Inhaltsverzeichnis

Inhaltsverzeichnis III

Abbildungsverzeichnis VI

Tabellenverzeichnis VII

Abkürzungsverzeichnis VIII

1 Einleitung 1

1.1 Problemstellung 1

1.2 Zielsetzung 2

1.3 Abgrenzung dieser Arbeit 2

2 Monitoring von Webapplikationen 3

2.1 Von klassischem IT-Monitoring zu Observability 3

2.2 Kernkonzepte der Observability 3

2.3 Weitere Begriffe im Bereich der Observability 4

2.4 Die drei Säulen der Observability 5

2.4.1 Metriken 7

2.4.2 Logs 9

2.4.3 Traces 10

2.5 Technische Implementierung der drei Säulen 13

2.5.1 Metriken 13

2.5.2 Logs 14

2.5.3 Traces 15

2.6 Die drei Säulen der Observability sind oft nicht genug 17

3 Betrachtung der Tracing-Komponenten 19

3.1 Transaktionen beschreiben – Instrumentation ... 19
3.1.1 Grundlegende Konzepte der Instrumentierung ... 19
3.1.2 Manuelle und automatische Instrumentierung ... 20
3.1.3 Verfügbare Protokolle für die Übermittlung von Trace-Daten an das Backend ... 21
3.1.4 OpenTelemetry‘s API-Spezifikation ... 22
3.2 Transaktionen aufzeichnen – Tracer ... 22
3.3 Transaktionen analysieren – Analysetool ... 23
4 Kriterienfindung ... 24
4.1 Anforderungen aufstellen ... 24
4.2 Anforderungen eingrenzen ... 25
4.3 Prioritäten erstellen ... 26
4.4 Kriterien priorisieren ... 27
4.5 Erfüllungsgrade erarbeiten ... 28
5 Nutzwertanalyse von Analysetools ... 29
6 Beispielintegration ... 32
6.1 Umgebung und Servicetopologie ... 32
6.2 Tempo installieren ... 33
6.3 Tempo als Microservice ... 33
7 Notwendige Anpassungen in der Entwicklung und im Betrieb von Webapplikationen 35
8 Zusammenfassung ... 36
8.1 Rückblick ... 36
8.2 Kritische Nachbetrachtung ... 36

8.3 Ausblick ... 37

Literaturverzeichnis ... V

Anlagenverzeichnis ... VII

Ehrenwörtliche Erklärung

dotSource®

Abbildungsverzeichnis

Abbildung 1: Service Level-Begriffe mit Beispielen 5
Abbildung 2: Observability-Bausteine und ihre grafische Darstellung 6
Abbildung 3: visualisierte Metriken Memory/CPU, Logins, Requests, Page Loads 8
Abbildung 4: visualisierte Logs 9
Abbildung 5: Komponenten einer Tracing-Pipeline 11
Abbildung 6: Detailansicht eines Traces in Grafana 12
Abbildung 7: Ausgabe von Metriken auf der Kommandozeile 13
Abbildung 8: Konfiguration von Prometheus zum Abfragen der Metriken 13
Abbildung 9: Visualisierung der Metrik „Ping Request Count“ 14
Abbildung 10: Kommandos zur Einrichtung der Log-Funktion am Webserver Nginx 14
Abbildung 11: Log-Eintrag in Nginx nach angewandter Konfiguration 15
Abbildung 12: Konfigurationsbeispiel für die Instrumentation mit Jaeger 16
Abbildung 13: Detailansicht eines Traces in Jaeger 17
Abbildung 14: Vergleich der Analysetools Zipkin, Jaeger, Tempo 30
Abbildung 15: Microservice-Topologie des Integrationsbeispiels 32

Tabellenverzeichnis

Tabelle 1:Eingrenzen der Anforderungen für die Nutzwertanalyse.. 26
Tabelle 2: Prioritäten zur Gewichtung der Kriterien... 26
Tabelle 3: Zuweisen der Gewichtung den Kriterien 27
Tabelle 4: Erstellte Skala der Erfüllungsgrade der Kriterien 28
Tabelle 5: Punktevergabe und Gesamtergebniserrechnung der Nutzwertanalyse 31

Abkürzungsverzeichnis

HTTP............................Hypertext Transfer Protocol

SLA ....Service Level Agreement

SLI...............................Service Level Indicator

SLO ....Service Level Objective

1 Einleitung

Seit 2006 entwickelt und realisiert dotSource skalierbare Digitalprodukte für Marketing, Vertrieb und Services. Dabei setzen spezialisierte Consulting- und Entwicklungsteams hochintegriert auf die Verbindung von Strategieberatung und Technologieauswahl – von Branding, Konzeption und UX-Design über Conversion-Optimierung bis zum Betrieb in der Cloud. Für den Betrieb der IT-Infrastruktur intern als auch für das angebotene Leistungsportfolio sind zwei Teams in der Agentur zuständig. Das IT Internal Services Team betreibt Soft-, Hardware und Dienste für den Eigenbedarf. Das Cloud Services Team betreibt und wartet die digitalen Systemlandschaft der Kunden, berät bei der IT-Architektur, sorgt mit professionellen Penetrationstests für die Sicherheit externer Systeme, Netzwerke und IT-Umgebungen, leitet im Prozess der Cloud Migration, erledigt Cloud Service Management und betreibt Business Monitoring.

1.1 Problemstellung

Es wird eine Möglichkeit benötigt, Telemetriedaten zu sammeln und an ein Observability Backend zu senden, welches die Daten abfragt und visualisiert, um den Systemstatus zu verstehen. Um beispielsweise zu verstehen, welche API Endpoints Errors erzeugen oder welche Webserver am langsamsten sind. Besonders bei Microservice-Architekturen ist das wichtig, um beispielsweise HTTP Requests besser verfolgen zu können.

1.2 Zielsetzung

Ziel ist es, solides Grundwissen zu dem Thema Tracing aufzubauen. Dabei sollen Merkmale des Tracing herausgestellt werden, um darauf ausgerichtete Anforderungen an ein Tracing Tool zu erarbeiten. Die Anforderungen sollen anhand einer Nutzwertanalyse an mehreren Tools evaluiert werden. Das Beste den Anforderungen entsprechende Tool soll beispielhaft aufgesetzt und erste Erkenntnisse im Umgang damit gewonnen werden. Eine Nachbetrachtung der Integration und die Erkenntnis, welche Maßnahmen notwendig sind, um Tracing in bestehende IT-Landschaften und Softwarearchitekturen zu integrieren, runden das Ziel ab. Verglichen werden kann das am Beispiel des Trackings von Zugvögeln. Beide Methoden dienen dazu, die Bewegung und das Verhalten von Objekten zu verfolgen, die sich über ein verteiltes System bewegen.

1.3 Abgrenzung dieser Arbeit

Im Rahmen dieser Arbeit liegt der Fokus auf der Erweiterung des Monitorings, um den Aspekt Tracing. Bevorzugt werden dabei Werkzeuge in Betracht gezogen, die frei und Open Source für den kommerziellen Einsatz zur Verfügung stehen. Werkzeuge, die umfangreiche und komplette Plattformen darstellen, gehen über die Betrachtung hinaus.

2 Monitoring von Webapplikationen

2.1 Von klassischem IT-Monitoring zu Observability

Monitoring gilt als die Methode der Wahl in der IT-Branche, wenn es darum geht, den reibungslosen Betrieb aller Systeme sicherzustellen. Monitoring beschränkt sich jedoch weitestgehend auf die reine Überwachung. Observability meint eine umfassendere Beobachtbarkeit, die insbesondere Entwicklern einen tieferen Einblick in die Applikation ermöglicht. Mit der Entwicklung von Microservices-Architekturen reicht angesichts der steigenden Komplexität verteilter Systeme klassisches Monitoring nicht mehr aus. Monitoring liefert den für den Betrieb verantwortlichen Teams lediglich Hinweise auf Störungen im Betrieb – etwa durch Fehlermeldungen oder überdurchschnittlich lange Datenbankabfragen. Durch die engere Zusammenarbeit von Entwicklungs- (Dev) und Betriebsteams (Ops) nach dem DevOps-Ansatz, verfolgt Observability das Ziel, tiefere Einblicke in das Verhalten von Anwendungen zu gewinnen, um die Ursache möglicher Fehler zu finden.[1]

2.2 Kernkonzepte der Observability

Durch Beobachtbarkeit kann ein System von außen verstanden werden, indem Fragen zu diesem System gestellt werden, ohne dessen innere Abläufe zu kennen. Darüber hinaus wird es ermöglicht, neuartige Probleme (z. B. „unbekannte Unbekannte“) einfach zu beheben und zu bewältigen, und bei der Beantwortung der Frage, warum ein Verhalten auftritt, zu helfen.[2] Um diese Fragen an ein System stellen zu können, muss die Anwendung instrumentiert sein. Das heißt, der Anwendungscode muss Telemetrie-Signale erzeugen. Die Instrumentierung einer

[1] [Par23]
[2] [The23a]

Anwendung unterstützt Entwickler bei der Problembehebung, indem aufschlussgebende Informationen dafür zur Verfügung stehen.

Telemetrie bezieht sich auf Daten, die von einem System über sein Verhalten gesendet werden. Die Daten können in Form von Metriken, Logs und Traces vorliegen. Ein Log ist eine zeitgestempelte Nachricht, die von Diensten oder anderen Komponenten ausgegeben wird. Sie sind fast überall in Software zu finden und wurden in der Vergangenheit sowohl von Entwicklern als auch von Betreibern stark genutzt, um das Systemverhalten zu verstehen. Metriken sind Zusammenfassungen von numerischen Daten über die Infrastruktur oder Anwendung über einen bestimmten Zeitraum hinweg. Beispiele sind: Systemfehlerrate, CPU-Auslastung, Anforderungsrate für einen bestimmten Dienst. Ein verteiltes Trace, besser bekannt als Trace, zeichnet die Pfade auf, die von Anforderungen (durch eine Anwendung oder einen Endbenutzer) genommen werden, während sie sich durch Multi-Service-Architekturen wie Microservices und serverlose Anwendungen verbreiten. Im Gegensatz zu Logs sind sie mit einer bestimmten User Request oder Transaktion verbunden.

