Universität Leipzig
Wirtschaftswissenschaftliche Fakultät
Wirtschaftsinformatik/ Softwareentwicklung für Wirtschaft & Verwaltung
Prof. Dr. Ulrich Eisenecker

Thema

Optimierung der Visualisierung eines Dashboards für das Microservice-Monitoring

Bachelorarbeit zur Erlangung des akademischen Grades
Bachelor of Science – *Wirtschaftsinformatik B.Sc.*

vorgelegt von: *Urban, Dario*

E-Mail-Adresse: *du11xuma@studserv.uni-leipzig.de*

Leipzig, den 05.11.2021

Abstract

Microservice-Architekturen haben sich mittlerweile etabliert und werden von immer mehr Firmen übernommen. Die erhöhte Komplexität der Microservice-Architektur aufgrund der Verteilung des Systems hat jedoch zur Folge, dass die effiziente und erfolgreiche Administration des Systems erschwert wird.

Ziel der Arbeit ist es, alle notwendigen Metriken für ein effizientes und erfolgreiches Microservice-Monitoring zu identifizieren und auf Basis dieser Erkenntnisse das Linkerd-Dashboard prototypisch weiterzuentwickeln. Hierfür wurde eine Literaturrecherche durchgeführt. Darüber hinaus wurden Praktiker mittels eines Online-Fragebogens befragt. Abschließend wurde die prototypische Weiterentwicklung mithilfe eines halbstrukturierten Interviews evaluiert.

Die Literaturrecherche ergab, dass Central-Processing-Unit (CPU)- und Random-Access-Memory (RAM)-Nutzung, Antwortzeit, Workload, Fehlerrate und Service-Interaktion eine Metrik-Menge sind, mit der Microservice-Architekturen effektiv überwacht werden können. Außerdem konnte konstatiert werden, dass die Darstellung der Metriken hauptsächlich mit Visualisierungen realisiert wird.

CPU- und RAM-Auslastung sind eine sinnvolle Erweiterung des Linkerd-Dashboards, da diese in der Literatur sowie im Fragebogen als wichtige Kennzahlen deklariert und alle anderen als essenziell eingestuften Metriken bereits vom Linkerd-Dashboard abgedeckt werden.

Der Prototyp wurde als gelungen eingestuft, benötigt aber einige kleinere Verbesserungen der Visualisierung, bevor er in der Produktion eingesetzt werden kann.

Schlüsselwörter: Microservices, Microservice-Monitoring, Monitoring-Dashboard, Linkerd

Inhaltsverzeichnis

Inhaltsverzeichnis I
Abbildungsverzeichnis III
Listings IV
Abkürzungsverzeichnis V
1 Einführung 1
1.1 Thematik 1
1.2 Motivation 1
1.3 Ziel der Arbeit 3
1.4 Struktur der Arbeit 3
2 Grundlagen 4
2.1 Microservice-Architektur 4
2.2 Monitoring 6
2.3 Linkerd-Service-Mesh 7
3 Forschungsfragen 9
4 Literaturrecherche 10
4.1 Methodik 10
4.2 Planung 11
4.3 Selektion 12
4.4 Extraktion 13
4.5 Ausführung (Ergebnisse) 13
4.5.1 Anforderungen an das Microservice-Monitoring 13
4.5.2 Metriken 14
4.5.3 Darstellung 18
5 Explorativer Fragebogen 19
5.1 Umsetzung 19
5.2 Ergebnisse 20
6 Resultierende Implementierungsentscheidungen 23
7 Implementierung des Prototyps 24
7.1 Umsetzung 24
7.1.1 Architektur 25
8 Evaluation des Prototyps 27
8.1 Umsetzung 27
8.2 Ergebnisse 27
9 Diskussion 29
Literaturverzeichnis VI

Anhang VIII
Eigenständigkeitserklärung XIX

Abbildungsverzeichnis

Abb. 1: Linkerd-Architektur (In Anlehnung an Linkerd-Dokumentation [o. V. o. D.]) 8
Abb. 2: Phasenmodell der Literaturrecherche nach Okoli (In Anlehnung an [Okoli/Schabram 2010]) 10
Abb. 3: Am wichtigsten erachtete Metriken 20
Abb. 4: Anteil der Befragten, die die jeweilige Ebene als sinnvoll erachten 21
Abb. 5: Anteil der Befragten, die die jeweilige Ebene für CPU- und RAM-Nutzung als sinnvoll erachten 22
Abb. 6: Linkerd-Dashboard mit Standard-Graph 24
Abb. 7: Linkerd-Dashboard mit modifiziertem Graph 25
Abb. 8: Explorativer Fragebogen 1/2 X
Abb. 9: Explorativer Fragebogen 2/2 XI

Listings

Listing 1: fetchData-Methode mit PromQL-Abfragen der gewünschten Metriken XII
Listing 2: updatesvcMetrics-Methode XIII
Listing 3: Methode createMetricDstructure zur Zusammenführung der Metriken XIV
Listing 4: prettyPodNames Helfermethode XIV
Listing 5: Erstellung der Intervalle in der ComponentDidMount-Methode..... XV
Listing 6: Löschen der Intervalle in der Methode ComponentWillUnmount XV

Abkürzungsverzeichnis

API	*application programming interface*
CLI	*command-line interface*
CPU	*central processing unit*
QoS	*quality of service*
RAM	*random-access memory*

1 Einführung

1.1 Thematik

Gegenstand der vorliegenden Arbeit ist das Monitoring von Microservice-Architekturen, dafür verwendete Dashboards und deren Visualisierungen. Im Vordergrund steht die Auswahl passender Metriken und aufbauend auf diesen, die Optimierung eines Monitoring-Dashboards und dessen Visualisierungen. Ziel ist die Erweiterung des Monitoring-Dashboards, indem die Visualisierungen mit diesen Metriken angereichert beziehungsweise neue Visualisierungen der Metriken implementiert werden. Hierfür wird untersucht, welche Metriken essenziell für ein erfolgreiches Microservice-Monitoring sind. Für diese Arbeit wurde das Linkerd-Dashboard ausgewählt, um als Basis für einen verbesserten Prototyp zu dienen. Linkerd wurde gewählt, da es gegenüber Alternativen wie Istio leichtgewichtiger ist, aber trotzdem ein Service-Mesh mit umfangreicher Funktionalität darstellt, eine umfassende Dokumentation besitzt und die Integration diverser Tools wie Grafana und eigener Prometheus-Instanzen erlaubt. Neben der Auswahl passender Metriken im Hinblick auf deren Informationsgehalt für das Microservice-Monitoring ist zudem zu beachten, dass die Metriken in das Gesamtkonzept des vorhandenen Dashboards passen. Weiterführend ist eine der größten Herausforderungen dieser Arbeit, die Metriken bedacht in die vorhandenen Visualisierungen zu integrieren und diese so zu erweitern beziehungsweise neue Visualisierungen zu entwickeln, sodass die Konsistenz des Dashboards nicht beeinträchtigt wird. Parallel dazu sollen der Informationsgehalt und damit einhergehend die Monitoring-Kapazitäten signifikant erhöht werden, ohne dabei die Übersichtlichkeit des Dashboards zu beeinträchtigen.

1.2 Motivation

Microservice-Architekturen sind mittlerweile weit verbreitet und werden von immer mehr Firmen übernommen (vgl. [Aceto et al. 2013]). Einer der größten Nachteile der Microservice-Architektur ist die Komplexität des Systems und das dadurch bedingte Problem, das System effizient und erfolgreich zu administrieren (vgl. [Dragoni et al. 2016]). Da Microservices aber diverse Vorteile bieten und deshalb eine sehr beliebte Architektur sind, lohnt es sich hier, Forschung zu betreiben und darauf aufbauend Lösungen für die genannten Probleme zu entwickeln.

