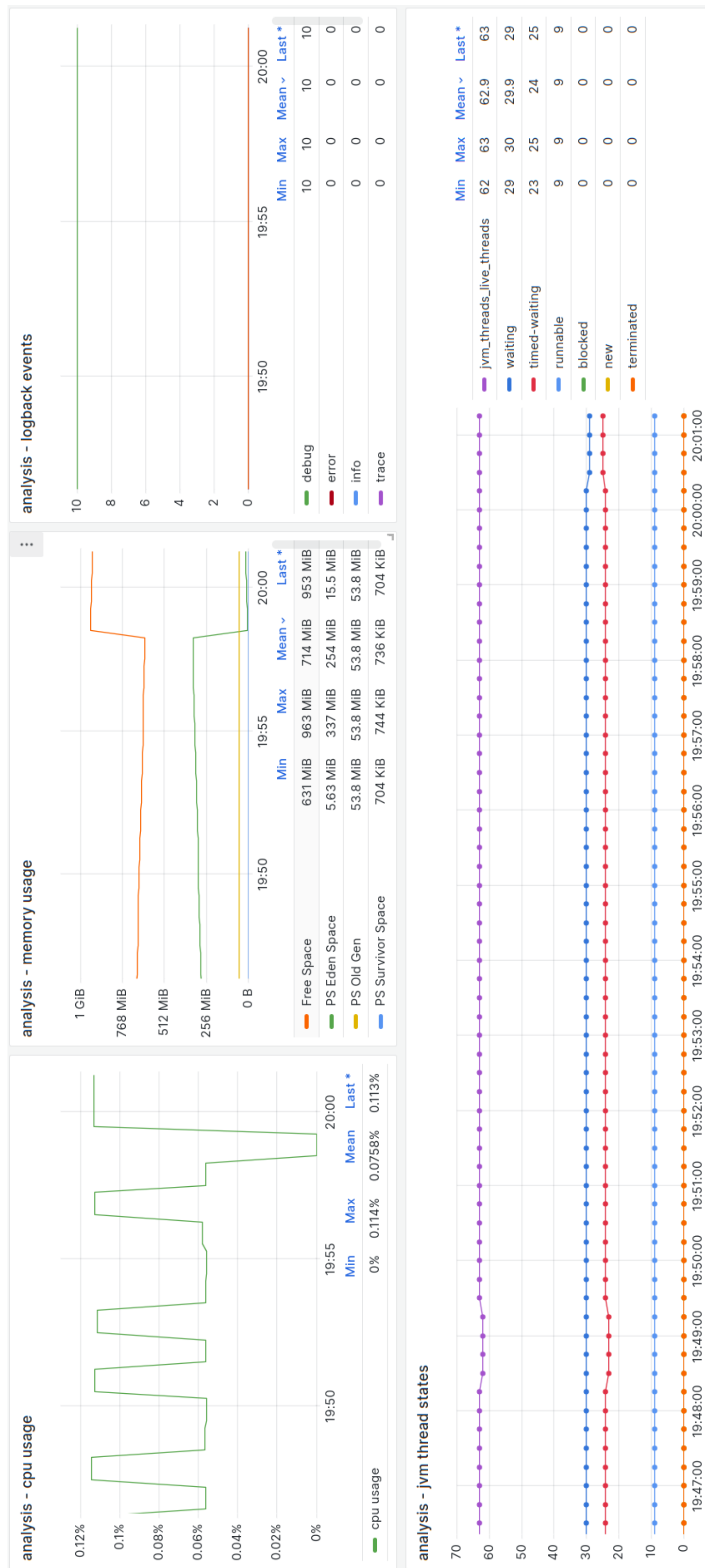


*Abbildung 9: Ausschnitt des "Service Details" Dashboards (Quelle: Eigene Bildschirmaufnahme)*