2.3 Weitere Begriffe im Bereich der Observability

Zuverlässigkeit beantwortet die Frage, ob der Dienst das ausführt, was Benutzer von ihm erwarten.[3] Ein System könnte immer zu einhundert Prozent in Betrieb sein, aber wenn ein Benutzer beispielsweise auf „In den Warenkorb" klickt, um eine schwarze Hose in seinen Warenkorb zu legen, und das System stattdessen nicht immer schwarze Hosen hinzufügt, dann gilt das System als unzuverlässig. Ein Service Level Indicator (SLI) stellt eine Messung des Verhaltens eines Dienstes dar.[4] Ein guter SLI misst den Service aus der Perspektive der

[3] [The23a]
[4] [The23a]

Benutzer. Beispiele für SLI sind die Verfügbarkeit, also wie oft ein Service nutzbar ist und die Geschwindigkeit, mit der ein Request beantwortet wird.

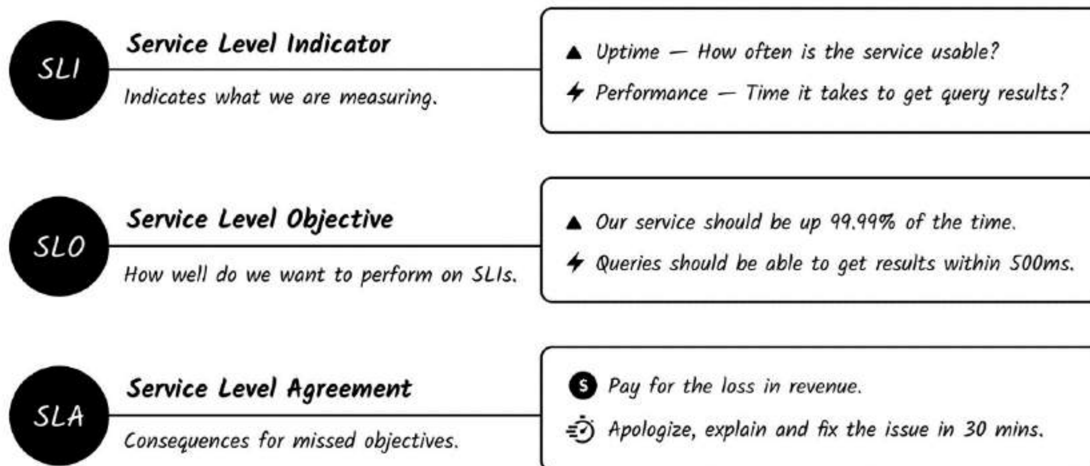

Abbildung 1: Service Level-Begriffe mit Beispielen[5]

Ein Service Level Objective (SLO) ist das Mittel, mit dem Zuverlässigkeit einem Kunden kommuniziert wird.[6] Dies wird erreicht, indem ein oder mehrere SLIs dem Geschäftswert zugeordnet werden. Beispiele für SLO sind Verfügbarkeiten eines Service zu 99,99 % und Beantwortungen von Requests innerhalb von 500 Millisekunden. Ein Service Level Agreement (SLA) beinhaltet Konsequenzen bei verfehlten Zielen.[7] Beispiele für SLA sind Zahlungen in Höhe des Einkommensverlustes und eine Entschuldigung, Erklärung und Problembehebung in 30 Minuten.

2.4 Die drei Säulen der Observability

Es gibt viele potenzielle Datenquellen, die für die Überwachung einer Umgebung genutzt werden können. Für die meisten Anwendungsfälle der Observability sind drei Datentypen am

[5] [Ahm21]
[6] [The23a]
[7] [The23a]

wichtigsten: Metriken, Logs und Traces. Diese Datentypen werden auch als „die drei Säulen der Observability“ bezeichnet.

Abbildung 2: Observability-Bausteine und ihre grafische Darstellung[8]

Jede Säule bietet eine andere Perspektive auf die Ressourcen. Die Kombination und Analyse dieser Datenquellen ermöglicht ein ganzheitliches Verständnis der Vorgänge in komplexen Anwendungsumgebungen.

[8] [Lin21]

2.4.1 Metriken

Metriken sind Zusammenfassungen von numerischen Daten über die Infrastruktur oder Anwendung über einen bestimmten Zeitraum hinweg. Sie sind quantifizierbare Messwerte, die den Zustand und die Leistung der Infrastruktur oder Anwendung widerspiegeln.[9] Metriken erfassen beispielsweise wie viele Transaktionen die Anwendung pro Sekunde verarbeitet oder wie viele CPU- oder Speicherressourcen auf einem Server belegt sind. Es gibt verschiedene Methoden, um Metriken zu bestimmen. Nach der RED-Methode von Weaveworks wird sich auf Raten (Rates), Fehler (Errors) und Anfragedauer (Duration) konzentriert.[10] Der Four Golden Signals-Methode von Google zu Folge, werden Latenz, Traffic, Fehler und Auslastung gemessen.[11] Metriken können auf verschiedene Weisen gemessen werden. Der Typ „Meter“ berechnet die Häufigkeit von Ereignissen (z. B. die Anzahl der Besucher auf der Website). „Timer“ misst die Zeit, die für den Abschluss eines Vorgangs benötigt wird (z. B. die Reaktionszeit des Webservers). Mit dem Typ „Counter“ werden ganzzahlige Werte inkrementiert und dekrementiert (z.B.: Anzahl der angemeldeten Benutzer). Bei „Gauge“ wird ein beliebiger Wert gemessen (z.B. CPU).[12]

Der Vorteil von Metriken ist der Echtzeiteinblick in den Zustand der Ressourcen, welcher Auskunft darüber gibt, wie reaktionsschnell die Anwendung ist und ob Anomalien als ein frühes Anzeichen für ein Leistungsproblem erkannt werden. Grenzen ergeben sich bei der Beschränkung der Auskunft auf konfigurierte Komponenten. Erfasst werden nur Anwendungs- und Infrastrukturdaten, die definiert wurden.

[9] [Toz22]
[10] [Wil17]
[11] [EB20]
[12] [Jan22a]

Abbildung 3: visualisierte Metriken Memory/CPU, Logins, Requests, Page Loads[13]

Metriken allein sind nicht dazu geeignet, die Ursache eines Problems zu ermitteln, insbesondere nicht in einem komplexen verteilten System. Sie können zwar darauf hinweisen, dass die Anwendung eine hohe Fehlerquote aufweist, jedoch ist nicht genau ermittelbar, welcher Dienst innerhalb einer Microservices-Architektur die Fehler auslöst. Beispiele für Metriken sind etwa HTTP-Requests in einem bestimmten Zeitraum, die Auslastung der CPU, Request-Dauer und aktive Verbindungen einer Datenbank sowie Treffer eines Cache.

[13] [Lin21]

2.4.2 Logs

Ein Log ist eine zeitgestempelte Nachricht, die von Diensten oder anderen Komponenten ausgegeben wird. Diese Nachricht wird dem Inhalt einer Log-Datei angehängt. So werden fortlaufend Ereignisse, Warnungen und Fehler aufgezeichnet, die in einer Softwareumgebung auftreten. Logs enthalten Kontextinformationen, zum Beispiel den Zeitpunkt eines Ereignisses und den Benutzer oder Endpunkt, der damit verbunden ist.[14] Die Log-Datei für einen Webserver enthält beispielsweise Informationen wie den Startzeitpunkt des Servers und die User Requests sowie die zugehörige Response des Servers. Es werden sowohl Informationen über jede erfolgreiche Transaktion als auch über Fehler wie fehlgeschlagene Verbindungen zu Clients aufgezeichnet. Zur Unterteilung von Logs werden Log Level wie Debugging, Information, Warning und Fatal Error angewandt.[15]

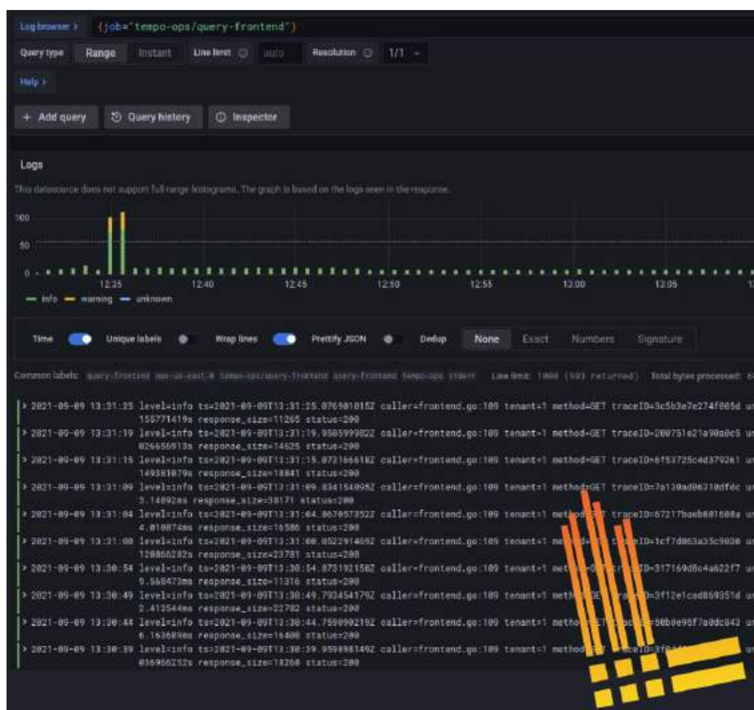

Abbildung 4: visualisierte Logs[16]

[14] [Toz22]
[15] [Jan22a]
[16] [Lin21]

Vorteile von Logs liegen in der Möglichkeit der umfassenden Aufzeichnung und der Aufzeichnung aller Ereignisse und Fehler während des Lebenszyklus von Software-Ressourcen. Daher werden sie auch als Grundpfeiler angesehen. Logs werden eingesetzt, um zu erfahren, wann ein Problem aufgetreten ist oder welche Ereignisse damit korrelieren. Die Grenzen beim Einsatz von ausschließlich Logs zeigen sich bei der Konfiguration und Speicherdauer. Der Umfang der aufgezeichneten Ereignisse hängt wie bei Metriken von der implementierten Konfiguration ab. Wenn Logging-Tools und -einstellungen nicht umfassend konfiguriert sind, werden bestimmte Informationen nicht aufgezeichnet und erscheinen nicht in Logdateien. Weiterhin könnte der Zugriff auf Logs von containerisierten Anwendungen verfallen, wenn sie im Container gespeichert werden und dieser heruntergefahren wird. Ein Ansatz zur Verhinderung dessen ist das Schreiben von Logs an einen anderen Ort. Beispiele für Logs sind Aufzeichnungen zu Zugriffskontrollen, Request-/Error-Raten eines Proxys, Timeouts eines Webservers und Transaktionen bei Datenbanken.