Um trotz der Komplexität des Systems ein effektives Monitoring zu realisieren, ist es wichtig, dass das verwendete Dashboard eine möglichst umfassende Informationsbasis für diese Herausforderung bereitstellt und diese Informationen übersichtlich und zielorientiert darstellt, um so die Stakeholder beim Administrationsprozess optimal zu unterstützen (vgl. [Mansouri-Samani/Sloman 1993]). Diese zwei Ziele miteinander zu vereinbaren, ist eine der größten Herausforderungen bei der Umsetzung, da mehr Informationen schnell einen Rückgang der Übersichtlichkeit bedingen können. Hier ist es wichtig abzuwägen, inwieweit beide Aspekte vereinbar sind und wie die konkrete Umsetzung auszusehen hat. Sollte die Balance dieser beiden Ziele nicht gelingen, ist die Effizienz des Dashboards massiv beeinträchtigt, da zu wenig Informationen das umfassende Monitoring unmöglich machen. Eine umfassende Informationsbasis ohne Übersichtlichkeit und Struktur kann die Monitoring-Kapazitäten aber ebenso einschränken. Bei der Gewichtung ist der Mangel an Information jedoch etwas kritischer zu bewerten, solange die Darstellung dieser nicht völlig vernachlässigt wird.

Eine weitere Herausforderung betrifft die Informationsbasis selbst. Die Frage nach den richtigen Metriken stellt eine weitere essenzielle Aufgabe im Entwicklungsprozess dar. Hierbei ist zu beachten, dass alle Metriken einen praktischen Mehrwert besitzen, das heißt in der Praxis Anwendungsfälle vorliegen, für die die aufgenommenen Metriken essenzielle Informationen bereitstellen. Darüber hinaus ist zu beachten, dass gegebenenfalls mehrere bereits vorhandene Metriken in Kombination Informationen liefern, die die Einführung einer weiteren Metrik für diesen Anwendungsfall überflüssig machen. Den Informationsgehalt der vorhandenen Metriken vollkommen auszuschöpfen ist wichtig, um die Übersichtlichkeit nicht unnötig zu gefährden.

Sind diese Herausforderung bewältigt, bietet das Dashboard dem Nutzer eine reichhaltige, aber übersichtliche Informationsbasis, um die Microservice-Architektur zu administrieren und Entscheidungen bezüglich deren Weiterentwicklung zu treffen. Da die Lösungen für die zuvor beschriebenen Herausforderungen bei der Erstellung und Weiterentwicklung eines Dashboards nicht trivial sind, ist es sinnvoll, diesen Prozess mit wissenschaftlichen Methoden anzugehen. Außerdem lohnt es sich, besonders aufgrund der zunehmenden Verbreitung der Architektur die Qualitäten des Ansatzes auszubauen beziehungsweise Lösungen für bekannte Probleme zu finden.

1.3 Ziel der Arbeit

Ziel der Arbeit ist herauszufinden, mit welchen Metriken das Linkerd-Dashboard im Hinblick auf seine Monitoring-Kapazitäten verbessert werden kann. Ausgangspunkt hierfür ist die Untersuchung, welche Metriken fundamental für ein effektives Microservice-Monitoring sind und wie diese den Stakeholdern optimal dargestellt werden. Auf diesem Forschungsergebnis lässt sich dann das Linkerd-Dashboard wissenschaftlich fundiert verbessern. Anschließend sollen die gewonnenen Erkenntnisse in einer prototypischen Weiterentwicklung des Dashboards umgesetzt werden. Zum Abschluss soll der Prototyp evaluiert werden, um festzustellen, ob das angestrebte Ziel erreicht werden konnte. Idealerweise steht am Ende dieses Prozesses eine prototypisch weiterentwickelte Version des Linkerd-Dashboards, das die DevOps-Stakeholder beim Microservice-Monitoring effizient unterstützt und im Vergleich zum ursprünglichen Dashboard einen deutlichen Mehrwert bietet.

1.4 Struktur der Arbeit

Im ersten Kapitel werden die Leser*innen in die Thematik der Arbeit eingeführt und es wird die Motivation der Arbeit erläutert. Im zweiten Kapitel werden zentrale Begriffe definiert und die der Arbeit zugrundeliegenden Konzepte erläutert. Das dritte Kapitel behandelt die Forschungsfragen und erläutert deren Hintergrund. Die Forschungsaktivitäten zur Beantwortung der Forschungsfragen sind in den folgenden Kapiteln Gegenstand der Ausführungen. Im vierten Kapitel werden das methodische Vorgehen und die Ergebnisse der Literaturrecherche dargelegt, auf welcher die vorliegende Arbeit fußt. Das folgende Kapitel beschreibt die komplementär zur Literaturrecherche durchgeführte Befragung, insbesondere das Vorgehen sowie die Ergebnisse. Die Zusammenführung der Ergebnisse aus Literaturrecherche und Befragung wird im sechsten Kapitel dargestellt, wobei aus diesen Ergebnissen die Implementierungsentscheidungen für die prototypische Weiterentwicklung des Linkerd-Dashboards abgeleitet werden. Zudem werden diese Entscheidungen ausgeführt und begründet. Das siebte Kapitel erläutert die Implementierung des Prototyps auf Basis der zuvor gewonnenen Erkenntnisse. Im achten Kapitel wird der Prototyp evaluiert und abschließend wird die vorliegende Arbeit im neunten Kapitel diskutiert.

2 Grundlagen

2.1 Microservice-Architektur

Microservices sind ein Konzept der Software-Architektur, basierend auf entkoppelten autonomen Services, die unabhängig voneinander entwickelt, deployed und betrieben werden (vgl. [Wolff 2017]).

Durch die Entkopplung der einzelnen Services im verteilten System und der Kapselung von Funktionalität in den einzelnen Services mit wohldefinierten Schnittstellen für Zugriffe durch andere Services ermöglicht die Microservice-Architektur eine weitgehend isolierte Weiterentwicklung durch unabhängige Teams. Hierbei ist es für die einzelnen Teams nur notwendig, die Schnittstellen der anderen Services zu verstehen. Mittels dieser Architektur der Services werden die zwei Leitsätze der Softwareentwicklung, Modularisierung und Geheimnisprinzip, umgesetzt. Die durch den Architekturansatz gewonnene Autonomie der Teams begünstigt den Ansatz des Continuous-Delivery, wodurch Verbesserungen oder notwendige Änderungen schnell in die Produktivumgebung eingepflegt werden können. Dadurch werden schnellere Fehlerbehebungen ermöglicht, wodurch Mängel im Produktiveinsatz schnell behoben werden können. Die angesprochene Entkopplung sorgt zudem für eine gute Skalierbarkeit und, auf diesem Aspekt aufbauend, die problemlose Erweiterbarkeit des Systems mit neuen Services (Funktionalitäten). Weiterführend ist noch die Ersetzbarkeit zu nennen, durch die eine nachhaltige Entwicklung der Anwendung realisiert wird. Veraltete Microservices können durch neue ersetzt werden, zum Beispiel wenn die Wartung zu kostenintensiv wird und sich eine Refaktorierung nicht mehr lohnt oder neue Technologien verfügbar werden, die eine bessere und einfachere Implementierung des Service ermöglichen.

Da die Services über Schnittstellen miteinander kommunizieren, ist Technologiefreiheit gegeben. Das heißt, Services können in diversen Sprachen und mit unterschiedlichen Frameworks entwickelt werden, ohne die Kompatibilität der Services zu gefährden. Dies ermöglicht die Nutzung der optimalen Werkzeuge für die jeweilige Aufgabe, wodurch suboptimale Lösungen aufgrund technischer Einschränkung ausgeschlossen werden können.

Typische Probleme sind vor allem Ausfälle und die Unerreichbarkeit eines Service aufgrund eines Ausfalls oder wegen Netzwerkproblemen. Aufgrund der Verteilung ist das System robuster gegenüber dem Ausfall einzelner Services. Lösungsansätze für diese typischen

Probleme sind insbesondere die Vorhaltung mehrere Replikate eines Microservice und Überwachungsmechanismen, die automatisiert Fehler erkennen und Services neustarten.

Zudem bietet die Microservice-Architektur gute Voraussetzungen für ein gelungenes Sicherheitskonzept, da die Services isoliert sind und Zugriffsrechte zwischen diesen durch Firewalls eingeschränkt werden können. Des Weiteren kann der Zugriff von außerhalb auf ein Cluster und dessen Services eingeschränkt werden, sodass alle Kommunikation in das Cluster über einen Eintrittspunkt mit Authentifizierung läuft (vgl. [Wolff 2018a]).