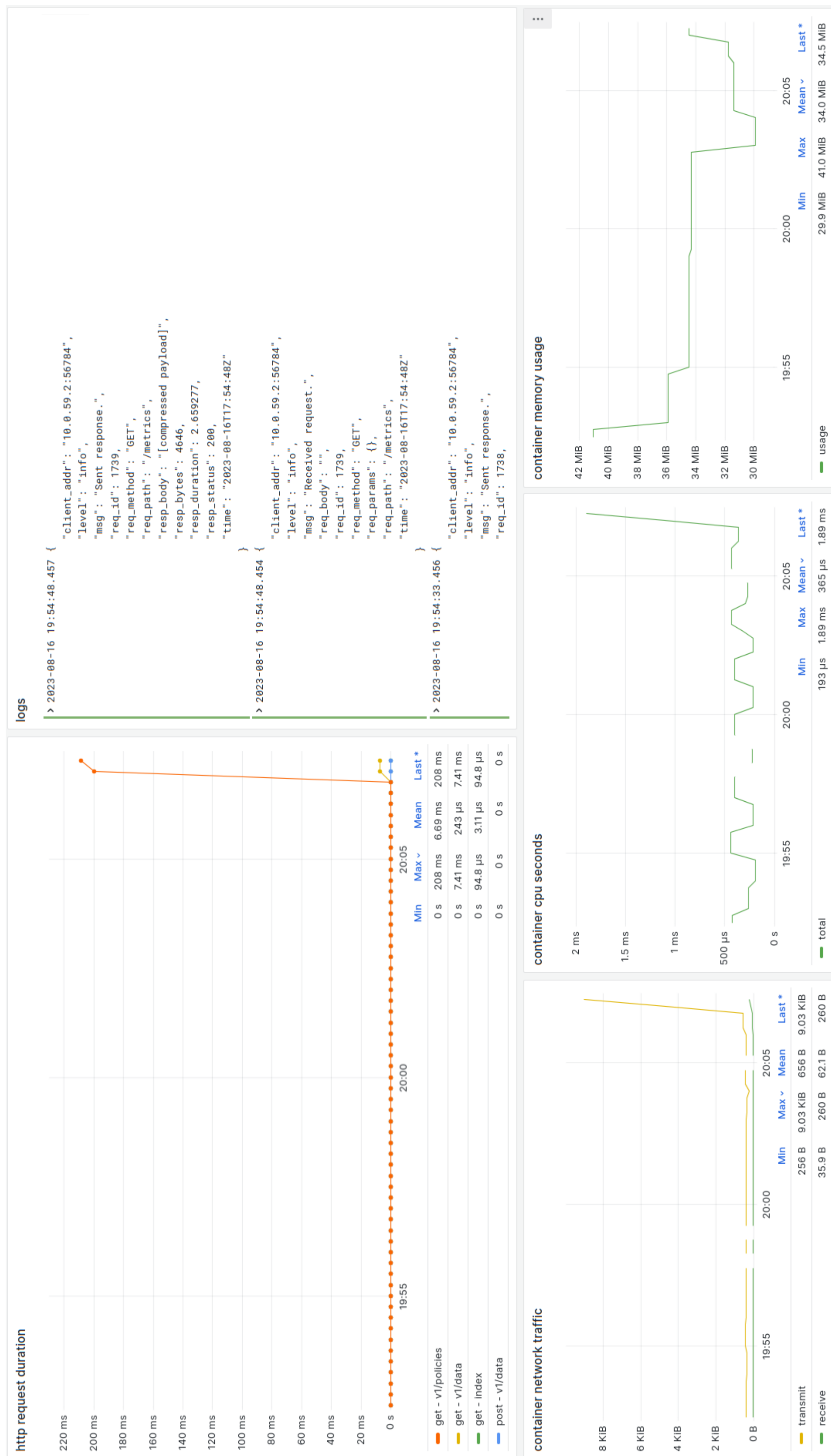


*Abbildung 10: Open Policy Agent Dashboard (Quelle: Eigene Bildschirmaufnahme)*

Die Alarme des vorherigen Systems sollen mit identischer Funktionalität übertragen werden. Da nun statt Prometheus und einem Alertmanager für die Alarme ausschließlich Grafana verwendet wird, kann das Grafana Interface zur Implementierung genutzt werden. Als "Contact Points" werden die Benachrichtigungskanäle konfiguriert, also die Adressen und Anwendungen, an die Grafana Nachrichten über ausgelöste Alarme versendet. Hier werden die allgemeine E-Mail Adresse und die Microsoft Teams Webhooks der Firma eingetragen. Durch Testfunktionen, die direkt im Grafana Interface integriert sind, wird bestätigt, dass die Benachrichtigungen ihre Ziele erreichen.

## 4.5 Dokumentation

Die Dateien und Dokumentation des Telemetriesystems befinden sich zusammen mit mehreren anderen Systemen in einem GitLab Repository der Firma, welches als eine Art Informationszentrum dient. Die Dokumentation ist in der englischen Sprache, der Firmensprache, verfasst.

An der Oberfläche der Dokumentation liegt eine allgemeine Übersicht der Architektur des Systems, zusammen mit einer Zusammenfassung aller verwendeten Technologien. Die Übersicht wird durch ein Mermaid Markdown Diagramm visualisiert. Am Fuß der Seite befindet sich ein Verzeichnis von Links zu weiteren Dokumentationsseiten. Zu diesen zählen nicht nur kurze Erklärungen der Systemkomponenten, sondern auch Hinweise zum allgemeinen Gebrauch des Systems. Es existieren beispielsweise auch Listen von Docker-Befehlen und Beschreibungen der Erzeugung von Abfragen in Grafana. Auch die Instrumentierung mit dem OpenTelemetry Java Agent und dem selbst entwickelten Node.js Modul wird dokumentiert. In der Dokumentation einiger Systemkomponenten werden weitere Mermaid Diagramme eingesetzt, um ihre Interaktion mit anderen Komponenten darzustellen. An erster Stelle jeder Seite, und immer dann, wenn Erklärungen zur erhöhten Lesbarkeit stark zusammengefasst sind, werden Verlinkungen zu offiziellen Dokumentationen eingefügt.

# 5 Evaluation

Alle Funktionen, die für das neue Telemetriesystem geplant waren, wurden implementiert, von der Instrumentierung der Services bis zur Visualisierung der Daten und dem Versenden von Benachrichtigungen basierend auf der Auswertung von Alarmen. Außerdem wird jetzt insgesamt eine größere Menge an Daten erzeugt,

durch die Implementierung von Traces, cAdvisor und dem neuen Node.js Instrumentierungsmodul. Dabei sind keine der Fähigkeiten des vorherigen Systems verloren gegangen. Durch die gezielte Planung der Architektur und Reduktion der Anzahl von Hilfs-Services wie Promtail, Prometheus, dem Alertmanager und dem Blackbox-Exporter, ist das System kompakter und homogener geworden. Die Wahl des Grafana Agenten als Pipeline und die vielfältigen Konfigurationsmöglichkeiten der Instrumentierungen und Backends machen es außerdem deutlich flexibler als das vorherige System. Auch die Geschäftsführung der Peterson Technologies GmbH bestätigt den Erfolg des Projektes.

Mehrere Umgebungen der Firma, unter anderem die Produktionsumgebung, sind noch nicht mit der neuen Instrumentierung versehen. Daher wird das vorherige System vorerst weiterhin parallel verwendet. Eine oberflächliche Analyse der Leistung des Systems mit Hilfe eines Grafana Dashboards zeigt, dass die CPU Auslastung des Tracelog Agenten durchschnittlich deutlich höher ist, als die aller anderen Systemkomponenten zusammen (siehe Abbildung 11). Diese Auslastung ist dadurch, dass einige Umgebungen noch nicht instrumentiert sind, besonders alarmierend. Beim Ausführen einiger einfacher Abfragen auf das Loki Backend ist zu erkennen, dass einige Services konstant Warnungen und Error aussenden. Das System ist jedoch auch nach längerer Beobachtung über mehrere Wochen hinweg nicht zusammengebrochen.