2.4.3 Traces

Ein verteiltes Trace, besser bekannt als Trace, ist ein Datensatz, der Pfade aufzeichnet, die von Requests (durch eine Anwendung oder einen Endbenutzer) genommen werden, während sie Multi-Service-Architekturen wie Microservices und serverlose Anwendungen durchlaufen.[17] In einer Microservice-Welt kommunizieren die Dienste ständig über das Netzwerk miteinander. Beim Tracing wird den von der Anwendung gesendeten Nachrichten eine eindeutige Kennung hinzufügt. Diese eindeutige Kennung ermöglicht es, einzelne Transaktionen auf ihrem Weg durch das System zu verfolgen.[18] Tracing setzt sich aus drei Kernbereichen zusammen. Die Analyse von Transaktionen umfasst die nützliche Darstellung der Pfade. Die Aufzeichnung von Transaktionen sorgt für die Weitergabe des Kontexts, der mit einem Request in einen Dienst gelangt, an andere Prozesse weitergegeben und an Transaktionsdaten angehängt wird, die an

[17] [Toz22]
[18] [The23b]

ein Tracing-Backend zur Visualisierung gesendet werden. Mit diesem Kontext können die Transaktionen später zusammengefügt werden. Mit der Beschreibung von Transaktionen werden Instrumente in einen Service eingebunden, um aufzuzeichnende Vorgänge festzulegen.[19]

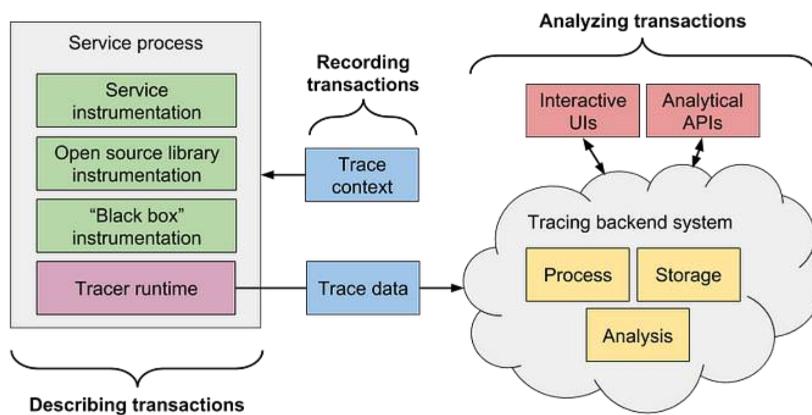

Abbildung 5: Komponenten einer Tracing-Pipeline[20]

Vorteile des Tracing sind Aufzeichnungen darüber, wie lange jede Anwendungskomponente braucht, um den Request zu verarbeiten und das Ergebnis an die nächste Komponente weiterzuleiten (Span). Traces können aufzeigen, welche Teile der Anwendung einen Fehler auslösen. Diese Informationen können verwenden werden, um den Zustand der Anwendung nachzuvollziehen und problematische Microservices oder Aktivitäten zu debuggen. Tracing gilt als der effektivste Weg, um die Ursache eines Problems zu erforschen. Logs und Metriken stellen Probleme fest; Traces ermöglichen, die Ursache zu finden.

[19] [Arn18]
[20] [Arn18]

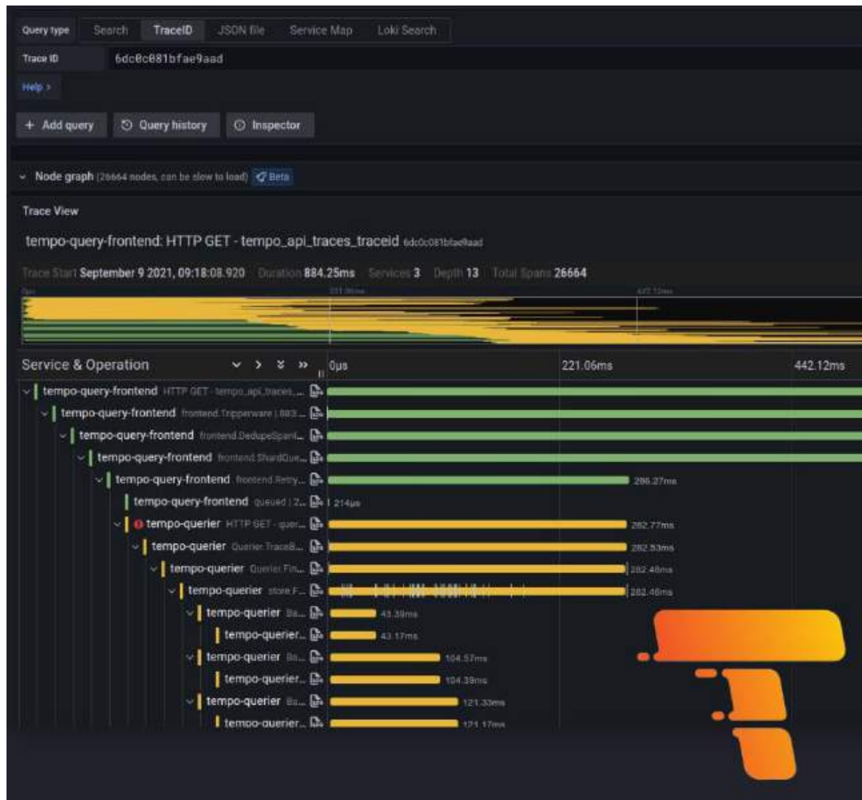

Abbildung 6: Detailansicht eines Traces in Grafana[21]

Tracing stößt an Grenzen, wenn nur ein Bruchteil aller Application Requests verfolgt werden kann. Denn die Ausführung benötigt viel Zeit und viele Ressourcen, um jeden Request zu überwachen. Dadurch stehen nicht immer Tracing-Daten zur Verfügung, wenn ein Fehler auftritt. Oft ist es zudem nicht möglich, aus den Traces eines Requests zu extrapolieren, woher die Fehler eines anderen Request kommen. Dafür variieren die Informationen zu Requests, Endpoints und clientseitigen Konfigurationen oft zu sehr zwischen den einzelnen Vorgängen. Beispiele, in denen Tracing angewandt werden kann sind die Produktsuche, das Betrachten des Warenkorbes und der Bezahlvorgang.

[21] Vgl. [Lin21]

2.5 Technische Implementierung der drei Säulen

2.5.1 Metriken

Prometheus ist ein beliebtes Open-Source-System zur Überwachung und Warnung, das Metriken von verschiedenen Quellen sammelt und speichert. Um mit Prometheus beispielsweise eine Linux Node zu überwachen, wird auf dieser eine Node Exporter als Dienst gestartet. Dieser Dienst bietet in seiner Grundkonfiguration bereits einige abzufragende Metriken an, die dann über einen Endpunkt bereitgestellt werden, wie folgend ersichtlich.

```
> curl http://localhost:9100/metrics
# HELP go_gc_duration_seconds A summary of the GC invocation durations.
# TYPE go_gc_duration_seconds summary
go_gc_duration_seconds{quantile="0"} 3.8996e-05
go_gc_duration_seconds{quantile="0.25"} 4.5926e-05
go_gc_duration_seconds{quantile="0.5"} 5.846e-05
# etc.
```

Abbildung 7: Ausgabe von Metriken auf der Kommandozeile

Prometheus kann dann – ebenfalls als gestarteter Dienst - die bereitgestellten Metriken abrufen. Anhand folgenden Ausschnitts einer Beispielkonfigurationsdatei wird festgelegt, welche Node (Target) wie oft (Interval) abgefragt werden soll (scrape).

```
global:
  scrape_interval: 15s

scrape_configs:
- job_name: node
  static_configs:
  - targets: ['localhost:9100']
```

Abbildung 8: Konfiguration von Prometheus zum Abfragen der Metriken

Die Metriken können dann mit verschiedenen Tools wie PromQL, Grafana oder Alertmanager analysiert und visualisiert werden. Die folgende Abbildung visualisiert die Anzahl an Server Requests.

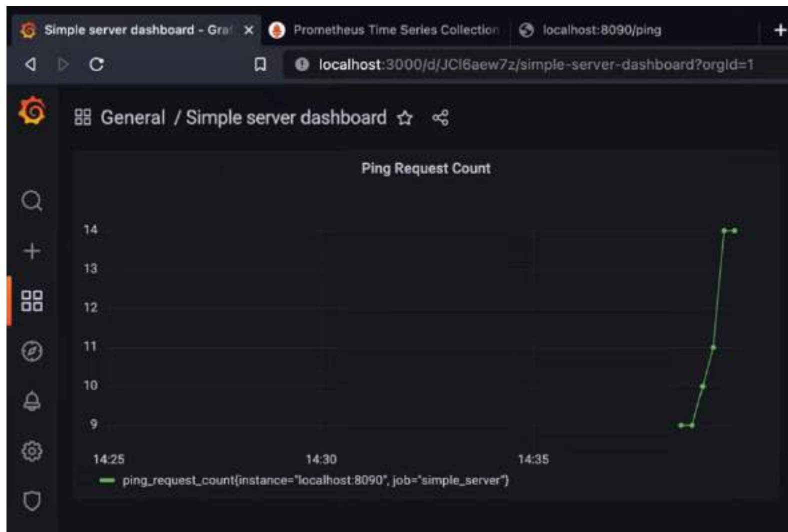

Abbildung 9: Visualisierung der Metrik „Ping Request Count“[22]

2.5.2 Logs

Um die Log-Funktion am Beispiel von Nginx, einem Webserver, zu aktivieren, müssen die entsprechenden Pfade in der Konfigurationsdatei „nginx.conf“ oder in einer Server- oder Location-Block-Datei angegeben werden. Folgende Pfade können verwendet verwenden, um einen Access-Log-Dateinamen, ein Log-Format und eine Log Rotation zu definieren:

```
# Set the name of the log file
access_log /var/log/nginx/access.log;
# Set the log format
log_format combined '$remote_addr - $remote_user [$time_local] "$request" '
'$status $body_bytes_sent "$http_referer" '
'"$http_user_agent";
# Rotate the log file every day
access_log /var/log/nginx/access.log main rotate=1;
```

Abbildung 10: Kommandos zur Einrichtung der Log-Funktion am Webserver Nginx

[22] [Lin21]

```
127.0.0.1 - - [07/Apr/2023:12:34:56 +0200] "GET /index.html HTTP/1.1" 200
612 "-" "Mozilla/5.0 (X11; Linux x86_64) AppleWebKit/537.36 (KHTML, like
Gecko) Chrome/89.0.4389.114 Safari/537.36"
```

Abbildung 11: Log-Eintrag in Nginx nach angewandter Konfiguration

Nach dem zuvor gewählten Format wird, ein wie dem oben ersichtlichen, Log-Eintrag generiert. Dieser Eintrag zeigt an, dass ein Client mit der IP-Adresse 127.0.0.1 am 7. April 2023 um 12:34:56 Uhr die Datei „index.html“ vom Server angefordert hat, indem er das HTTP-Protokoll Version 1.1 verwendet hat. Der Server hat den Request mit dem Statuscode 200 (OK) beantwortet und 612 Bytes an den Client gesendet. Der Client hat keinen Referer angegeben und hat sich als Chrome-Browser auf einem Linux-System identifiziert.