Aufgrund der zuvor genannten Vorteile lösen Microservice-Architekturen klassische Client-Server-Architekturen immer weiter ab und werden laut Prognosen in den nächsten Jahren zum Standard im DevOps-Bereich werden (vgl. [Ostler 2021]).

Allerdings bringt die Microservice-Architektur auch einige Nachteile mit sich. Hauptnachteile sind die erhöhte Komplexität des Systems und damit einhergehende Schwierigkeiten, das System effektiv und erfolgreich zu administrieren (vgl. [Dragoni et al. 2016]). Weitere Nachteile sind die Anfälligkeit für Netzwerkausfälle, Latenz, Schwierigkeiten bei strukturellen Änderungen, Probleme bei Code-Abhängigkeiten, das State-Management, die Notwendigkeit erfahrener Administratoren und erhöhte Komplexität beim Testen der Anwendungen (vgl. [Larrucea et al. 2018; Wolff 2018b]).

Da ein Microservice-System in der Regel aus sehr vielen Services besteht, die alle deployt und überwacht werden müssen, entsteht automatisch ein sehr komplexes System. Dies ist nicht nur auf die Masse der Services zurückzuführen, sondern vor allem auf die Interaktion dieser untereinander. Darüber hinaus steigt die Komplexität des Gesamtsystems durch den vom Architekturstil ermöglichten Technologie-Pluralismus weiter an.

Änderungen in einem Microservice sind zwar sehr einfach, strukturelle Änderungen wie die Verlagerung von Funktionalitäten in andere Services sind jedoch eine große Herausforderung und betreffen in der Regel diverse weitere Microservices, die mit den zu ändernden Services interagieren. Außerdem muss der Code aufgrund der Verwendung anderer Technologien in einem der Services gegebenenfalls neu geschrieben werden.

Code-Abhängigkeiten sollten dem Ansatz nach von vornherein komplett vermieden werden, was in der Praxis aber nicht immer vollständig befolgt wird und dadurch einen Ansatzpunkt für Probleme schafft. So kann zum Beispiel eine neuere Version einer Bibliothek auf einem Microservice eingespielt werden, welche nicht mehr mit der älteren Version der Bibliothek auf einem anderen Microservice abwärtskompatibel ist und so für Probleme sorgt. Auch hier

wird wieder deutlich, dass der Ursprung der Herausforderung in der Interaktion der Services liegt. Darüber hinaus ist das Vermeiden von Code-Abhängigkeiten zwar für die Autonomie und den Ansatz damit ein Vorteil, aus einer anderen Perspektive verhindert dieses Konzept aber mögliche Code-Wiederverwendung und sorgt für einen Mehraufwand während der Entwicklung.

Aufgrund der Abhängigkeiten untereinander ist das Testen von Microservices recht aufwändig und kann sich schwieriger gestalten, da die genaue Problemherkunft durch die Kapselung teilweise schwer zu erkennen ist, wenn diese in anderen Microservices liegt.

Ein weiterer großer Nachteil von Microservices ist die Latenz, bedingt durch die Verteilung des Systems. Ist das Netzwerk langsam, hat dies direkte Auswirkungen auf die Applikationen, was bei einer sehr feingranularen Verteilung der Microservice-Architektur besonders fatal sein kann. Dementsprechend muss besonderer Aufwand in die Optimierung des Netzwerks gesteckt werden, um diesem Nachteil zu begegnen. Darüber hinaus muss bei der Entwicklung versucht werden, die Interservice-Kommunikation zu minimieren, was zumindest einen Mehraufwand in der Entwicklung bedeutet, aber in diversen Fällen auch schlicht nicht praktikabel ist.

Im Kontext der Netzwerkproblematik kann es des Weiteren zu Ausfällen kommen, wodurch einzelne Nachrichten nicht ankommen oder aber ganze Microservices nicht mehr erreichbar sind (vgl. [Wolff 2018b]).

Aufgrund all dieser Herausforderungen sind erfahrene Administratoren notwendig, um Microservice-Architekturen effektiv zu betreiben. Hierbei ist nicht nur Wissen über die theoretischen Grundlagen des Ansatzes nötig, sondern auch detailliertes Wissen über das zu administrierende System und dessen Eigenheiten.

2.2 Monitoring

Monitoring im Microservice-Kontext ist nach Benbernou et al. ein Prozess zur Sammlung und Auswertung relevanter Informationen über den Betrieb und die Evolution von service-basierten Applikationen (vgl. [Benbernou et al. 2008]). Diese Kennzahlen, die den Zustand der Microservices widerspiegeln, werden als Metriken bezeichnet.

Es kann beim Monitoring in funktionale und nicht-funktionale Eigenschaften unterschieden werden. Monitoring von funktionalen Eigenschaften betrifft die Services an sich und deren

Performance, wohingegen nicht-funktionale Eigenschaften Quality-of-Service-Aspekte (QoS) wie die Verfügbarkeit der Services abdecken (vgl. [Saralaya/D'Souza 2013]).

Monitoring von verteilten Systemen läuft nach Mansouri-Samani und Sloman im Allgemeinen in vier Schritten ab. Im ersten Schritt werden Daten des Systems gesammelt. Dies übernimmt in der vorliegenden Arbeit das Linkerd-Service-Mesh mithilfe der mitgelieferten Prometheus-Instanz. Im zweiten Schritt werden die erlangten Daten verarbeitet, das heißt, es werden Informationen generiert und dabei gegebenenfalls Filter und Aggregationen auf die Daten angewendet. Für den Prototyp in dieser Arbeit werden die Filter und Aggregationen direkt in der PromQL-Abfrage umgesetzt. Darauffolgend werden die Informationen im dritten Schritt an die Stakeholder weitergeleitet und müssen im vierten Schritt für die jeweiligen Stakeholder angemessen dargestellt werden (vgl. [Mansouri-Samani/Sloman 1993]). Die vorliegende Arbeit beschäftigt sich hauptsächlich mit den Schritten Zwei bis Vier und insbesondere mit der Optimierung des vierten Schritts.

Gezieltes Monitoring ermöglicht unter anderem die Entdeckung von Laufzeitfehlern und Bottlenecks in den Applikationen, die Laufzeitoptimierung des Systems, die informationsgetriebene Administration des Systems und die Aggregation von Informationen zur strukturierten Weiterentwicklung des Microservice-Systems (vgl. [Benbernou et al. 2008]). Monitoring ist also eine essenzielle Methodik, um die zuvor genannten Nachteile bezüglich der Administration von Microservice-Architekturen besser zu bewältigen.

2.3 Linkerd-Service-Mesh

Ein Service-Mesh dient in erster Linie zur Interservice-Kommunikationsverwaltung innerhalb von Microservice-Clustern, wodurch diese vereinheitlicht wird und aus dem Aufgabenbereich der Microservice-Entwicklung fällt. Ein Service-Mesh ist vor allem bei großen und stetig wachsenden Clustern sinnvoll, da so viel Arbeit für die manuelle Definition der Netzwerklogik in jedem Microservice gespart werden kann. Außerdem sind bei der Umsetzung ohne Service-Mesh Fehler in der Interservice-Kommunikationslogik schwerer zu identifizieren, da sie in jedem Microservice verborgen liegen können.

Alle Interaktion mit dem Service-Mesh läuft per Command-Line-Interface (CLI) über die Controller-Komponente, welche ein Application-Programming-Interface (API) bereitstellt. Im Kontext des Service-Mesh wird die Interservice-Kommunikation mit Netzwerk-Proxys (Sidecars) realisiert, welche vom Proxy-Injector des Service-Mesh an jeden Service im

Cluster automatisch angehängt werden. Über diese Proxys läuft die gesamte Interservice-Kommunikation, wobei weitere Kommunikationsregeln manuell definiert werden können. Das heißt, dass Proxy A zum Beispiel nur mit Proxy B kommunizieren darf, jedoch nicht mit C. Die Destination-Komponente stellt die Routing-Informationen für die Linkerd-Proxys bereit. Außerdem sorgt das Service-Mesh für Load-Balancing und leitet Anfragen bei Ausfällen an Replikate des ausgefallenen Service um. Für Sicherheit bei der Interproxy-Kommunikation sorgt die Identity-Komponente, die Zertifikate für die Kommunikation der Proxys untereinander ausstellt.