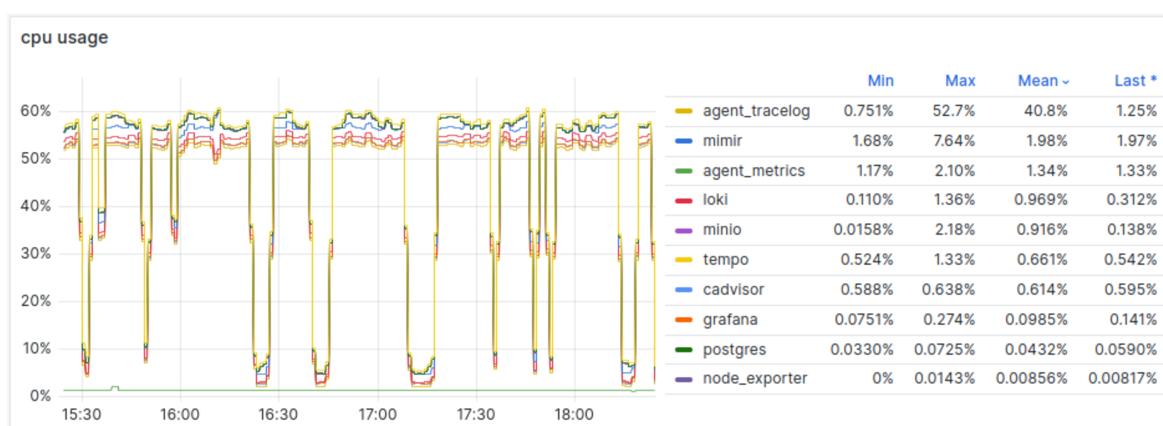


*Abbildung 11: Hohe CPU-Auslastung des Tracelog-Agenten (Quelle: Eigene Bildschirmaufnahme)*

# 6 Zusammenfassung und Ausblick

## 6.1 Zusammenfassung

Das Ziel dieser Arbeit war, einen neuen Telemetriestack für die Firma Peterson Technologies GmbH zu entwickeln. Dabei war der Firma wichtig, dass der neue Stack alle Funktionen ihres vorherigen Telemetriesystems weiterhin unterstützt. Dazu zählte auch, alle Visualisierungen und Alarme in das neue System zu übertragen. Außerdem sollte das System um Tracing erweitert werden. Vor der Planung und Umsetzung wurden einige Prioritäten festgelegt. Dazu zählte beispielsweise, die Menge der Eingriffe in den Code der Anwendungen der Firma so niedrig wie möglich zu halten. Durch diese Prioritäten wurde versucht, die spätere Nutzung, Wartung und Erweiterung des Systems zu erleichtern.

Basierend auf den Komponenten des vorherigen Systems und den gesetzten Zielen und Prioritäten wurde die Architektur des neuen Stacks geplant. Als Methode der Orchestrierung wurde Docker verwendet, da Docker bereits in der Firma in Gebrauch war. Während der Planungsphase wurden kontinuierlich Gespräche mit den Entwicklern der Firma geführt, um sicherzustellen, dass die getroffenen Entscheidungen ihren Vorstellungen entsprachen. Die Auswahl fiel überwiegend auf Lösungen von OpenTelemetry und Grafana Labs, da diese zusammen den Großteil des Systems abdecken konnten und teilweise bereits von der Firma genutzt wurden. Zur Instrumentierung der Java Services der Firma wurde der Java Agent von OpenTelemetry genutzt, vor allem wegen der Möglichkeit der automatischen Instrumentierung. Um Node.js Services mit einem einzigen Funktionsaufruf für Logs, Metriken und Traces instrumentieren zu können, wurde ein eigenes Modul entwickelt. Um zusätzliche Metriken für Containerisierte Anwendungen zu generieren, wurden cAdvisor Instanzen eingesetzt. Drei unterschiedlich konfigurierte Arten von Grafana Agenten bilden die Pipeline des Telemetriesystems. Das Aufteilen der Pipeline diente der deutlichen Trennung ihrer Aufgabenbereiche. Die Agenten wurden, um erhöhte Flexibilität zu erreichen, mit der "River" Sprache im "Flow Modus" konfiguriert und ausgeführt. Als Backends wurden Mimir, Loki und Tempo von Grafana Labs verwendet, welche alle im "monolithischen Modus " ausgeführt wurden, um den Konfigurationsaufwand vorerst so gering wie möglich zu halten. Sie speichern ihre Daten mit einer MinIO Instanz, weil die Grafana Labs Backends darauf ausgelegt sind, "Object Storage" zu verwenden. Zur Visualisierung der Daten wurde weiterhin Grafana verwendet. Auch wurde weiterhin wie im vorherigen System eine

Postgres Instanz zum Speichern der Benutzerdaten und Visualisierungen von Grafana verwendet. Aufgrund von Neuerungen in Grafana und einiger Unterschiede zwischen dem vorherigen und dem neuen System wurden die Visualisierungen nicht direkt übertragen. Stattdessen wurden die vorherigen Dashboards als Ausgangspunkt genutzt, um mit Rückmeldungen von den Entwicklern der Firma neue Dashboards zu entwerfen. Es wurde entschieden, statt einer Prometheus und einer Alertmanager Instanz vorerst Grafana für alle Alarme zu verwenden, unter anderem aufgrund der Möglichkeit, mehrere Datenquellen gleichzeitig verwenden zu können. Die Alarme wurden dann direkt über das Grafana Interface implementiert. Eingesetzte Technologien wurden im GitLab Wiki desselben Repositorys dokumentiert, in dem die Dateien des Systems gespeichert wurden. Zur Visualisierung der Abhängigkeiten und Interaktionen des Systems wurden Mermaid Diagramme verwendet. Außerdem wurden Links zu offiziellen Dokumentationen eingefügt.