2.5.3 Traces

Um für einen Microservice Traces zu generieren, muss man in der Regel ein Tracing-Tool oder eine Bibliothek verwenden, die die Traces von dem Microservice sammelt und an eine externe Plattform überträgt, wo sie analysiert, visualisiert und korreliert werden können. Jaeger ist ein Open-Source-System für Distributed Tracing, das Traces von verschiedenen Quellen sammelt und speichert. Um Jaeger zu verwenden, muss man einen Jaeger Agent auf der Node bereitstellen, die den Microservice ausführt, und eine Jaeger-Bibliothek in den Microservice integrieren, die die Traces an den Agent sendet. Die notwendigen Konfigurationen könnten wie folgt aussehen.

```
// Import the Jaeger dependencies
import io.jaegertracing.Configuration;
import io.jaegertracing.internal.JaegerTracer;
import io.opentracing.Span;
import io.opentracing.Tracer;

public class WebService {
  // Create a Jaeger tracer
  private static final Tracer tracer = initTracer("web-service");

  // Initialize the tracer with the service name and the agent host and
port
  private static JaegerTracer initTracer(String service) {
    Configuration.SamplerConfiguration samplerConfig =
Configuration.SamplerConfiguration.fromEnv()
        .withType("const")
        .withParam(1);
    Configuration.ReporterConfiguration reporterConfig =
Configuration.ReporterConfiguration.fromEnv()
        .withLogSpans(true)
        .withFlushInterval(1000)
        .withMaxQueueSize(10000)
        .withSender(
            Configuration.SenderConfiguration.fromEnv()
                .withAgentHost("localhost")
                .withAgentPort(6831));
    return Configuration.fromEnv(service)
        .withSampler(samplerConfig)
        .withReporter(reporterConfig)
        .getTracer();
  }
  // Create a span for each request
  public void handleRequest() {
    Span span = tracer.buildSpan("handle-request").start();
    try {
      // Do some work
      span.log("work done");
    } catch (Exception e) {
      // Log any errors
      span.log(e.getMessage());
    } finally {
      // Finish the span
      span.finish(); } } }
```
Abbildung 12: Konfigurationsbeispiel für die Instrumentation mit Jaeger

Der Jaeger Agent sendet dann die Traces an ein Backend. Dort werden sie gesammelt und können mit verschiedenen Tools wie dem Jaeger-UI, Grafana oder Prometheus visualisiert und analysiert werden. Eine Detailansicht eines Trace in Jaeger UI ist wie folgt ersichtlich.

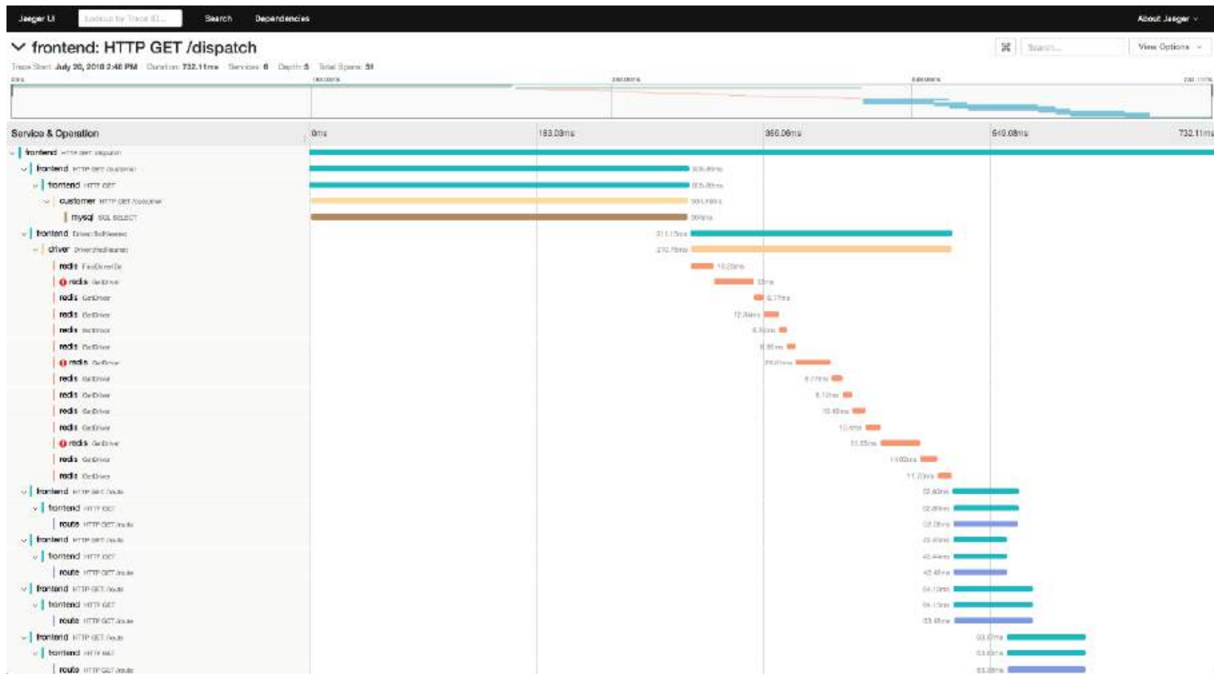

Abbildung 13: Detailansicht eines Traces in Jaeger[23]

2.6 Die drei Säulen der Observability sind oft nicht genug

Durch die Kombination der drei Säulen ergibt sich ein relativ vollständiges Bild der Vorgänge im System. Bei der Verfolgung von Metriken ist in einem erdachten Szenario feststellbar, dass sich die Antwortrate der Anwendung verlangsamt. Das deutet auf ein Leistungsproblem hin. Ein Blick in die Anwendungslogs verrät, ob langsamere Antworten durch eine harmlose Änderung erklärbar sind. Beispielsweise verarbeitet die Anwendung komplexere Transaktionen als normalerweise. Nach der Feststellung, dass die Leistungsverschlechterung der Anwendung auf einen Fehler zurückzuführen ist, kann anhand der verteilten Trace-Daten festgestellt werden, welcher spezifische Microservice betroffen ist.

Die drei Säulen der Beobachtbarkeit bilden insgesamt eine solide Grundlage für die Überwachung von Systemen. Auf diese sollte sich jedoch nicht beschränkt werden. Je mehr

[23] [The23c]

Daten zur Verfügung stehen, um die Monitoring-Workflows zu bereichern, desto besser. Beispielsweise kann es sinnvoll sein, Metriken, Logs und Traces mit Daten aus einer CI/CD Pipeline zu kontextualisieren, um festzustellen, welches Anwendungsupdate oder welche Neuverteilung mit einer Verschlechterung der Performance korreliert.[24] Ebenso können Geschäftsmetriken wie Kundenbindungsraten mit technischen Beobachtungsmetriken korreliert werden, um die Auswirkungen technischer Probleme auf die Geschäftsleistung zu messen.

[24] [Toz22]

3 Betrachtung der Tracing-Komponenten

Tracing setzt sich aus drei Kernbereichen zusammen – dem Beschreiben, Aufzeichnen und Analysieren von Transaktionen. Entsprechend dieser Kernbereiche unterteilen sich auch die benötigten Tracing-Komponenten in die Kategorien Instrumentierung, Tracer und Analysetool. Die Instrumentierung meint die Library, die angibt, was aufgezeichnet werden soll. Der Tracer übernimmt die Aufzeichnung und Übermittlung dieser Daten. Das Analysetool – als Backend – erhält die Trace-Informationen zur weiteren Verarbeitung.[25] In der Praxis sind diese Kategorien fließend. Die Unterscheidung zwischen Instrumentierung und Tracer ist nicht immer eindeutig. ist.

3.1 Transaktionen beschreiben – Instrumentation

Jeder Ursprung im verteilten Tracing befindet sich in der Instrumentierung einer Anwendung, um Trace-Daten von jedem Dienst während der Ausführung zu emittieren oder zu extrahieren. Es gibt viele Möglichkeiten der Instrumentierung, einschließlich der Verwendung von SDKs und vorkonfigurierten Frameworks, und viele Protokolle für die Übertragung der Trace-Daten an das Analysetool.

3.1.1 Grundlegende Konzepte der Instrumentierung

Wenn ein Dienst instrumentiert ist, erzeugt jeder Aufruf einer Operation des Dienstes einen Span. Spans können manuell im Code erstellt werden, indem die API und das SDK verwendet werden, die von einer Tracer-Client-Bibliothek bereitgestellt werden. In einigen Fällen können auch Auto-Instrumentation-Agents verwendet werden, die Spans automatisch erzeugen, ohne

[25] [Kro19]

dass der Applikationscode geändert werden muss.[26] Der Span enthält Daten über den aufgerufenen Dienst und die Operation, den Zeitstempel des Aufrufs, den Kontext des Spans (etwa Trace-ID, Span-ID, übergeordnete Span-ID) sowie zusätzliche Metadaten wie Tags und Protokolle. Der Span wird in einem bestimmten Protokoll formatiert und über einen Tracer an ein verteiltes Tracing-Backend – typischerweise ein Agent oder eine Collector-Komponente – gesendet. Die Spans werden vom Backend empfangen und gesammelt, und die Traces werden rekonstruiert, indem die Spans in der Reihenfolge des Aufrufs aneinandergereiht werden.

Instrumentierung ist also die Fähigkeit der Dienste, gut formatierte Spans mit dem richtigen Kontext auszugeben. Um Instrumentierung zu können, müssen die Fragen beantwortet werden, wie ein Span erzeugt wird und in welchen Protokollen die Spans ausgegeben werden. Die Antworten auf diese Fragen hängen von den spezifischen Anforderungen der Anwendung ab.