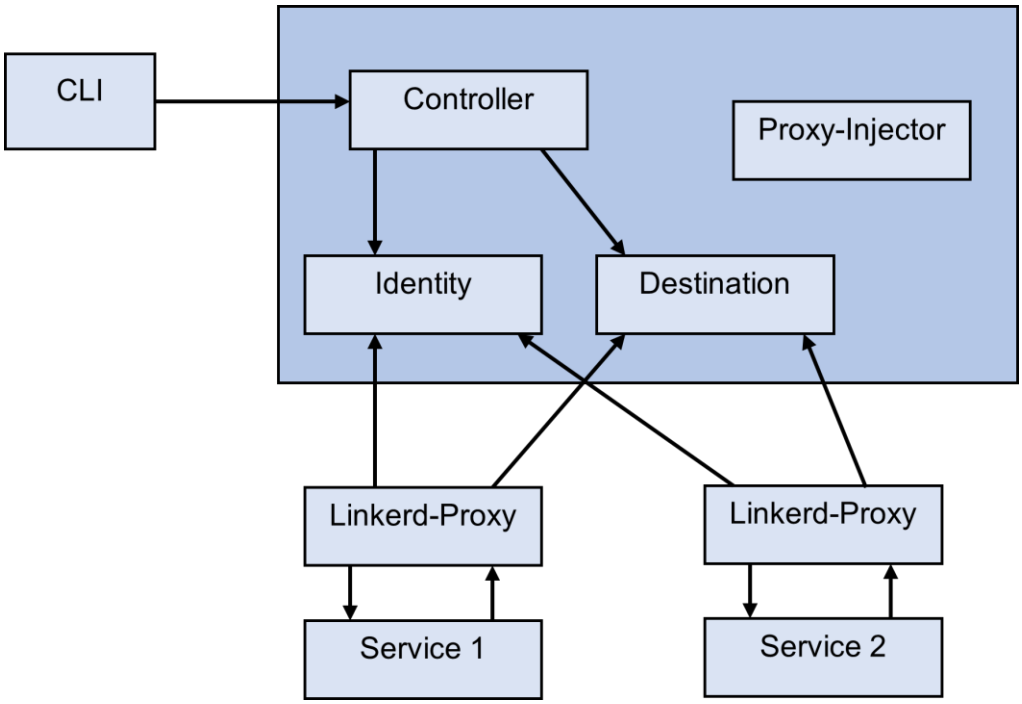

Abb. 1: Linkerd-Architektur (In Anlehnung an Linkerd-Dokumentation [o. V. o. D.])

Da der gesamte Datenverkehr über diese Proxys fließt, werden hier auch die Metriken des Clusters erhoben. In der Regel sammeln die Proxys die Metriken und stellen für jeden Service einen API-Endpunkt namens „/metrics“ bereit. Von diesem bezieht die Prometheus-Instanz regelmäßig die definierten Metriken des jeweiligen Service und speichert diese in einer Datenbank. Prometheus ist ein integriertes externes Tool zur Sammlung von Metriken, jedoch ist es fester Bestandteil des Linkerd-Service-Mesh. Aus dieser Datenbank werden die Daten dann geholt und im Linkerd-Dashboard dargestellt. Dort kann sie der Stakeholder einsehen und auf Basis der dabei gewonnenen Erkenntnisse das System administrieren (vgl. [Morgan 2017; Morgan 2021; o. V. o. D.]).

3 Forschungsfragen

Um die essenziellen Metriken für das Microservice-Monitoring zu erarbeiten und auf einer wissenschaftlich fundierten Wissensgrundlage die Visualisierung des Linkerd-Dashboards zu optimieren, werden folgende Forschungsfragen formuliert:

> F1: Was sind die essenziellen Metriken für ein effektives Microservice-Monitoring?
>
> F2: Was sind sinnvolle Metriken, um die Monitoring-Kapazitäten des Linkerd-Dashboards zu verbessern?
>
> F3: Welchen Mehrwert bietet das verbesserte Dashboard mit den ausgewählten Metriken aus Nutzersicht?

Anhand der Forschungsfragen soll ermittelt werden, welche Metriken für ein gelungenes Microservice-Monitoring notwendig sind. Aufbauend auf diesen Erkenntnissen ist zu untersuchen, welche dieser Metriken das Dashboard dahingehend verbessern, sodass den Nutzer*innen das Monitoring und der Administrationsprozess von Microservice-Architekturen erleichtert werden. Weiterführend soll evaluiert werden, ob die Umsetzung des Dashboard-Prototyps eine Verbesserung darstellt und welcher Mehrwert durch die zusätzlichen Metriken und die Kombination dieser mit den vorhandenen Metriken generiert wird. Teil der Evaluation des Prototyps soll außerdem die Bewertung der Gestaltung und Einbindung der Metriken in das Dashboard sein, wie zufrieden die Nutzer im Allgemeinen mit der Umsetzung sind und ob die Weiterentwicklung in dieser Form in Zukunft Verwendung in der Produktion finden wird.

4 Literaturrecherche

4.1 Methodik

Die methodische Grundlage der Literaturrecherche bildet das Modell der systematischen Literaturrecherche von Okoli und Schabram. Das Vorgehen besteht dem Modell nach aus insgesamt vier Phasen mit jeweils zwei Schritten.

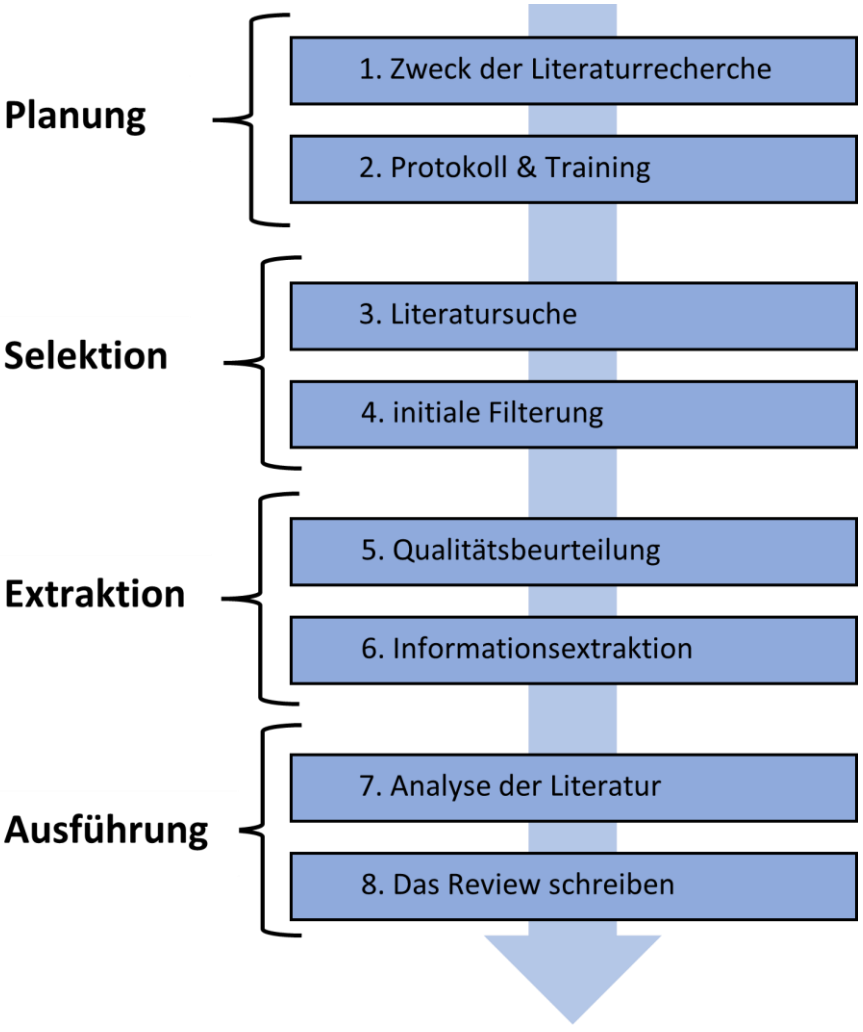

Abb. 2: Phasenmodell der Literaturrecherche nach Okoli (In Anlehnung an [Okoli/Schabram 2010])

Im ersten Schritt der ersten Phase (Planung) werden der Gegenstand sowie der Zweck und das verfolgte Ziel der Literaturrecherche definiert. Im zweiten Schritt wird festgeschrieben, wie die Rahmenbedingungen definiert sind und der Suchraum abgegrenzt wird. Darüber hinaus werden bei Teams noch alle Mitarbeiter dahingehend trainiert, dass sie die zuvor geplanten Bedingungen bei der Recherche kennen und umsetzen können.