Der resultierende Telemetriestack ist von der Instrumentierung der Services bis zur Visualisierung der Daten und dem Aussenden von Alarm Benachrichtigungen funktionstüchtig. Alle gesetzten Ziele sind erreicht und alle Prioritäten eingehalten worden. Allerdings weist die Implementierung dennoch einige Schwächen auf. Die Instrumentierung einiger Umgebungen der Firma steht noch aus, weshalb das alte Telemetriesystem vorerst parallel weiter genutzt wird. Die CPU-Auslastung des Tracelog-Agenten ist vor allem in Anbetracht der fehlenden Instrumentierungen sehr hoch. Des Weiteren gibt das System einen konstanten Strom von Fehlermeldungen aus.

## 6.2 Ausblick

Es gibt verschiedene Schritte, die in der Zukunft zur Stabilisierung und Erweiterung des Systems vorgenommen werden können. Bevor das vorherige System abgeschaltet und allein das Neue verwendet werden kann, müssen alle restlichen Umgebungen der Peterson Technologies GmbH instrumentiert werden. Dann wäre es auch möglich, unter voller Auslastung detaillierte Analysen bezüglich der CPU-, Arbeitsspeicher- und Speicherauslastung des Systems durchzuführen. Basierend auf den Ergebnissen solcher Untersuchungen könnten die Konfigurationen der Systemkomponenten verfeinert werden. Eine Möglichkeit der Entlastung wäre gegebenenfalls, die Backends im Microservices Modus auszuführen und experimentell die optimale Menge paralleler Instanzen ihrer Unterkomponenten zu

bestimmen. Es könnte auch untersucht werden, welche der generierten Metriken und Logs für die Firma wenig oder nicht von Interesse sind, um das Filtern der Daten strenger zu gestalten und so den Speicheraufwand zu reduzieren. Sollte das System trotz weiterer Optimierungen Leistungsprobleme aufweisen, müssten dann entweder Kompromisse in der Generierung und Verarbeitung der Daten gefunden oder der Telemetrie-Server mit leistungsfähigerer Hardware versehen werden.

Um den Umfang des Systems weiter zu reduzieren, können die restlichen Exporter, die auf den Servern der Firma laufen, durch die Integrationen des Grafana Agenten ersetzt werden. Dazu würde es sich anbieten, eine vierte Art von Agent zu konfigurieren, die ausschließlich für das Erzeugen von Metriken lokaler Services durch eingebettete Exporter zuständig ist. Auch eine cAdvisor Integration für den Grafana Agenten ist laut dem "v0.36.0" Meilenstein von Grafana Labs vorgesehen (*V0.36.0*, o. J.) und könnte nach seiner Veröffentlichung sofort in den "Exporter-Agent" integriert werden. Die Struktur aller Agenten könnte durch das Modulsystem überarbeitet werden, welches Grafana Labs mit der Version "0.33.0" veröffentlichte (*V0.33.0*, o. J.) und das erlaubt, Komponenten und ganze Pipelines in Dateien auszulagern.

Spätestens wenn die Peterson Technologies GmbH ihre Anwendungen, wie von der Geschäftsführung geplant, vollständig mit Kubernetes orchestriert, werden die Konfigurationen des Telemetriesystems angepasst werden müssen. Sobald das System aktiv genutzt wird, kann durch weitere Tests mit den Entwicklern der Firma festgestellt werden, ob die Dashboards, Visualisierungen und Filtermöglichkeiten ihren Vorstellungen entsprechen. Um die Versionskontrolle der Visualisierungen und Alarme durch GitLab zu ermöglichen, können diese nach ausgiebigem Testen so wie die Datenquellen provisioniert werden. Die Konstruktion einer GitLab-Pipeline könnte den Entwicklern der Firma erlauben, das System ohne tieferes Verständnis über die verwendeten Technologien fernzusteuern.

# Literaturverzeichnis

Adiscon GmbH. (o. J.). *The rocket-fast Syslog Server*. The Rocket-Fast Syslog Server - Rsyslog. Abgerufen 7. August 2023, von https://www.rsyslog.com/

Ahmed, T. M., Bezemer, C.-P., Chen, T.-H., Hassan, A. E., & Shang, W. (2016). Studying the effectiveness of application performance management (apm) tools for detecting performance regressions for web applications: An experience report. *Proceedings of the 13th International Conference on Mining Software Repositories*, 1–12.

Al Jawarneh, I. M., Bellavista, P., Bosi, F., Foschini, L., Martuscelli, G., Montanari, R., & Palopoli, A. (2019). Container orchestration engines: A thorough functional and performance comparison. *ICC 2019-2019 IEEE International Conference on Communications (ICC)*, 1–6.

Al Maruf, A., Bakhtin, A., Cerny, T., & Taibi, D. (2022). Using microservice telemetry data for system dynamic analysis. *2022 IEEE International Conference on Service-Oriented System Engineering (SOSE)*, 29–38.

Amazon Web Services, Inc. (o. J.). *Amazon S3 pricing*. Amazon S3 Simple Storage Service Pricing - Amazon Web Services. Abgerufen 12. August 2023, von https://aws.amazon.com/s3/pricing/

Aniszczyk, C. (2012, Juni 7). *Distributed Systems Tracing with Zipkin*. Distributed Systems Tracing with Zipkin. https://blog.twitter.com/engineering/en_us/a/2012/distributed-systems-tracing-with-zipkin

Bento, A., Correia, J., Filipe, R., Araujo, F., & Cardoso, J. (2021). Automated analysis of distributed tracing: Challenges and research directions. *Journal of Grid Computing*, *19*, 1–15.

Beverly, R. (2002). RTG: A Scalable SNMP Statistics Architecture for Service Providers. *LISA*, 167–174.

Bifet, A. (2013). Mining big data in real time. *informatica*, *37*(1), 15–20.