3.1.2 Manuelle und automatische Instrumentierung

Manuelle Instrumentierung bedeutet, dass der Entwickler der Anwendung Code hinzufügen muss, um eine Messung zu starten und zu beenden und die Nutzlast zu definieren. Dies geschieht mit Hilfe von Client-Bibliotheken und SDKs, die für eine Vielzahl von Programmiersprachen verfügbar sind. Die manuelle Instrumentierung gibt maximale Kontrolle über die generierten Daten. Benutzerdefinierte Codeblöcke können so instrumentiert werden.[27] Es ermöglicht die Erfassung von Geschäftsmetriken oder anderen benutzerdefinierten Metriken innerhalb des Traces, einschließlich Ereignissen oder Nachrichten, die für die Überwachung oder das Geschäftsmonitoring verwenden werden wollen. Allerdings ist die manuelle Instrumentierung zeitaufwendig. Es gibt eine Lernkurve, um die Instrumentierung zu perfektionieren Bei unsachgemäßer Implementierung können Leistungseinbußen auftreten. Es besteht mehr Spielraum für menschliches Versagen, was zu falschen Messbereichen führen

[26] [Hor20]
[27] [Hor20]

kann. Änderungen an der Instrumentierung können eine Neukompilierung der Anwendung erfordern.

Die automatische Instrumentierung erfordert keine Änderungen am Code. Bei dieser Methode wird ein intelligenter Agent verwendet, der sich an die laufende Anwendung anhängt und Trace-Daten extrahiert. Agenten für die automatische Instrumentierung sind für gängige Programmiersprachen wie Python, Java, .NET und PHP verfügbar. Darüber hinaus bieten gängige Bibliotheken und Frameworks für diese Sprachen ebenfalls integrierte Instrumentierung an. Viele gängige Service-Meshes und Proxies unterstützen ebenfalls die automatische Instrumentierung, wie z.B: Istio, Envoy, Traefik.[28] Damit bietet diese Variante eine gute Abdeckung von Endpunkten und Operationen der Anwendung. Der Bedarf an Code-Änderungen aufgrund von Aktualisierungen der Instrumentierung (z.B. neue Metadaten, die in der Nutzlast erfasst werden) wird reduziert. Im Normalfall wird Zeit bei der automatischen Instrumentierung des Codes gespart. Jedoch bieten nicht alle Sprachen und Frameworks automatische Instrumentierung an. Sie bietet weniger Flexibilität als manuelle Instrumentierung, typischerweise als Teil einer Funktion oder eines Methodenaufrufs. Automatisch instrumentiert werden nur grundlegende Metriken zu Nutzung und Leistung. Geschäftsmetriken oder andere benutzerdefinierte Metriken müssen manuell instrumentiert werden. Automatische Instrumentierung erfasst oft nur Fehlerdaten in Form von zugehörigen Ereignissen oder Logs innerhalb eines Traces.

3.1.3 Verfügbare Protokolle für die Übermittlung von Trace-Daten an das Backend

Die Protokolle der Open-Source-Tools Zipkin und Jaeger finden eine breite Unterstützung in der Industrie und im Fall von Jaeger auch von der Cloud Native Computing Foundation (CNCF). Der bevorzugte Weg, die Bindung an einen bestimmten Hersteller zu vermeiden und den Code zukunftssicher zu machen, ist, einer offenen Spezifikation oder einem offenen

[28] [Hor20]

Protokoll zu folgen. OpenTelemetry definiert eine hersteller- und toolunabhängige Protokollspezifikation namens OTLP für die Übertragung von Metriken, Logs, Traces und weiteren Telemetriedaten. Dies bedeutet, dass der Austausch eines Tools nur eine Konfigurationsänderung am Backend-Kollektor erfordert.

3.1.4 OpenTelemetry‘s API-Spezifikation

Es gibt einen offenen, von einer Community verwalteten Standard – OpenTelemetry –, der herstellerunabhängig ist. Er wird von der Cloud Native Computing Foundation (CNCF) und allen großen Observability- und Cloud-Anbietern sowie der Endnutzer-Community gefördert. Wenn Code mit einer OpenTelemetry-kompatiblen API instrumentiert wird, sollte der Wechsel zwischen verteilten Tracing-Tools so einfach sein, wie der Austausch des verwendeten Tracers oder Autoinstrumentierungs-Agents, wobei der restliche Instrumentierungscode unverändert bleibt. Der derzeit bevorzugte Weg für Tracing-Instrumentierung ist die OpenTelemetry-Spezifikation.[29] OpenTelemetry unterstützt Tracer für alle gängigen Open Source Monitoring und Applikation Performance Monitoring Tools wie Jaeger, Zipkin und Apache Skywalker ebenso wie proprietäre Tools wie VMWare Wavefront, LightStep und Elastic APM. Es zielt darauf ab, das gesamte Spektrum von Observability-Daten über Metriken, Logs und Traces abzudecken.

3.2 Transaktionen aufzeichnen – Tracer

Tracing ermöglicht die Nachverfolgung vieler Transaktionen. Im Wesentlichen wird der Kontext, der mit einem Request in einen Dienst gelangt, von einer Art Tracer an andere Prozesse weitergegeben und an Transaktionsdaten angehängt, die an ein Tracing-Backend

[29] [Hor20]

gesendet werden. Mit diesem Kontext können die Transaktionen später wieder zusammengeführt werden. New Relic macht dies bereits mit applikationsübergreifendem Tracing. Da sich der Architekturfortschritt von Monolithen hin zu Microservices bewegt, wird es immer wichtiger, Transaktionen über Prozessgrenzen hinweg zu verfolgen, wo keine APM Agents installiert werden können.[30] Mit zunehmender Anzahl an Microservices und Multi-Cloud-Infrastrukturen wird mehr Trace-Kontext generiert, der über Schnittstellen transportiert wird, an denen er verloren gehen kann. Das World Wide Web Consortiumie veröffentlicht dazu seit 2021 eine eigene Spezifikation. Sie definiert Standard-HTTP-Header und ein Werteformat, um Kontextinformationen zu verbreiten, die verteilte Tracing-Szenarien ermöglichen. Die Spezifikation standardisiert, wie Kontextinformationen zwischen den Diensten gesendet und geändert werden.

3.3 Transaktionen analysieren – Analysetool

Mit der Visualisierung werden Grundlagen ermöglicht, wie die Zusammensetzung eines Traces um herauszufinden, warum er langsam war, die Darstellung historischer Trends für einen bestimmten Workflow bis hin zu der Verwendung von maschinellem Lernen zur Erkennung von Mustern im System[31]. Im Rahmen dieser Betrachtung werden drei bekannte Open-Source-Projekte aufgeführt. Zipkin, Jaeger und Grafana Tempo. Es gibt zahlreiche weitere Projekte und Anbieter, z. B. Skywalking, Pinpoint, AWS X-Ray oder Google Stackdriver. Mit der Begrenzung auf Open Source Software und der Konzentration auf die Untersuchung von Traces werden diese aber nicht betrachtet. Meist wird der Begriff „Analysetool“ bei kommerziell vertriebene Werkzeuge auch weiter gefasst, da sie vollständige Beobachtungsplattformen sind.

[30] [Hor20]
[31] [Hor20]

4 Kriterienfindung

Um eine fundierte Entscheidung zu treffen, welches Analysewerkzeug für den zukünftigen Einsatz geeignet ist, wird eine Evaluation vorgenommen. Für messbare, relevante Unterscheidungen oder Gemeinsamkeiten in dieser Evaluation werden Anforderungen benötigt.

4.1 Anforderungen aufstellen

Diese Anforderungen werden erarbeitet und setzen sich aus den Anforderungen an den Betrieb des Analysewerkzeugs, und an die Entwicklung von Webapplikationen zusammen. Die Entwicklung von Webapplikationen ist zwar kein direkter Bestandteil des Analysewerkzeugs, aber sie schafft anhand der gewählten Instrumentation die Grundlagen für die Aufzeichnung von Traces und die wiederum deren Analyse. Die Analyse von Traces ist also nur so gut wie deren Aufzeichnung, die nur so gut wie deren Instrumentation sein kann. Bei der Entwicklung von Microservices für Webapplikationen werden verschiedene Programmiersprachen verwendet. Die Instrumentation muss über verfügbare Libraries für häufig genutzte Programmiersprachen möglich sein.

Die Anforderungen an den Betrieb des Analysewerkzeugs betreffen überwiegend die Anbindung an bestehende Infrastruktur. Microservices werden in Containern betrieben, welche mit Kubernetes orchestriert werden. Das Analysetool sollte sich in einer orchestrierten Umgebung betreiben lassen, und zwar Cloud-agnostisch und anhand verfügbarer Helm Charts (Paketmanager für Kubernetes). Webapplikationen und Cloud-Infrastrukturen werden überwacht, um Fehlverhalten festzustellen. Dabei kommt Prometheus und Grafana zum Einsatz. Die Visualisierung von Traces in Grafana ist mindestens erwünscht. Microservices kommunizieren untereinander. Auch diese Kommunikation kann beobachtet, verwaltet und gesteuert werden – beispielsweise mit Service Meshs wie Traefik. Das Tracing-Analysetool

sollte daher ein Protokoll unterstützen, welches von Traefik verwenden werden kann. Um Monitoring-Daten persistent zu speichern, werden Cloud-Speicher verwendet. Die Unterstützung von Cloud-Speicher wird daher vorausgesetzt.

Weitere zu betrachtende Eigenschaften sollen die Möglichkeiten des Betriebs per Managed Hosting einschließen, die (Open Source-) Lizenz, unter welcher das Analysetool veröffentlicht wurde, der Ressourcenverbrauch, die Möglichkeiten Konfigurationen als Code zu versionieren, der Grad an möglicher Automatisierung, die Herstellerneutralität und Formatkompatibilität (beim Datentransfer). Ergänzt werden Merkmale zu Kernfunktionalitäten im Tracing. Dazu gehören Span-Transport, unterstützte Sampling-Strategie und die Komplexität der Architektur (Anzahl an verteilten Komponenten). Abgerundet wird der Vergleich von der Betrachtung des Software-Reifegrads und der Anzahl der zur Software beitragenden Entwicklergemeinschaft.