In der zweiten Phase (Selektion) wird die Literatur anhand der in der Planung definierten Bedingungen recherchiert und im zweiten Schritt nach ersten Relevanzkriterien entweder aufgenommen oder verworfen.

In der dritten Phase (Extraktion) wird die Literatur eingehend gelesen und auf ihre Qualität geprüft. Hierbei findet eine weitere Aussonderung mangelhafter und unpassender Literatur statt. Die Literatur, die diesen Schritt übersteht, wird im Anschluss anhand ihrer Qualität und ihrem Informationsgehalt in eine Rangliste eingeordnet, um eine Hierarchie der Quellen zu realisieren. Anschließend werden die Informationen extrahiert und festgehalten.

In der vierten Phase (Ausführung) werden die Erkenntnisse analysiert, indem sie integriert werden und Bezug zu der Fragestellung hergestellt wird (vgl. [Okoli/Schabram 2010]).

4.2 Planung

Gegenstand der Literaturrecherche ist die Sammlung vorhandener Erkenntnisse zum Thema Microservice-Monitoring, den dabei zu verwendenden Metriken und der Darstellung dieser Informationen für die Stakeholder. Damit wird das Ziel verfolgt, eine umfangreiche Wissensbasis zu generieren, auf deren Grundlage die Forschungsfragen F1 und F2 beantwortet werden können. Die Literaturrecherche bündelt den momentanen Wissensstand zur genannten Thematik und bildet somit das Fundament für spätere Design-Entscheidungen bei der prototypischen Weiterentwicklung des Linkerd-Dashboards, indem herausgearbeitet wird, wie effizientes Microservice-Monitoring auszusehen hat.

Dafür werden die drei Teilgebiete Anforderungen an Systeme zum Microservice-Monitoring, Metriken für effektives Microservice-Monitoring und optimale Darstellung der Metriken des Themenkomplex Microservice-Monitoring untersucht. Damit soll eine konkrete Beantwortung der Frage realisiert werden, welche Metriken für effizientes Microservice-Monitoring notwendig sind und wie diese Metriken den Stakeholdern präsentiert werden sollen. Die Anforderungen an Systeme zum Microservice-Monitoring werden dahingehend untersucht, inwieweit diese einen Einfluss auf die anderen beiden Teilgebiete der Untersuchung haben. Dabei ist die Trennung eher konzeptionell, in der Praxis werden vermutlich viele Arbeiten Erkenntnisse zu mehreren der definierten Teilgebiete beinhalten.

Die Literaturrecherche bezieht sich explizit nur auf das Monitoring von Microservice-Architekturen und dafür verwendete Metriken. Auch bei der Darstellung werden nur Studien

berücksichtigt, welche die Darstellung von Informationen von Microservice-Systemen behandeln.

Als Literatur werden im Kontext dieser Arbeit Artikel aus wissenschaftlichen Fachzeitschriften oder von Konferenzen, Reviews und Fachbücher bezeichnet. Die Literatur wird mittels Google Scholar, IEEE Xplore und des Online-Katalogs der Leipziger Universitätsbibliothek zusammengetragen. Hierbei diente der Online-Katalog der Universität nur dem Zugriff auf die Volltexte gefundener Literatur. Der Fokus liegt aufgrund der Mehrheit der Veröffentlichungen in Englisch auf englischer Literatur, jedoch werden auch deutsche Arbeiten aufgenommen.

Für die Literaturrecherche werden die zuvor konkretisierten Datenbanken mit folgenden Schlüsselwörtern durchsucht:

- Microservice monitoring
- Microservice monitoring requirements
- Microservice monitoring metrics
- Microservice metrics
- Microservice dashboard

Zusätzlich wird an alle Schlüsselwörter noch das Erweiterungsschlüsselwort Review angehangen. Ziel dieser Maßnahme ist das Auffinden von bereits aggregierten Informationen zu den oben genannten Themenbereichen.

Ziel ist es, so eine Literaturbasis mit passenden Texten zu schaffen und aus diesen mittels Rückwärtssuche weitere relevante Literatur zu extrahieren.

4.3 Selektion

Im ersten Schritt der Selektion wird die Literatur nach den in der Planung festgelegten Kriterien gesucht. Das heißt, es wird in den zuvor festgelegten Datenbanken eine Suche mit den zuvor definierten Schlüsselwörtern durchgeführt. Hierbei wird die Sichtung auf die ersten fünf Seiten der Suchergebnisse beschränkt, dies entspricht den ersten 50 Suchergebnissen für Google Scholar und den ersten 125 Suchergebnissen für IEEE Xplore. Die ersten fünf Seiten wurden gewählt, da bei der Sichtung klar wurde, dass in der Regel ab Seite drei keine relevanten Artikel mehr für die entsprechende Suchanfrage gelistet werden.

Darauf aufbauend wird im zweiten Schritt die gefundene Literatur nach Relevanz sortiert. Kriterien für die Bewertung der Relevanz der Literatur für den Gegenstand der

Literaturrecherche sind der Titel und die Zusammenfassung. Literatur, die nicht relevant ist oder zu weit vom Kernthema abweicht, wird aussortiert.

Nachdem im Schritt zuvor relevante Literatur herausgearbeitet wurde, wird in diesen Texten eine Rückwärtssuche durchgeführt und so weitere relevante Literatur herausgearbeitet, welche ebenfalls mithilfe der oben genannten Kriterien auf Relevanz untersucht wird. Dieser iterative Vorgang wird durchgeführt, bis die Rückwärtssuche keine neue, als relevant eingestufte Literatur mehr hervorbringt. Nach dieser Phase wurden 29 Quellen als relevant eingestuft.

4.4 Extraktion

Die Literatur, welche in vorherigen Selektionsschritt ausgewählt wurde, wird nun eingehend betrachtet und erneut auf ihre Qualität und Relevanz geprüft. Qualitätsmerkmale sind hierbei die Relevanz für die Fragestellung, Informationsgehalt und methodische Qualität der Quellen. Qualitativ mangelhafte oder irrelevante Quellen werden aussortiert und alle verbleibenden anhand der genannten Qualitätsmerkmale in eine Rangliste gebracht. Nach diesem Schritt verblieben noch 14 relevante Quellen für die weitere Extraktion (siehe Anhang A).

Im zweiten Schritt der Extraktion werden die verbleibenden Quellen mithilfe von Citavi-Kategorien thematisch gruppiert und anschließend die Informationen der Quellen extrahiert.

4.5 Ausführung (Ergebnisse)

In der letzten Phase werden die extrahierten Informationen aus dem vorherigen Schritt aggregiert und nach den zuvor definierten Themenbereichen dargestellt. Hierbei werden Erkenntnisse der Quellen in Bezug zueinander gesetzt, interpretiert und vor dem Hintergrund der vorliegenden Arbeit und deren Fragestellung erläutert.

In der zusammengetragenen Literatur gibt es diverse Erkenntnisse zum Monitoring von Microservice-Systemen, den dabei zu verwendenden Metriken und der Darstellung dieser.

4.5.1 Anforderungen an das Microservice-Monitoring

Monitoring mithilfe von Metriken des verteilten Systems ist essenziell, um Entscheidungen bezüglich des Managements des Systems treffen zu können und bei Fehlfunktion gerichtete

Behebungen dieser durchzuführen (vgl. [Mansouri-Samani/Sloman 1993]). Genauer gesagt können mit Microservice-Monitoring die folgenden vier Aspekte bewältigt werden (vgl. [Benbernou et al. 2008; Saralaya/D'Souza 2013]).