Birrell, A. D., & Nelson, B. J. (1984). Implementing remote procedure calls. *ACM Transactions on Computer Systems (TOCS)*, *2*(1), 39–59. https://doi.org/10.1145/2080.357392

Cantrill, B. (2006). Hidden in Plain Sight: Improvements in the Observability of Software Can Help You Diagnose Your Most Crippling Performance Problems. *Queue*, *4*(1), 26–36. https://doi.org/10.1145/1117389.1117401

Case, J. D., Fedor, M., Schoffstall, M. L., & Davin, J. R. (1989). *Simple network management protocol (SNMP)* (RFC 1157; Request for Comments). Network Working Group. https://www.rfc-editor.org/info/rfc1157

*Cloud Storage pricing*. (o. J.). Pricing | Cloud Storage | Google Cloud. Abgerufen 12. August 2023, von https://cloud.google.com/storage/pricing

Copperman, M. (1992). *Producing an Accurate Call-Stack Trace in the Occasional Absence of Frame Pointers* (UCSC-CRL-92-25; Board of Studies in Computer and Information Sciences). University of California at Santa Cruz.

Cruz, R., Guimarães, T., Peixoto, H., & Santos, M. F. (2021). Architecture for Intensive Care Data Processing and Visualization in Real-time. *Procedia Computer Science*, *184*, 923–928.

Datadog. (o. J.). *Modern monitoring & security*. Cloud Monitoring as a Service | Datadog. Abgerufen 5. August 2023, von https://www.datadoghq.com/

Docker Inc. (o. J.-a). *Configure remote access for Docker daemon*. Configure Remote Access for Docker Daemon | Docker Documentation. Abgerufen 16. August 2023, von https://docs.docker.com/config/daemon/remote-access/

Docker Inc. (o. J.-b). *Dockerd*. Dockerd | Docker Documentation. Abgerufen 16. August 2023, von https://docs.docker.com/engine/reference/commandline/dockerd/

Docker Inc. (o. J.-c). *Overview of Docker Compose*. Overview of Docker Compose | Docker Documentation. Abgerufen 12. August 2023, von https://docs.docker.com/compose/

Dragoni, N., Giallorenzo, S., Lafuente, A. L., Mazzara, M., Montesi, F., Mustafin, R., & Safina, L. (2017). Microservices: Yesterday, Today, and Tomorrow. In *Present and Ulterior Software Engineering* (S. 195–216). Springer International Publishing.

Dragoni, N., Lanese, I., Larsen, S. T., Mazzara, M., Mustafin, R., & Safina, L. (2018). Microservices: How to make your application scale. *Perspectives of System Informatics: 11th International Andrei P. Ershov Informatics Conference, PSI 2017, Moscow, Russia, June 27-29, 2017, Revised Selected Papers 11*, 95–104.

Dubey, A., & Wagle, D. (2007). Delivering software as a service. *The McKinsey Quarterly*, *6*, 1–12.

Dwyer, J., & Truta, T. M. (2013). Finding anomalies in windows event logs using standard deviation. *9th IEEE International Conference on Collaborative Computing: Networking, Applications and Worksharing*, 563–570. https://doi.org/10.4108/icst.collaboratecom.2013.254136

Dynatrace LLC. (o. J.). *Unified observability and security*. Dynatrace | Modern Cloud Done Right. Abgerufen 5. August 2023, von https://www.dynatrace.com/

Elasticsearch B.V. (o. J.). *Past Releases*. Past Releases of Elastic Stack Software. Abgerufen 7. August 2023, von https://www.elastic.co/downloads/past-releases

Fenton, N. E., & Neil, M. (1999). Software metrics: Successes, failures and new directions. *Journal of Systems and Software*, *47*(2–3), 149–157.

Fonseca, R., Porter, G., Katz, R. H., Shenker, S., & Stoica, I. (2007). X-Trace: A Pervasive Network Tracing Framework. *Proceedings of the 4th USENIX Conference on Networked Systems Design & Implementation*, 20.

Fowler, M., & Lewis, J. (2015). Microservices: Nur ein weiteres Konzept in der Softwarearchitektur oder mehr? *ObjektSpektrum, Online Themenspecial Innovation in und durch Architekturen, SigsDatacom*.

Gannon, D., Barga, R., & Sundaresan, N. (2017). Cloud-native applications. *IEEE Cloud Computing*, *4*(5), 16–21. https://doi.org/10.1109/MCC.2017.4250939

*GitLab Documentation*. (o. J.). Abgerufen 12. August 2023, von https://docs.gitlab.com/

Google LLC. (o. J.). *What is Object Storage?* What Is Object Storage? Use Cases & Benefits | Google Cloud. Abgerufen 12. August 2023, von

https://cloud.google.com/learn/what-is-object-storage

*Google/cadvisor*. (o. J.). [Software]. Google. Abgerufen 12. August 2023, von https://github.com/google/cadvisor

Grafana Labs. (o. J.-a). *Flow mode*. Flow Mode | Grafana Agent Documentation. Abgerufen 12. August 2023, von https://grafana.com/docs/agent/latest/flow/

Grafana Labs. (o. J.-b). *Grafana Agent*. Grafana Agent | Grafana Agent Documentation. Abgerufen 4. August 2023, von https://grafana.com/docs/agent/latest/

Grafana Labs. (o. J.-c). *Grafana documentation*. Grafana Documentation | Grafana Documentation. Abgerufen 12. August 2023, von https://grafana.com/docs/grafana/latest/

Grafana Labs. (o. J.-d). *Grafana Loki documentation*. Grafana Loki Documentation | Grafana Loki Documentation. Abgerufen 12. August 2023, von https://grafana.com/docs/loki/latest/

Grafana Labs. (o. J.-e). *Grafana Mimir documentation*. Grafana Mimir Documentation | Grafana Mimir Documentation. Abgerufen 12. August 2023, von https://grafana.com/docs/mimir/latest/

Grafana Labs. (o. J.-f). *Tempo documentation*. Tempo Documentation | Grafana Tempo Documentation. Abgerufen 12. August 2023, von https://grafana.com/docs/tempo/latest/

*Graphite*. (o. J.). Graphite. Abgerufen 7. August 2023, von https://graphiteapp.org/

Gülcü, C. (2003). *The Complete Log4j Manual*. QOS. ch.