4.2 Anforderungen eingrenzen

Die zahlreichen, aufgekommenen Kriterien werden für die Nutzwertanalyse eingegrenzt. Viele Kriterien überschneiden sich in ihrer Bedeutung, bedingen sich oder können zusammengefasst werden. Die Anforderungen an die Anbindung an bestehende Infrastruktur werden zusammengefasst, die Unterstützung von Cloud-Speicher in die generelle Unterstützung von Speichern überführt, die Möglichkeiten Konfigurationen als Code zu versionieren sowie der Grad an möglicher Automatisierung abgegrenzt. Somit ergeben sich folgend aufgelistete Kriterien für die Nutzwertanalyse:

Nr.	Kriterium
C01	Anbindung an bestehende Infrastruktur
C02	Anbindung an bestehende Entwicklungsprozesse
C03	Hostingmöglichkeiten
C04	Lizenzart
C05	Herstellerneutralität
C06	Möglichkeiten des Datentransfers zwischen Komponenten
C10.01	OpenTelemetry-Kompatibilität
C10.02	Unterstützter Storage
C10.03	Möglichkeiten des Betriebs (Deployment Arch.)
C10.04	Möglichkeiten des Span-Transports
C10.05	Unterstützte Sampling-Strategien
C10.06	Ressourcenverbrauch
C10.07	Komplexität (Anzahl bestehender Komponenten)
C11.01	Reifegrad (Erstrelease und Versionsfortschritte)
C11.02	Architektur (Systemdesign)
C11.03	Entwicklergemeinschaft (Contributor)

Tabelle 1:Eingrenzen der Anforderungen für die Nutzwertanalyse

4.3 Prioritäten erstellen

Kriterien können unterschiedlich relevant sein und sich in ihrer Wichtigkeit bei der Betrachtung für den Einsatz unterscheiden. Um dies in der Nutzwertanalyse widerzuspiegeln werden Prioritäten erstellt, die anschließend einem Kriterium zugewiesen werden. In der Nutzwertanalyse werden die zugewiesenen Prioritäten verwendet, um die Bewertung zu gewichten. Dazu wird die vergebene Punktzahl beim Erfüllungsgrad mit der Priorität multipliziert. So erreicht ein Kriterium, welches das gleiche Erfüllungsgrad wie ein anderes aufweist, aber relevanter ist, eine höhere Punktzahl. Folgende Prioritäten werden erstellt:

Bez.	Priorität	Beschreibung	Punkte
P01	hoch	Hoher Stellenwert für den Einsatz	3
P02	mittel	Mittlerer Stellenwert für den Einsatz	2
P03	niedrig	Niedriger Stellenwert für den Einsatz	1

Tabelle 2: Prioritäten zur Gewichtung der Kriterien

4.4 Kriterien priorisieren

Die Anbindung an bestehende Infrastruktur, die Hostingmöglichkeiten, die Lizenzart und der Reifegrad werden mit einem niedrigen Stellenwert versehen. Damit wird Wert auf die Tracing-Funktionalitäten gelegt und es soll eine Verschiebung von Tendenzen bei der Erreichung der Punktzahl vermieden werden, bei denen sich die Evaluation eher am Ist-Zustand anstatt auf den Zweck und Nutzen des Tracings orientiert. Einen mittleren Stellenwert erhalten deswegen die Kriterien zur Anbindung an bestehende Entwicklerprozesse (verfügbare Libraries häufig genutzter Programmiersprachen), die Herstellerneutralität, die Möglichkeiten des Datentransfers zwischen Komponenten, die Kompatibilität zu OpenTelemetry, die Möglichkeiten des Span-Transports, unterstützte Sampling-Strategien, die Komplexität sowie die Entwicklergemeinschaft. Der Ressourcenverbrauch hängt maßgeblich von der gewählten Sampling-Strategie ab und stellt einen hohen Stellenwert dar, weil die Cloud-Ressourcennutzung kostet und wirtschaftliches Handeln erfordert. Weiterhin haben die Kriterien zur Speicher-Unterstützung, die Möglichkeiten des Betriebs und die Architektur des Analysetools einen hohen Stellenwert zugewiesen bekommen.

Nr.	Kriterium	Prioritätspunkte
C01	Anbindung an bestehende Infrastruktur	1
C02	Anbindung an bestehende Entwicklungsprozesse	2
C03	Hostingmöglichkeiten	1
C04	Lizenzart	1
C05	Herstellerneutralität	2
C06	Möglichkeiten des Datentransfers zwischen Komponenten	2
C10.01	OpenTelemetry-Kompatibilität	2
C10.02	Unterstützter Storage	3
C10.03	Möglichkeiten des Betriebs (Deployment Arch.)	3
C10.04	Möglichkeiten des Span-Transports	2
C10.05	Unterstützte Sampling-Strategien	2
C10.06	Ressourcenverbrauch	3
C10.07	Komplexität (Anzahl bestehender Komponenten)	2
C11.01	Reifegrad (Erstrelease und Versionsfortschritte)	1
C11.02	Architektur (Systemdesign)	3
C11.03	Entwicklergemeinschaft (Contributor)	2

Tabelle 3: Zuweisen der Gewichtung den Kriterien

4.5 Erfüllungsgrade erarbeiten

Der Hauptbestandteil der Analyse ist die Bewertung der Kriterien. Diese wird anhand von Erfüllungsgraden vorgenommen. Jeder Erfüllungsgrad besitzt eine Punktzahl. Ist ein Kriterium zu einem bestimmten Grad erfüllt, erhält es die zugehörige Punktzahl. Grundlegend werden folgende Erfüllungsgrade aufgestellt. Bei der Betrachtung jedes Kriteriums wird der Erfüllungsgrad auf den Kontext des Kriteriums bezogen und danach ausgerichtet.

Bez.		Beschreibung des Erfüllungsgrads	Punkte
F01		Hervorragend erfüllt	3
F02		Vollständig erfüllt	2
F03		Überwiegend erfüllt	1
F04		Nicht erfüllt	0

Tabelle 4: Erstellte Skala der Erfüllungsgrade der Kriterien

5 Nutzwertanalyse von Analysetools

In der Anbindung an die bestehende Infrastruktur unterscheiden sich die Tools nicht sonderlich. Lediglich Tempo hebt sich dadurch hervor, dass es in das bestehende Visualisierungstool Grafana nahtlos eingebunden werden kann. Ansonsten sind alle Tools in einer containerisierten Umgebung ausführbar. Alle drei Tools unterstützen häufig genutzte Programmiersprachen (Anbindung an bestehende Entwicklerprozesse) und bieten dafür Libraries an. Tempo hat für weitere Programmiersprachen kleine Abstriche zu verzeichnen. Alle Tools können selbst betrieben werden. Tempo wird darüber hinaus zusätzlich in der Grafana Cloud angeboten. Jedes Tool ist unter einer Open Source Lizenz veröffentlicht. Die AGPL-Lizenz, unter welcher Tempo veröffentlicht ist, verlangt die erneute Veröffentlichung des Quellcodes unter derselben Lizenz und birgt damit ein geringeres Risiko für eine Lizenzwechsel. Tempo bietet die umfangreichste Unterstützung für Speicher an und hebt sich damit von Zipkin und Jaeger ab. Zipkins‘ Architektur wurde nicht im Microservice-Stil entwickelt und ist weniger nativ in containerisierten Umgebungen betreibbar. Es unterstützt jedoch mehr Möglichkeiten – als Jaeger, und Jaeger mehr als Tempo – Spans zu transportieren. Umgekehrt bietet Tempo ganzheitliches Sampling auf dem gesamten Trace an. Jaeger nur anhand der probabilistischen und adaptiven Methode, Zipkin nur anhand der probabilistischen Methode. Abhängig von der Sampling-Strategie steigt auch der Ressourcenverbrauch beim Betrieb einer Applikation. Die probabilistische verbraucht am wenigsten, Sampling auf dem gesamten Trace am meisten Ressourcen. Der Reifegrad ist anhand der ersten Veröffentlichung und dem Versionsfortschritt bei Jaeger am geringsten. Dafür besteht es aus den wenigsten Komponenten und sollte damit weniger Aufwand bei der Wartbarkeit der Komponenten benötigen. Die Tendenz spiegelt sich bei der Entwicklergemeinschaft wider. Diese ist bei Jaeger am kleinsten, bei Zipkin am größten.

Tools Nr.	Zipkin	Jaeger	Tempo
C01	Nicht erfüllt	nicht erfüllt	erfüllt
C02	Node.js, java, c#, go, php, scala, ruby	node.js, java, c#, go, python, c++	node.js, java, c#, go, python
C03	self-hosted	self-hosted	self-hosted, cloud-managed
C04	Apache-2.0	Apache-2.0	AGPL-3.0
C05	kann Spuren in gängige Formate aufnehmen und exportieren	kann Spuren in gängige Formate aufnehmen und exportieren	kann Spuren in gängige Formate aufnehmen und exportieren
C06	HTTP, Kafka	HTTP, gRPC, Thrift, Kafka	HTTP, gRPC
C10.01	ja	ja	ja
C10.02	in-memory, ElasticSearch, Kassandra, MySQL	in-memory, ElasticSearch, Kassandra	Local Storage, Object Storage
C10.03	Java-Programm, Docker	Kubernetes, Docker	Kubernetes, Docker
C10.04	HTTP, Scribe, Kafka, AMQP	HTTP, UDP	HTTP
C10.05	probabilistisch	probabilistisch, adaptiv	Tail-basiert
C10.06	gering	Gering, mittelmäßig	Gering bis hoch
C10.07	collector, storage, query	agent, client, storage, collector, ingester, query, console	distributor, ingester, query, querier, compactor
C11.01	2012 veröffentlicht, aktuelle Version 2.24.2	2017 veröffentlicht, aktuelle Version 1.50.0	2020 veröffentlicht, aktuelle Version 2.2.3
C11.02	Vor Microservices-Architektur	Microservice, cloud-native	Microservice, speziell für Tracing
C11.03	CNCF-Gemeinschaft; 160 Github-Beitragende	Zipkin-Team; 290 Github-Beitragende	Grafana-Team; 170 Github-Beitragende

Abbildung 14: Vergleich der Analysetools Zipkin, Jaeger, Tempo

Das Ergebnis der Evaluation ist in folgender Tabelle visualisiert. Die Erfüllungsgrade wurden bewertet und die dem Grad entsprechenden Punkte vergeben. Die erreichten Punktzahlen wurden mit der Wertigkeit der Priorität verrechnet und addiert. Im Gesamtergebnis erzielt Tempo die meisten Punkte, nachfolgend Zipkin, an dritter Stelle Jaeger.