- Laufzeitfehler entdecken und zielgerichtet beheben
- Die Laufzeit-Optimierung des Systems
- die zielgerichtete Evolution des Systems
- die Entdeckung kontextueller Veränderungen und Anpassung an diese

Damit diese Aspekte von Microservice-Monitoring-Systemen bewältigt werden können, müssen bestimmte Anforderungen vom System erfüllt werden.

Haselböck und Weinreich haben drei fundamentale Anforderungen an Systeme zum effektiven Microservice-Monitoring identifiziert. Die drei Anforderungen an das Monitoring-System sind:

- die Bereitstellung von Infrastruktur-Informationen zur Laufzeit
- Informationen zu einzelnen Services
- Informationen zur Interservice-Interaktionen

Die drei Anforderungen werden durch Metriken zum Zustand der Infrastruktur wie Host-Metriken, System-Metriken für jeden Service wie CPU- und RAM-Nutzung, Fehlerrate, Antwortzeit und Metriken zur Analyse der Interaktion von Services durch Anfragenverfolgung befriedigt (vgl. [Haselböck/Weinreich 2017; Ma et al. 2019]).

Für das Monitoring von dynamischen Systemen ist es notwendig, immer aktuelle Daten zur Systembewertung zur Verfügung zu haben, weshalb alle Metriken in Echtzeit bereitgestellt werden müssen. Hierfür gibt es diverse Herangehensweisen und Implementierungen. Der Schlüsselaspekt des Laufzeit-Monitorings ist aber die Aktualität der Daten und entspricht dadurch einer Wiedergabe des Systemzustands in Echtzeit (vgl. [Delgado et al. 2004]).

4.5.2 Metriken

Aus diesen Erkenntnissen über das Microservice-Monitoring lässt sich schließen, das Monitoring der richtigen Metriken eine probate Herangehensweise ist, um den Administrationsprozess im DevOps-Bereich zu unterstützen. Um das zuvor abgesteckte Ziel der Literaturrecherche zu erreichen, ist nun zu klären, welche Metriken essenziell für ein funktionierendes Microservice-Monitoring sind.

Im Kontext von Microservice-Monitoring gibt es zwei große Teilbereiche, in die Metriken unterteilt werden können. Zum einen die System- beziehungsweise Ressourcen-Metriken wie Disk-I/O, CPU- und RAM-Nutzung, zum anderen die Anwendungs-Metriken wie Antwortzeit oder Fehlerrate. Beide Teilbereiche sind wichtig für effektives Monitoring, jedoch kann es schnell zu Performance-Problemen kommen, wenn zu viele Metriken prozessiert werden. Hier setzen Thalheim et al. an und stellen einen Ansatz vor, um vorhandene Metriken zusammenzufassen, ohne die statistische Äquivalenz zu den Informationen der ursprünglichen Metrik-Menge zu verlieren (vgl. [Thalheim et al. 2017]). In der vorliegenden Bachelorarbeit wird hingegen versucht, die Performance durch gezielte Auswahl von Metriken zu erhalten und den Informationsgehalt für die Stakeholder zu maximieren.

In der Literatur werden vom Umfang her sehr unterschiedliche Metrik-Mengen verwendet. In einer oft zitierten Studie von Mayer und Weinreich werden die System-Metriken CPU- und RAM-Nutzung, Antwortzeit, Workload und Fehlerrate von den Befragten als besonders wichtig eingestuft. Darüber hinaus werden Service-Interaktion und Abhängigkeiten zwischen Services sowie die Anzahl der Replikate jedes Service als wichtige Kennzahlen für ein effektives Monitoring benannt (vgl. [Mayer/Weinreich 2017]). Die hier vorgestellte Metrik-Menge kann als der Standard für effektives Microservice-Monitoring angesehen werden, da mit diesen Metriken alle drei Anforderungen an Systeme zum Microservice-Monitoring befriedigt werden (vgl. [Haselböck/Weinreich 2017]).

Die Auswahl an Metriken ist sehr divers. In vielen Arbeiten fehlt aber eine umfassende Metrik-Menge, da die Anforderungen für bestimmte Forschungsziele begrenzter sind als die Anforderungen in der Praxis. So verwenden Ma et al. nur die Metriken Service-Fehlerrate, service-interne Fehler, mittlere Service-Antwortzeit, Anteil der verzögerten Antwortzeiten und Service-Nutzung. Auf Basis dieser Metriken können regelmäßig auftretende interne Fehler, Services mit zu langer mittlerer Antwortzeit und Versionen von Services mit zu geringer Nutzung identifiziert werden (vgl. [Ma et al. 2019]).

Diese schon sehr begrenzte Metrik-Menge ist im Vergleich zu den verwendeten Metriken der Arbeitsgruppen um Noor oder Wang noch recht umfangreich. So beschränken sich Noor et al. bei ihrer Arbeit zum Monitoring von service-basierten Anwendungen auf die drei Metriken Latenz, CPU- und RAM-Nutzung (vgl. [Noor et al. 2019]). Wang et al. verwenden sogar nur die Metriken CPU-Nutzung und Antwortzeit für ihr Microservice-Monitoring-Framework (vgl. [Wang et al. 2020]).

Bei diesen minimaleren Ansätzen ist die Vernachlässigung expliziter Informationen in Form passender Metriken zur Befriedigung aller drei Anforderungen an Systeme zum effektiven Microservice-Monitoring (vgl. [Haselböck/Weinreich 2017]) kritisch zu sehen. Dabei ist aber auch zu beachten, dass die meisten Autoren sich der Beschränkungen ihrer Ansätze für die Praxis bewusst sind. Zum Beispiel sehen unter anderem Wang et al. ihre Arbeit als Durchstich, auf dessen Basis in Zukunft ein vollumfängliches Monitoring-Tool entwickelt werden kann (vgl. [Wang et al. 2020]). So belegen viele der Arbeiten die Wichtigkeit einzelner Metriken zum Microservice-Monitoring und qualifizieren diese für die praktische Anwendung. Für den praktischen Anwendungsfall ist jedoch die Integration dieser Metriken und der Erkenntnisse über sie notwendig, um ein umfassendes Microservice-Monitoring zu realisieren.

Pina et al. haben ihren Fokus auf die Interservice-Kommunikation gelegt und einen umfassenden Blick auf die Interservice-Interaktion realisiert, dabei jedoch andere Metriken vernachlässigt. Mit dem von ihnen vorgestellten Ansatz lassen sich Informationen über die Topologie, das Maß an Interaktion zwischen bestimmten Services des Clusters, Antwortzeiten und Informationen über die Last auf einzelnen Services gewinnen. Aus diesen Metriken lassen sich der Arbeitsgruppe nach Dimensionierungsentscheidungen für das Cluster ableiten (vgl. [Pina et al. 2018]). Metriken zur Interservice-Interaktion können über den nicht invasiven „Sniffing-Ansatz“ generiert werden, indem die Kommunikation der Microservices untereinander überwacht und zu Service-Aufrufen Logs erstellt werden, in denen der Response-Code und die Reaktionszeit festgeschrieben sind (vgl. [Cinque et al. 2018]). Die Arbeit von Pina et al. zeigt, dass das Monitoring von Kommunikationsmustern zwischen Services wichtig für effektives Microservice-Monitoring und insbesondere das Aufspüren von Fehlerursachen ist. Jedoch beeinträchtigt das Fehlen von System-Metriken die Monitoring-Kapazitäten im Hinblick auf den Informationsgehalt und die Granularität des Monitorings, was den Stellenwert erklärt, der System-Metriken in den diversen anderen Studien eingeräumt wird (vgl. [Mayer/Weinreich 2017; Noor et al. 2019; Wang et al. 2020]).

Neben diesen eher orthodoxen Ansätzen gibt es auch experimentellere Herangehensweisen, die etwas an Auswertungen im Kontext von Data-Warehouses erinnern. Baresi und Guinea beobachten Grenzwertüberschreitungen, um dann ein „Drill-Down“ mit den Metriken mittlere Antwortzeit der beteiligten Services, Arbeitsspeicher- und CPU-Nutzung für eine genauere Bestimmung der Fehlerherkunft zu realisieren. Der Ansatz verfolgt eine Speicherung der Daten für 24 Stunden, ähnlich dem Linkerd-Dashboard (vgl. [Baresi/Guinea 2013]).