Henderson, C. (o. J.). *The Original StatsD* [Software]. Abgerufen 7. August 2023, von https://github.com/iamcal/Flickr-StatsD (Original work published 2010)

Henriksson, S. (2011). A brief history of the stack. *HAPOC11 History and Philosophy of Computing, Ghent, Belgium*.

Horalek, J., Urbanik, P., Sobeslav, V., & Svoboda, T. (2023). Proposed Solution for Log Collection and Analysis in Kubernetes Environment. *Nature of Computation and Communication: 8th EAI International Conference*, 9–22. https://doi.org/10.1007/978-3-031-28790-9_2

Irwin, B. V. W., & Loyola, F. (2022). Towards Scalable Secure Syslog Compatible Remote Logging. *International Conference on Intelligent and Innovative Computing Applications*, 13–21. https://doi.org/10.59200/ICONIC.2022.002

Kabinna, S., Bezemer, C.-P., Shang, W., & Hassan, A. E. (2016). Logging library migrations: A case study for the apache software foundation projects. *Proceedings of the 13th International Conference on Mining Software Repositories*, 154–164. https://doi.org/10.1145/2901739.2901769

Karumuri, S., Solleza, F., Zdonik, S., & Tatbul, N. (2021). Towards Observability Data Management at Scale. *ACM SIGMOD Record*, *49*(4), 18–23. https://doi.org/10.1145/3456859.3456863

Keller, M. S. (1999). Take Command: Cron: Job Scheduler. *Linux Journal*, *1999*(65es), 15-es. https://doi.org/10.5555/327966.327981

Killian, T. (1984). Processes as Files. *Proceedings of the USENIX Association Summer Conference*, 203–207. https://cir.nii.ac.jp/crid/1572543024683894016

Kiourti, A., Psathas, K. A., & Nikita, K. S. (2014). Implantable and ingestible medical devices with wireless telemetry functionalities: A review of current status and challenges. *Bioelectromagnetics*, *35*(1), 1–15. https://doi.org/10.1002/bem.21813

LeFebvre, W. (2004, April 20). *About Top*. Frequently Asked Questions and their Answers. https://web.archive.org/web/20040420005914/http://www.unixtop.org/faq.shtml

Lester, B. P. (1971). *Cost Analysis of Debugging Systems* (MAC-TR-090; MAC Technical Reports). Massachusetts Institute of Technology.

Liu, X., Iftikhar, N., & Xie, X. (2014). Survey of Real-Time Processing Systems for Big Data. *Proceedings of the 18th International Database Engineering & Applications Symposium*, 356–361. https://doi.org/10.1145/2628194.2628251

*Main Page*. (o. J.). collectd Wiki. Abgerufen 7. August 2023, von https://collectd.org/wiki/index.php/Main_Page

Malpass, I. (2011, Februar 15). *Measure Anything, Measure Everything*. Etsy Engineering | Measure Anything, Measure Everything. https://www.etsy.com/codeascraft/measure-anything-measure-everything?utm_source=OpenGraph&utm_medium=PageTools&utm_campaign=Share

MinIO, Inc. (o. J.). *High Performance Object Storage for AI*. MinIO | High Performance, Kubernetes Native Object Storage. Abgerufen 12. August 2023, von https://min.io

New Relic, Inc. (o. J.). *Observability. All-in-One.* New Relic | Monitoring, Debugging und Optimierung für den gesamten Stack. Abgerufen 5. August 2023, von https://newrelic.com/de

Oetiker, T. (o. J.). *Documentation*. MRTG. Abgerufen 7. August 2023, von https://oss.oetiker.ch/mrtg/doc/mrtg.en.html

One Identity LLC. (o. J.). *The foundation of log management*. Syslog-Ng - Log Management Solutions. Abgerufen 7. August 2023, von https://www.syslog-ng.com/

*OpenTelemetry Collector Exporter for node with grpc*. (2023, August 8). @opentelemetry/Exporter-Trace-Otlp-Grpc - Npm. https://www.npmjs.com/package/@opentelemetry/exporter-trace-otlp-grpc

*OpenTelemetry Collector Logs Exporter for node with grpc*. (2023, August 8). @opentelemetry/Exporter-Logs-Otlp-Grpc - Npm. https://www.npmjs.com/package/@opentelemetry/exporter-logs-otlp-grpc

*OpenTelemetry Collector Metrics Exporter for node with grpc*. (2023, August 8). @opentelemetry/Exporter-Metrics-Otlp-Grpc - Npm. https://www.npmjs.com/package/@opentelemetry/exporter-metrics-otlp-grpc

*OpenTelemetry Instrumentation for Java*. (o. J.). [Software]. Abgerufen 12. August 2023, von https://github.com/open-telemetry/opentelemetry-java-instrumentation

*OpenTelemetry Logs SDK*. (2023, August 8). @opentelemetry/Sdk-Logs - Npm. https://www.npmjs.com/package/@opentelemetry/sdk-logs

*OpenTelemetry Meta Packages for Node*. (2023, Juli 12). @opentelemetry/Auto-Instrumentations-Node - Npm. https://www.npmjs.com/package/@opentelemetry/auto-instrumentations-node

*OpenTelemetry Metrics SDK*. (2023, August 8). @opentelemetry/Sdk-Metrics - Npm. https://www.npmjs.com/package/@opentelemetry/sdk-metrics

*OpenTelemetry SDK Autoconfigure*. (o. J.). Opentelemetry-Java/Sdk-Extensions/Autoconfigure/README.Md at Main · Open-Telemetry/Opentelemetry-Java · GitHub. Abgerufen 12. August 2023, von https://github.com/open-telemetry/opentelemetry-java/blob/main/sdk-extensions/autoconfigure/README.md