dotSource®

Tools			Zipkin	Jaeger	Tempo
Nr.	**Kriterium**	**Prio.-pkt.**			
C01	Anbindung an bestehende Infrastruktur	1	1	1	2
C02	Anbindung an bestehende Entwicklungsprozesse	2	2	2	1
C03	Hostingmöglichkeiten	1	1	1	2
C04	Lizenzart	1	1	1	2
C05	Herstellerneutralität	2	2	2	2
C06	Möglichkeiten des Datentransfers zwischen Komponenten	2	1	2	2
C10.01	OpenTelemetry-Kompatibilität	2	3	3	3
C10.02	Unterstützter Storage	3	2	1	3
C10.03	Möglichkeiten des Betriebs (Deployment Arch.)	3	1	2	2
C10.04	Möglichkeiten des Span-Transports	2	2	1	1
C10.05	Unterstützte Sampling-Strategien	2	1	2	3
C10.06	Ressourcenverbrauch	3	3	2	1
C10.07	Komplexität (Anzahl bestehender Komponenten)	2	3	1	2
C11.01	Reifegrad (Erstrelease und Versionsfortschritte)	1	2	1	2
C11.02	Architektur (Systemdesign)	3	2	3	2
C11.03	Entwicklergemeinschaft (Contributor)	2	3	1	2
			57	54	60

Tabelle 5: Punktevergabe und Gesamtergebniserrechnung der Nutzwertanalyse

6 Beispielintegration

6.1 Umgebung und Servicetopologie

Im Folgenden wird Grafana Tempo als Analysetool für den Einsatz praktisch getestet. Als Infrastruktur dient Microsoft Azure mit einem Managed Kubernetes Service. In diesem wird eine Beispielanwendung, bestehend aus mehreren Microservices, implementiert und damit ein produktives Shopsystem simuliert. Die Implementierung erfolgt über Helm, der als Paketmanager ein Helm Chart startet und Workloads in der containerisierten Umgebung ausführt. Die Topologie der Microservices ist folgend dargestellt.

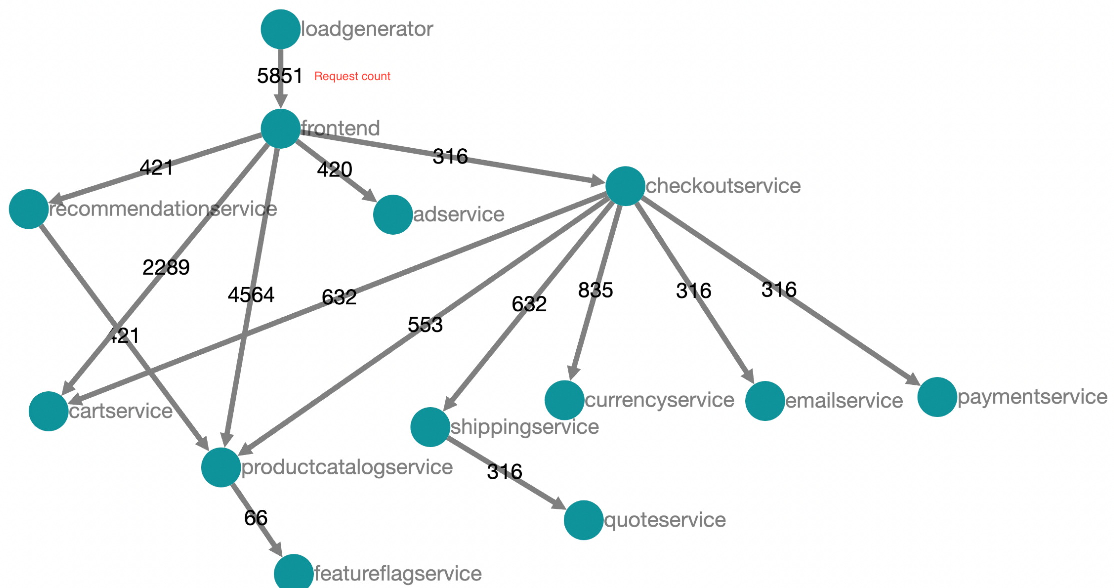

Abbildung 15: Microservice-Topologie des Integrationsbeispiels

6.2 Tempo installieren

Zuerst wird ein eigenes Kubernetes Namespace erstellt und das Grafana Helm Repository hinzugefügt. Bevor das Grafana Tempo Helm Chart installiert wird, müssen noch einige Konfigurationen vorgenommen werden. Dazu gehört zu Beginn das Anlegen eines eigenen Konfigurations-Files für das Helm Chart. Dieses befindet sich in Anlage 1. Im Anschluss werden Storage Optionen in in diesem gesetzt. Dort werden die Trace-Daten gespeichert. Die Optionen sind in Anlage 2 nachvollziehbar. Hier wird MinIO als einfache Lösung verwendet. Anhand der Einblicke in die Konfiguration ist die Erweiterung bzw. der Wechsel auf einen Cloud Storage wie Amazon S3, Azure Blob Storage oder Google Cloud Platform nachvollziehbar. Nun werden Trace receiver gesetzt. Dafür kommt OTLP von OpenTelemetry zum Einsatz. Tempo unterstützt auch Jaeger, Zipkin und Kafka. Die in Anlage 3 befindliche Konfiguration aktiviert OTLP. Auch diese wird dem eigenen Konfigurations-File angehängt. Nun wird das Tempo Helm Chart mit der Konfigurationsdatei in das angelegte Namespace installiert. Bei Erfolg ist der Text wie in Anlage 4 zu sehen. Nun kann mit dem Kommando „kubectl -n tempo-test get pods“ der Status der gestarteten Workloads ausgegeben werden. Dieser kann dem in Anlage 5 ähneln.

6.3 Tempo als Microservice

Nun folgt das Konfigurieren, um Traces schreiben zu können und sie später über Grafana abfragen zu können. Es wird ein Code Snippet im existierenden Grafana Agent eines Microservice hinzugefügt, welches das Schreiben von Traces ermöglicht. Dieses Snippet ist in Anlage 6 enthalten. Die Konfiguration wird mit „kubectl apply --namespace default -f agent.yaml“ gespeichert und der Grafana Agent deployt. Nun muss im Grafana über das Benutzerinterface unter Datenquellen Tempo als Datenquelle hinzugefügt werden, der Endpunkt gesetzt werden und gespeichert werden. Nach erfolgreichem Hinzufügen können nun über die Explore Page von Grafana Traces visualisiert werden.

Nun folgt die Generierung von Tracing-Testdaten über OpenTelemetry's „telemetrygen". Nach dessen Installation können Traces beispielsweise wie folgt generiert werden: „telemetrygen traces --otlp-insecure --rate 20 --duration 5s grafana-agent-traces.default.svc.cluster.local:4317". Dieses Kommando sendet fünf Sekunden lang Traces an Grafana Agent mit einer Rate von 20 Traces pro Sekunde. Optional können die Traces auch direkt an die Tempo-Datenbank gesendet werden, ohne Grafana Agent als Collector zu verwenden. Dafür wird der Endpunkt direkt auf den Tempo Distributor gesetzt. Über das Explore-Menü in Grafana können in der Tempo-Datenquelle nun die erzeugten Traces gesehen werden.

7 Notwendige Anpassungen in der Entwicklung und im Betrieb von Webapplikationen

Anhand der Integration und einiger nachträglichen Recherche wird ein Gesamtbild deutlich, wie Tracing in bestehende Umgebungen integriert werden kann. Zuerst sind Tracekonfigurationen vorzunehmen. Dazu gehören Libraries und Protokolldefinitionen. Dann gilt es, die Trace-Pipeline zu konfigurieren mit den Komponenten Receiver, Processor, Exporter und Connector. Hierfür können je nach Microservice die passenden Protokolle und Werkzeuge definiert werden. Im nächsten Schritt beginnt man mit der Definition von Span-Attributen. Hierbei werden automatisch generierte Spans zu Beginn bereitgestellt. Diese können nach Bedarf im Nachgang um manuelle erweitert werden. Aus Betriebssicht wird die Implementierung mit dem Export an ein Backend abgeschlossen, welches die Traces empfängt und visualisieren kann.

8 Zusammenfassung

8.1 Rückblick

Zu Beginn dieser Arbeit wurden Grundlagen über Observability geschaffen und die Kernkonzepte in Erfahrung gebracht. Es folgte die Erkenntnis, dass Observability auf drei grundlegenden Bereichen aufbaut, die im Anschluss näher beleuchtet und an einem ersten Praxisbeispiel nachvollzogen wurden. Im Anschluss wurden Tracing-Komponenten im Detail betrachtet und ein erster Ausblick gegeben. Um für den Einsatz ein Tool zu evaluieren wurden Anforderungen aufgestellt, eingegrenzt und gewichtet. Erfüllungsgrade für die Evaluation wurden erabeitet und dann die Evaluation vorgenommen. Das Tool mit den meisten erreichten Gesamtpunkten wurde beispielhaft in einer Cloud-Umgebung aufgesetzt und notwendige Anpassungen in der Entwicklung und im Betrieb adressiert.

8.2 Kritische Nachbetrachtung

Die Nutzwertanalyse könnte bei leicht veränderter Gewichtung bereits ein anderes Ergebnis aufzeigen. Das bedeutet, dass die Unterschiede zwischen den untersuchten Tools teils gering waren oder aber im Verfahren zu gering voneinander unterschieden und bewertet wurden. Die Integration des Tools verdeutlichte, dass es nur einen Teilbereich des Tracings im Gesamtbild der zu tätigenden Einrichtungen und Konfigurationen abdeckt und weitere Komponenten zu integrieren sind, um ein funktionsfähiges Tracing aufzusetzen. Weiterhin wurde während der Implementierung erkannt, dass das Protokoll, das für die Trace-Daten verwendet werden soll, abhängig vom Anwendungs-Stack, dem Tracing-Backend und dem Anwendungsfall zu wählen ist. Bei der Instrumentierung wurde klar, dass zuerst mit automatischer Instrumentierung begonnen werden sollte. Sobald diese vollständig genutzt wird, und Lücken erkannt werden, können diese durch manuelle Instrumentierung erweitert werden.

8.3 Ausblick

Observability ermöglicht DevOps-Teams einen vielversprechenden Ansatz, der über die Fehlererkennung hinausgeht. Sie hilft, Fehler grundsätzlich vermeiden zu können: Entweder durch frühzeitige Erkennung oder durch Behebung der eigentlichen Ursachen. Ursachen als solche erkennen zu können, ist eine der eigentlichen Stärken von Observability. Wenn Entwickler und Administratoren die Observability-Daten von Anfang an im Auge behalten und analysieren, werden sie in der Lage sein, aussagekräftigere Logs zu schreiben, aussagekräftigere Metriken zu finden und diese um geeignete Traces zu erweitern.