In der Praxis werden die vier „goldenen Signale“ Latenz, Datenverkehr, Fehlerrate und Auslastung als essenziell für das Monitoring von Microservices betrachtet und in Produktivsystemen eingesetzt (vgl. [Beyer et al. 2016]). Die Latenz zeigt an, wie lange ein Service zur Anfragebearbeitung benötigt. Der Datenverkehr wird in der Regel mit der Metrik „Requests per Second“ abgebildet. Für die Auslastung werden System-Metriken bei der Überwachung verwendet. Hier ist zusätzliches Wissen über das System von Vorteil, da zum Beispiel bei Systemen, die vom Arbeitsspeicher begrenzt werden, die Metrik „RAM-Auslastung“ sinnvoll ist, um die Auslastung des Systems zu überwachen. Die Auslastung wird also mit allen für das System essenziellen System-Metriken überwacht. Diese vier Metrik-Kategorien bieten eine annehmbare Monitoring-Abdeckung der Services; für genauere Einsichten sind möglicherweise aber weitere Metriken für bestimmte Use-Cases nötig (vgl. [Beyer et al. 2016]). Alle bisher genannten Quellen betrachten Metriken, die einer der vier Kategorien der „goldenen Signale“ zugeordnet werden können. Diese „goldenen Signale“ werden insbesondere in der Unternehmenspraxis verwendet, unter anderem bei Google (vgl. [Beyer et al. 2016]). Sie decken alle Anforderungen an Systeme zum effektiven Microservice-Monitoring ab, wenn der Datenverkehr mit weiteren Metriken überwacht wird, die ein Tracing ermöglichen und so Interservice-Interaktion abbilden. Ansonsten werden mit den herkömmlichen Metriken nur die Anforderungen zur Bereitstellung von Infrastruktur-Informationen zur Laufzeit und Informationen einzelner Services abgedeckt (vgl. [Haselböck/Weinreich 2017]).

CPU- und RAM-Nutzung, Antwortzeit, Auslastung, Fehlerrate, Service-Interaktion und Abhängigkeiten zwischen Services bilden eine minimale Metrik-Menge, durch die die vier „goldenen Signale“ optimal konkretisiert werden, da sie alle Anforderungen an Systeme zum Microservice-Monitoring abdeckt.

Viele der genannten Metriken werden bereits im Linkerd-Dashboard bereitgestellt, weshalb sich die Menge der angemessenen Metriken für eine Verbesserung automatisch verringert. Wie bereits beschrieben, wurde in vorherigen Arbeiten bereits der Stellenwert von Ressourcen-Metriken konstatiert, um den Deployment- und Monitoring-Prozess zu verbessern (vgl. [Mayer/Weinreich 2017; Haselböck/Weinreich 2017; Noor et al. 2019; Thalheim et al. 2017]). Insbesondere CPU- und RAM-Metriken haben sich schon in monolithischen-Architekturen bewährt und werden aufgrund ihrer Leistungen auch in Microservice-Umgebungen verwendet (vgl. [Pina et al. 2018]). Das Linkerd-Dashboard bietet momentan nur ein Last-Monitoring über die Datenverkehr-Metrik „Requests per Second“. Ressourcen-Metriken wie CPU- und RAM-Nutzung fehlen noch völlig, wodurch

das „goldene Signal“ der Auslastung unzureichend abgedeckt wird. Das heißt, es wird zwar die Auslastung der einzelnen Services mit der Metrik „Requests per Second“ gemessen, jedoch nicht der dabei auftretende Ressourcenverbrauch. Dementsprechend liegt hier Verbesserungsbedarf vor. Antwortzeit, Fehlerrate, Latenz, Service-Interaktion und Abhängigkeiten werden bereits abgedeckt. Außerdem fehlt im Linkerd-Dashboard noch eine Metrik für service-interne Fehler, welche aufgrund der nicht-invasiven Sidecar-Architektur aber nicht erstrebenswert ist, da so eine strikte Kapselung des Applikation-Codes nicht mehr gewährleistet ist.

4.5.3 Darstellung

Der Standard für die Präsentation der Metriken ist momentan die Darstellung der gesammelten Metriken in Dashboards, die einen Überblick über das System bieten und Benachrichtigungen für die Administratoren bereitstellen, sollten gesetzte Schwellenwerte überschritten werden (vgl. [Pina et al. 2018]). Haselböck und Weinrich geben in ihrer Arbeit Empfehlungen ab, welche Metriken den Stakeholdern im Dashboard präsentiert werden sollten, um ein effizientes Monitoring zu realisieren. Dazu gehören service-spezifische Metriken wie Antwortzeit, Fehlerrate und CPU-Nutzung, Präsentation der Infrastruktur wie verfügbare Hosts, Infrastruktur-Metriken wie CPU- und Arbeitsspeicher-Nutzung der Hosts, eine Darstellung der laufenden Services und deren Interaktion sowie eine Funktion zum Rückverfolgen von Aufrufen, um Fehlerquellen zu lokalisieren. Auf die genaue visuelle Umsetzung wird aber nicht genauer eingegangen (vgl. [Haselböck/Weinreich 2017]).

Mayer und Weinrich konzentrieren sich mehr auf die tatsächliche Darstellung der Metriken und stellen ein Dashboard mit mehreren prototypischen Views bereit. Diese Views bieten allgemeine Einsichten über das System, Laufzeit-Informationen, Informationen über einen bestimmten Service, die Möglichkeit, Services zu vergleichen und eine Visualisierung der Service-Interaktion. Hierbei setzen sie vor allem auf grafische Darstellungen in Form von Kreisdiagrammen, Graphen, Balkendiagrammen und Liniendiagramme, lassen aber auch reine Kennzahlen ohne Visualisierung einfließen (vgl. [Mayer/Weinreich 2017]).

Eine von Ma et al. formulierte zusätzliche Anforderung an die Darstellung ist die dynamische Visualisierung der Abhängigkeiten, um den aktuellen Systemzustand realitätsgetreu abzubilden und so das Monitoring zu verbessern (vgl. [Ma et al. 2019]).

5 Explorativer Fragebogen

Die Zielstellung des Fragebogens ist es, komplementär zur Literaturrecherche zu ermitteln, welche Metriken essenziell für ein gelungenes Microservice-Monitoring sind, um mit diesem Wissen die Forschungsfragen F1 und F2 zu beantworten. Die Erkenntnisse aus dieser Befragung sowie der Literaturrecherche dienen dann weiterführend als Basis für die Implementierung des Prototyps.

5.1 Umsetzung

Der explorative Fragebogen wird als Online-Fragebogen (siehe Anhang B) durchgeführt. Hierfür wird auf der Webseite „Umfrage Online“ ein Fragebogen angelegt und ein Einladungslink an alle Mitglieder des DevOps-Teams von Relaxdays GmbH versendet, mit der Bitte, den Fragebogen auszufüllen. An der Befragung nahmen fünf Versuchspersonen teil, was auf die kleine Teamgröße des DevOps-Teams zurückzuführen ist. Zwei der Befragten haben die Umfrage nur teilweise bearbeitet.

Thematisch ist der Fragebogen dahingehend konzipiert, explorativ alle wichtigen Metriken für ein effektives Microservice-Monitoring abzufragen und diese Metriken zu gewichten. Um dieses Ziel abzudecken, werden die Versuchspersonen gebeten, alle ihnen wichtig erscheinenden Metriken für das Microservice-Monitoring aufzuzählen. Aufbauend auf dieser Frage sollen sie dann in der zweiten Frage die aus ihrer Sicht drei wichtigsten Metriken und die bevorzugte Aggregationsebene („Namespace“, „Node“ oder „Service“) für jede der drei Metriken angeben. Ein Namespace ist eine Gruppierung mehrerer Services auf Basis gewählter Kriterien, wie zum Beispiel funktionale Abhängigkeit.