*OpenTelemetry SDK for Node.js*. (2023, August 8). @opentelemetry/Sdk-Node - Npm. https://www.npmjs.com/package/@opentelemetry/sdk-node

Pahl, C. (2015). Containerization and the PaaS Cloud. *IEEE Cloud Computing*, *2*(3), 24–31. https://doi.org/10.1109/MCC.2015.51

Pan, Y., Chen, I., Brasileiro, F., Jayaputera, G., & Sinnott, R. (2019). A Performance Comparison of Cloud-Based Container Orchestration Tools. *2019 IEEE International Conference on Big Knowledge (ICBK)*, 191–198. https://doi.org/10.1109/ICBK.2019.00033

Peterson and Control Union. (o. J.-a). *History*. History - Peterson and Control Union. Abgerufen 31. Juli 2023, von https://www.petersoncontrolunion.com/en/about-us/history

Peterson and Control Union. (o. J.-b). *Home*. Home - Peterson and Control Union. Abgerufen 31. Juli 2023, von https://www.petersoncontrolunion.com/en

Petre, I., Boncea, R., Radulescu, C. Z., Zamfiroiu, A., & Sandu, I. (2019). A Time-Series Database Analysis Based on a Multi-attribute Maturity Model. *Studies in Informatics and Control*, *28*, 177–188. https://doi.org/10.24846/v28i2y201906

*Pino*. (o. J.). [Software]. pino. Abgerufen 12. August 2023, von https://github.com/pinojs/pino

Ponce, F., Márquez, G., & Astudillo, H. (2019). Migrating from monolithic architecture to microservices: A Rapid Review. *2019 38th International Conference of the Chilean Computer Science Society (SCCC)*, 1–7. https://doi.org/10.1109/SCCC49216.2019.8966423

Potdar, A. M., Narayan, D. G., Kengond, S., & Mulla, M. M. (2020). Performance Evaluation of Docker Container and Virtual Machine. *Procedia Computer Science*, *171*, 1419–1428. https://doi.org/10.1016/j.procs.2020.04.152

Prometheus Authors. (o. J.). *ALERTMANAGER*. Alertmanager | Prometheus. Abgerufen 19. August 2023, von https://prometheus.io/docs/alerting/latest/alertmanager/

*Prometheus client for node.js*. (2023, März 6). Prom-Client - Npm. https://www.npmjs.com/package/prom-client

*Quick Start*. (o. J.). Quick Start | Garage HQ. Abgerufen 12. August 2023, von https://garagehq.deuxfleurs.fr/documentation/quick-start/

*SeaweedFS*. (o. J.). [Software]. Abgerufen 12. August 2023, von https://github.com/seaweedfs/seaweedfs

Shah, P. (2018, Januar 17). OpenCensus: A Stats Collection and Distributed Tracing Framework. *Google Open Source Blog*. https://opensource.googleblog.com/2018/01/opencensus.html

Sheeraz, M., Paracha, M. A., Haque, M. U., Durad, M. H., Mohsin, S. M., Band, S. S., & Mosavi, A. (2023). Effective Security Monitoring Using Efficient SIEM Architecture. *Human-centric Computing and Information Sciences*, *13*, 1–18. https://doi.org/10.22967/HCIS.2023.13.023

Shi, X., Shen, Y., Wang, Y., & Bai, L. (2018). Differential-Clustering Compression Algorithm for Real-Time Aerospace Telemetry Data. *IEEE Access*, *6*, 57425–57433. https://doi.org/10.1109/ACCESS.2018.2872778

Shkuro, Y. (2017, Februar 2). *Evolving Distributed Tracing at Uber Engineering*. Evolving Distributed Tracing at Uber Engineering | Uber Blog. https://www.uber.com/blog/distributed-tracing/

Sidqi, R., Rynaldo, B. R., Suroso, S. H., & Firmansyah, R. (2018). Arduino Based Weather Monitoring Telemetry System Using NRF24L01+. *IOP Conference Series: Materials Science and Engineering*, *336*, 012024. https://doi.org/10.1088/1757-899X/336/1/012024

Sigelman, B. (2016, Juni 16). Towards Turnkey Distributed Tracing. *OpenTracing*. https://medium.com/opentracing/towards-turnkey-distributed-tracing-5f4297d1736

Sigelman, B. (2019, März 28). Merging OpenTracing and OpenCensus: Goals and Non-Goals. *OpenTracing*. https://medium.com/opentracing/merging-opentracing-and-opencensus-f0fe9c7ca6f0

Sigelman, B. H., Barroso, L. A., Burrows, M., Stephenson, P., Plakal, M., Beaver, D., Jaspan, S., & Shanbhag, C. (2010). *Dapper, a Large-Scale Distributed Systems Tracing Infrastructure* (dapper-2010-1; Google Technical Report). Google, Inc. https://research.google.com/archive/papers/dapper-2010-1.pdf

Sigelman, B., & McLean, M. (2019, Mai 21). *A brief history of OpenTelemetry (So Far)*. A Brief History of OpenTelemetry (So Far) | Cloud Native Computing Foundation. https://www.cncf.io/blog/2019/05/21/a-brief-history-of-opentelemetry-so-far/

Splunk Inc. (o. J.). *Make your organization more resilient*. Splunk | The Key to Enterprise Resilience. Abgerufen 6. August 2023, von https://www.splunk.com

*Supported libraries, frameworks, application servers, and JVMs*. (o. J.). Opentelemetry-Java-Instrumentation/Docs/Supported-Libraries.Md at Main · Open-Telemetry/Opentelemetry-Java-Instrumentation · GitHub. Abgerufen 12. August 2023, von https://github.com/open-telemetry/opentelemetry-java-instrumentation/blob/main/docs/supported-libraries.md

systemd. (o. J.). *Native Journal Protocol*. Native Journal Protocol. Abgerufen 7. August 2023, von https://systemd.io/JOURNAL_NATIVE_PROTOCOL/