Literaturverzeichnis

[Ahm21] Ahmed, K.: „SLIs, SLOs and SLAs“, 2021. Abgerufen von https://roadmap.sh/guides/what-is-sli-slo-sla am 16.11.2023

[Ana23] Anand, A.: „OpenTelemetry and Jaeger | Key concepts, features, and differences“, 2023. Abgerufen von https://signoz.io/blog/opentelemetry-vs-jaeger/ am 16.11.2023

[Arn18] Arnold, E.: „The difference between tracing, tracing, and tracing“, 2018. Abgerufen von https://medium.com/opentracing/the-difference-between-tracing-tracing-and-tracing-84b49b2d54ea am 16.11.2023

[EB20] Ewaschuk, R., Beyer, B.: „Monitoring Distributed Systems“, 2020. Abgerufen von https://sre.google/sre-book/monitoring-distributed-systems/ am 16.11.2023

[Goo23] Google, Inc.: „Online Boutique: Sample cloud-first application with 10 microservices showcasing Kubernetes, Istio, and gRPC“, 2023. Abgerufen von https://github.com/GoogleCloudPlatform/microservices-demo am 16.11.2023

[Hor20] Horovits, D.: „Introduction to Instrumentation with OpenTracing and Jaeger“, 2020. Abgerufen von https://logz.io/learn/opentracing-jaeger-guide-to-instrumentation/#a-instrumentation am 16.11.2023

[Int23] International Business Machines Corp.: „What is Infrastructure Monitoring?“, 2023. Abgerufen von https://www.ibm.com/topics/infrastructure-monitoring am 16.11.2023

[Jan22a] Janani, A.: „9 Best Practices for Application Logging that You Must Know“, 2022. Abgerufen von https://www.atatus.com/blog/9-best-practice-for-application-logging-that-you-must-know/#Know-What-to-Log am 16.11.2023

[Kro19] Kroehling, J.: „A guide to the open source distributed tracing landscape“, 2019. Abgerufen von https://developers.redhat.com/blog/2019/05/01/a-guide-to-the-open-source-distributed-tracing-landscape am 16.11.2023

[Lin21] Lindesy, C.: „Getting Started with OpenTelemetry, Grafana, and More“, 2021. Abgerufen von https://docs.google.com/presentation/d/1Q8J12CWfFzLZBaghihbza5kU18nNvbhKH4TZXVgDkRE/edit#slide=id.gf4fc989415_0_639 am 16.11.2023

[Mar23] Marinelli, C.: „What is infrastructure monitoring and why is it mission-critical in the new normal?“, 2023. Abgerufen von https://www.dynatrace.com/news/blog/what-is-infrastructure-monitoring-2/ am 16.11.2023

[Par20] Paradkar, S.: „Application Monitoring Vs Business Monitoring“, 2020. Abgerufen von https://community.ibm.com/community/user/aiops/blogs/sameer-paradkar1/2020/10/19/application-monitoring-vs-business-monitoring am 16.11.2023

[Par23] Parbel, M.: „Drei Fragen und Antworten: Reibungsloser IT-Betrieb dank Observability“, heise online, 2023

[Pie22] Pietro, G.: „Introduction to OpenTelemetry“, 2022. Abgerufen von https://isitobservable.io/open-telemetry/traces/a-short-guide-to-opentelemetry am 16.11.2023

[Sum19] Sumo Logic, Inc.: „Infrastructure monitoring - definition & overview | Sumo Logic“, 2019. Abgerufen von https://www.sumologic.com/glossary/infrastructure-monitoring/ am 16.11.2023

[The23b] The Linux Foundation: „CNCF Landscape Guide“, 2023. Abgerufen von https://landscape.cncf.io/guide#observability-and-analysis--tracing am 16.11.2023

[The23c] The Linux Foundation: „Introduction“, 2023. Abgerufen von https://www.jaegertracing.io/docs/1.50/ am 16.11.2023

[The23a] The Linux Foundation: „Observability Primer“, 2023. Abgerufen von https://opentelemetry.io/docs/concepts/observability-primer/ am 16.11.2023

[Toz22] Tozzi, C.: „Die drei Säulen der Observability: Logs, Metriken und Traces“, ComputerWeekly, 2022

[Wak23] Wakayama, K.: „Open-Source Tracing Tools: Jaeger Vs. Zipkin Vs. Grafana Tempo“, 2023. Abgerufen von https://codersociety.com/blog/articles/jaeger-vs-zipkin-vs-tempo am 16.11.2023

[Wil17] Wilkie, T.: „The RED Method: key metrics for microservices architecture“, 2017. Abgerufen von https://www.weave.works/blog/the-red-method-key-metrics-for-microservices-architecture/ am 16.11.2023

Anlagenverzeichnis

Anlage 1: Tempo Helm Chart Values VIII

Anlage 2: Storage Options IX

Anlage 3: Trace Reiver setzen IX

Anlage 4: Erfolgreiche Installation des Tempo Helm Charts X

Anlage 5: Ausgabe des Status der Pods XI

Anlage 6: remote_write Konfiguration in existierenden Grafana Agent hinzufügen XII

Anlage 1: Tempo Helm Chart Values

```yaml
---
storage:
  trace:
    backend: s3
    s3:
      access_key: 'grafana-tempo'
      secret_key: 'supersecret'
      bucket: 'tempo-traces'
      endpoint: 'tempo-minio:9000'
      insecure: true
#MinIO storage configuration
minio:
  enabled: true
  mode: standalone
  rootUser: grafana-tempo
  rootPassword: supersecret
  buckets:
    # Default Tempo storage bucket
    - name: tempo-traces
      policy: none
      purge: false
traces:
  otlp:
    grpc:
      enabled: true
    http:
      enabled: true
  zipkin:
    enabled: false
  jaeger:
    thriftHttp:
      enabled: false
  opencensus:
    enabled: false
```

Anlage 2: Storage Options

```
---
storage:
  trace:
    backend: s3
    s3:
      access_key: 'grafana-tempo'
      secret_key: 'supersecret'
      bucket: 'tempo-traces'
      endpoint: 'tempo-minio:9000'
      insecure: true
```

Anlage 3: Trace Reiver setzen

```
traces:
  otlp:
    grpc:
      enabled: true
    http:
      enabled: true
  zipkin:
    enabled: false
  jaeger:
    thriftHttp:
      enabled: false
  opencensus:
    enabled: false
```

Anlage 4: Erfolgreiche Installation des Tempo Helm Charts

```
>  helm -n tempo-test install tempo grafana/tempo-distributed -f
custom.yaml

W0210 15:02:09.901064    8613 warnings.go:70]
spec.template.spec.topologySpreadConstraints[0].topologyKey: failure-
domain.beta.kubernetes.io/zone is deprecated since v1.17; use
"topology.kubernetes.io/zone" instead
W0210 15:02:09.904082    8613 warnings.go:70]
spec.template.spec.topologySpreadConstraints[0].topologyKey: failure-
domain.beta.kubernetes.io/zone is deprecated since v1.17; use
"topology.kubernetes.io/zone" instead
W0210 15:02:09.906932    8613 warnings.go:70]
spec.template.spec.topologySpreadConstraints[0].topologyKey: failure-
domain.beta.kubernetes.io/zone is deprecated since v1.17; use
"topology.kubernetes.io/zone" instead
W0210 15:02:09.929946    8613 warnings.go:70]
spec.template.spec.topologySpreadConstraints[0].topologyKey: failure-
domain.beta.kubernetes.io/zone is deprecated since v1.17; use
"topology.kubernetes.io/zone" instead
W0210 15:02:09.930379    8613 warnings.go:70]
spec.template.spec.topologySpreadConstraints[0].topologyKey: failure-
domain.beta.kubernetes.io/zone is deprecated since v1.17; use
"topology.kubernetes.io/zone" instead
NAME: tempo
LAST DEPLOYED: Fri Feb 10 15:02:08 2023
NAMESPACE: tempo-test
STATUS: deployed
REVISION: 1
TEST SUITE: None
NOTES:
***********************************************************************
 Welcome to Grafana Tempo
 Chart version: 1.0.1
 Tempo version: 2.2.0
***********************************************************************

Installed components:
* ingester
* distributor
* querier
* query-frontend
* compactor
* memcached
```

Anlage 5: Ausgabe des Status der Pods

```
NAME                                   READY   STATUS    RESTARTS   AGE
tempo-compactor-86cd974cf-8qrk2        1/1     Running   0          22h
tempo-distributor-bbf4889db-v8l8r      1/1     Running   0          22h
tempo-ingester-0                       1/1     Running   0          22h
tempo-ingester-1                       1/1     Running   0          22h
tempo-ingester-2                       1/1     Running   0          22h
tempo-memcached-0                      1/1     Running   0          8d
tempo-minio-6c4b66cb77-sgm8z           1/1     Running   0          26h
tempo-querier-777c8dcf54-fqz45         1/1     Running   0          22h
tempo-query-frontend-7f7f686d55-xsnq5  1/1     Running   0          22h
```

Anlage 6: remote_write Konfiguration in existierenden Grafana Agent hinzufügen

```
kind: ConfigMap
metadata:
  name: grafana-agent-traces
apiVersion: v1
data:
  agent.yaml: |
    traces:
        configs:
          - batch:
                send_batch_size: 1000
                timeout: 5s
            name: default
            receivers:
                jaeger:
                    protocols:
                        grpc: null
                        thrift_binary: null
                        thrift_compact: null
                        thrift_http: null
                opencensus: null
                otlp:
                    protocols:
                        grpc: null
                        http: null
                zipkin: null
            remote_write:
              - endpoint: <tempoDistributorServiceEndpoint>
                insecure: true  # only add this if TLS is not used
            scrape_configs:
              - bearer_token_file:
/var/run/secrets/kubernetes.io/serviceaccount/token
                job_name: kubernetes-pods
                kubernetes_sd_configs:
                  - role: pod
                relabel_configs:
                  - action: replace
                    source_labels:
                      - __meta_kubernetes_namespace
                    target_label: namespace
                  - action: replace
                    source_labels:
                      - __meta_kubernetes_pod_name
                    target_label: pod
                  - action: replace
                    source_labels:
                      - __meta_kubernetes_pod_container_name
                    target_label: container
                tls_config:
                    ca_file:
/var/run/secrets/kubernetes.io/serviceaccount/ca.crt
                    insecure_skip_verify: false
```