Thakur, A., & Chandak, M. B. (2022). A review on opentelemetry and HTTP implementation. *International journal of health sciences*, *6*(S2), 15013–15023. https://doi.org/10.53730/ijhs.v6nS2.8972

The Cacti Group, Inc. (o. J.). *About Cacti*. Cacti® - The Complete RRDTool-based Graphing Solution. Abgerufen 7. August 2023, von https://www.cacti.net/

The Jaeger Authors. (o. J.-a). *Client Libraries*. Client Libraries — Jaeger Documentation. Abgerufen 12. August 2023, von https://www.jaegertracing.io/docs/1.47/client-libraries/

The Jaeger Authors. (o. J.-b). *Introduction*. Jaeger Documentation. Abgerufen 12. August 2023, von https://www.jaegertracing.io/docs/1.47/

The Jaeger Authors. (2017, April 14). *News*. Jaeger – News. https://www.jaegertracing.io/news/

The Linux Foundation. (o. J.). *MAKE CLOUD NATIVE UBIQUITOUS*. Cloud Native Computing Foundation. Abgerufen 8. August 2023, von https://www.cncf.io/

The Linux Foundation. (2016, Mai 9). *Cloud Native Computing Foundation accepts Prometheus as second hosted project*. Cloud Native Computing Foundation Accepts Prometheus as Second Hosted Project | Cloud Native Computing Foundation. https://www.cncf.io/announcements/2016/05/09/cloud-native-computing-foundation-accepts-prometheus-as-second-hosted-project/

The Linux Foundation. (2018, August 9). *Cloud Native Computing Foundation announces Prometheus graduation*. Cloud Native Computing Foundation Announces Prometheus Graduation | Cloud Native Computing Foundation. https://www.cncf.io/announcements/2018/08/09/prometheus-graduates/

The Linux Foundation. (2019a, April 11). *Cloud Native Computing Foundation announces Fluentd graduation*. Cloud Native Computing Foundation Announces Fluentd Graduation | Cloud Native Computing Foundation. https://www.cncf.io/announcements/2019/04/11/cncf-announces-fluentd-graduation/

The Linux Foundation. (2019b, Oktober 31). *Cloud Native Computing Foundation announces Jaeger graduation*. Cloud Native Computing Foundation Announces Jaeger Graduation | Cloud Native Computing Foundation. https://www.cncf.io/announcements/2019/10/31/cloud-native-computing-foundation-announces-jaeger-graduation/

The Linux Foundation. (2021, August 26). *OpenTelemetry becomes a CNCF incubating project*. OpenTelemetry Becomes a CNCF Incubating Project | Cloud Native Computing Foundation. https://www.cncf.io/blog/2021/08/26/opentelemetry-becomes-a-cncf-incubating-project/

The Linux Foundation. (2022, Januar 31). *CNCF Archives the OpenTracing Project*. CNCF Archives the OpenTracing Project | Cloud Native Computing Foundation. https://www.cncf.io/blog/2022/01/31/cncf-archives-the-opentracing-project/

The OpenTelemetry Authors. (o. J.). *Documentation*. Documentation | OpenTelemetry. Abgerufen 4.

August 2023, von https://opentelemetry.io/docs/

Thompson, K., & Ritchie, D. M. (1973). *UNIX PROGRAMMER'S MANUAL Third Edition*. Bell Telephone Laboratories, Inc.

Thönes, J. (2015). Microservices. *IEEE software*, *32*(1), 116. https://doi.org/10.1109/MS.2015.11

Usman, M., Ferlin, S., Brunstrom, A., & Taheri, J. (2022). A Survey on Observability of Distributed Edge & Container-Based Microservices. *IEEE Access*, *10*, 86904–86919. https://doi.org/10.1109/ACCESS.2022.3193102

*V0.33.0*. (o. J.). Release v0.33.0 · Grafana/Agent · GitHub. Abgerufen 16. August 2023, von https://github.com/grafana/agent/releases/tag/v0.33.0

*V0.36.0*. (o. J.). V0.36.0 Milestone · GitHub. Abgerufen 16. August 2023, von https://github.com/grafana/agent/milestone/15

Volz, J., & Rabenstein, B. (2015, Februar 26). *Prometheus: Monitoring at SoundCloud*. Prometheus: Monitoring at SoundCloud | SoundCloud Backstage Blog. https://developers.soundcloud.com/blog/prometheus-monitoring-at-soundcloud

*Winston*. (o. J.). [Software]. winstonjs. Abgerufen 12. August 2023, von https://github.com/winstonjs/winston

Woods, N. (2016, Oktober 11). *OpenTracing joins the Cloud Native Computing Foundation*. OpenTracing Joins the Cloud Native Computing Foundation | Cloud Native Computing Foundation. https://www.cncf.io/blog/2016/10/11/opentracing-joins-the-cloud-native-computing-foundation/

Woods, N. (2017, September 13). *CNCF hosts Jaeger*. CNCF Hosts Jaeger | Cloud Native Computing Foundation. https://www.cncf.io/blog/2017/09/13/cncf-hosts-jaeger/

Young, T., Shah, P., Alberto, C., Drutu, B., Kanzhelev, S., & Shkuro, Y. (2019, Mai 21). Merging OpenTracing and OpenCensus: A Roadmap to Convergence. *OpenTracing*. https://medium.com/opentracing/a-roadmap-to-convergence-b074e5815289

*Zipkin*. (o. J.). OpenZipkin · A distributed tracing system. Abgerufen 12. August 2023, von https://zipkin.io/

# Ehrenwörtliche Erklärung

Hiermit versichere ich, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen oder Hilfsmittel benutzt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegt wurde.

Ort, Datum

*Brandenburg an der Havel, 27.08.2023*

Unterschrift