Weiterführend wird erfragt, auf welchen Aggregationsebenen die Metriken zum Microservice-Monitoring im Allgemeinen sinnvolle Einsichten bieten. Hierfür konnten die Befragten eine Mehrfachauswahl treffen. Zur Auswahl standen die Aggregationsebenen „Namespace“, „Node“, „Service“ und „Andere“ mit einem Textfeld für weitere Ebenen.

Auf Basis der Erkenntnisse der Literaturrecherche und dem Abgleich der vorhandenen Metriken im Linkerd-Dashboard wird noch eine Sektion mit Fragen zu CPU- und RAM-Nutzung als Metriken in den Fragebogen aufgenommen. In diesem Abschnitt wird mittels einer Ja/Nein-Frage abgefragt, ob CPU- und RAM-Nutzung relevante Metriken zum Monitoring von Microservices sind. Wenn „Ja“ ausgewählt wird, werden die Befragten in

einer Folgefrage aufgefordert, anzugeben, für welche Anwendungsfälle und Einsichten CPU- und RAM-Nutzung besonders hilfreich sein können. Zuletzt werden auch für CPU- und RAM-Nutzung explizit noch einmal die sinnvollen Aggregationsebenen abgefragt („Namespace“, „Node“, „Service“ und „Andere“).

5.2 Ergebnisse

Als wichtige Metriken für das Microservice-Monitoring wurden Netzwerk-I/O, RAM, CPU, Disk-I/O, Call-Duration, Verfügbarkeit der Endpunkte, Calls nach Antwortstatus und DNS-Lookup-Dauer genannt.

Bei der Auswahl der drei wichtigsten Metriken durch die Befragten wurden Disk-I/O für Nodes, Call-Duration pro Service, Verfügbarkeit ohne Angabe der Ebene, RAM-Auslastung jedes Service, Last ohne Angabe der Ebene, CPU-Auslastung jedes Service und Latenz genannt.

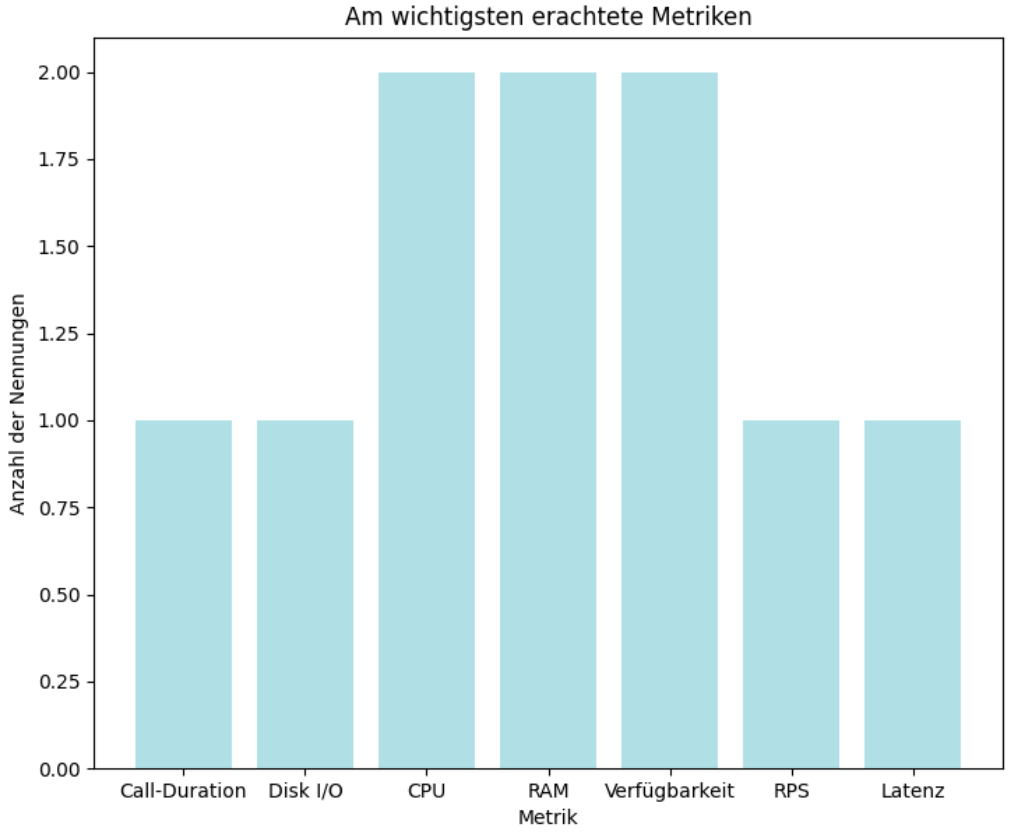

Abb. 3: Am wichtigsten erachtete Metriken

100 % der Befragten geben an, dass Metriken auf Service-Ebene einen sinnvollen informativen Mehrwert beim Microservice-Monitoring bieten. Metriken auf Node-Ebene werden von 66,6 % der Befragten als informativ angesehen. Ebenso fällt das Urteil über Metriken auf Namespace-Ebene aus. Andere Aggregationsebenen wurden von keinem der Befragten angegeben.

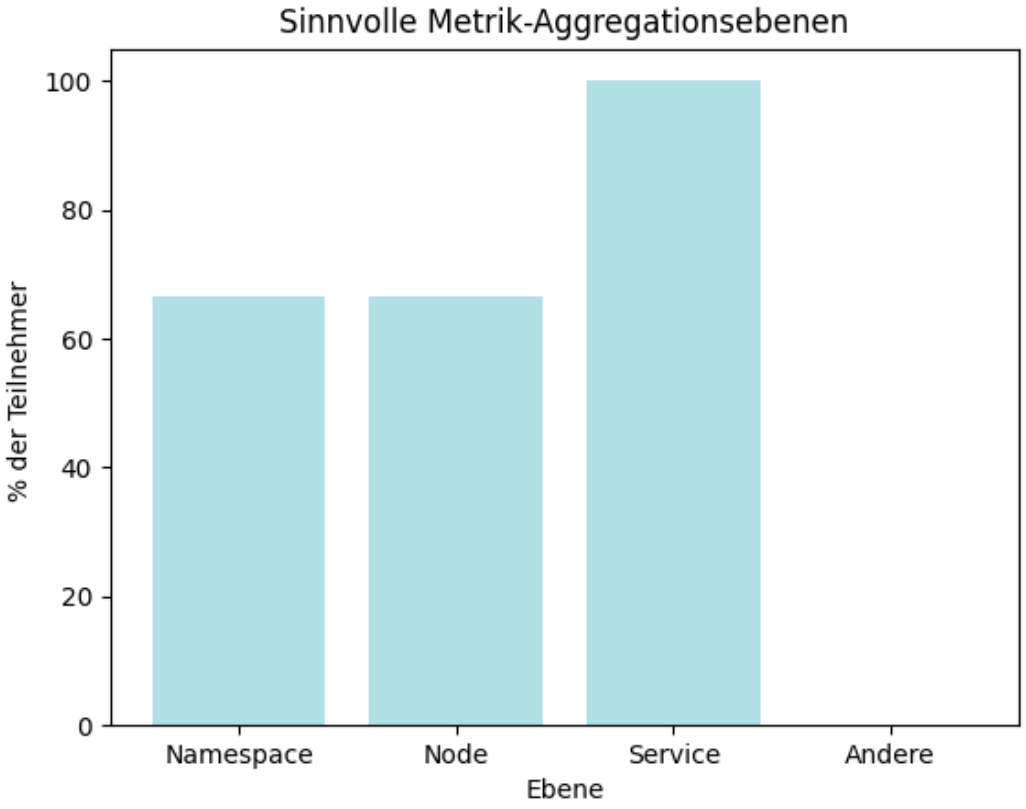

Abb. 4: Anteil der Befragten, die die jeweilige Ebene als sinnvoll erachten

Für alle Befragten sind CPU- und RAM-Nutzung relevante Metriken zum Microservice-Monitoring. Besonders hilfreich in der Praxis sind CPU- und RAM-Nutzung laut der Befragten für die Cluster-Dimensionierung, um Services mit zu hohem Verbrauch zu erkennen, zur Evaluation und Rückverfolgung von Problemursachen und als Unterstützungsmetrik bei Problemen in anderen Metriken für ein besseres Verständnis der Probleme.

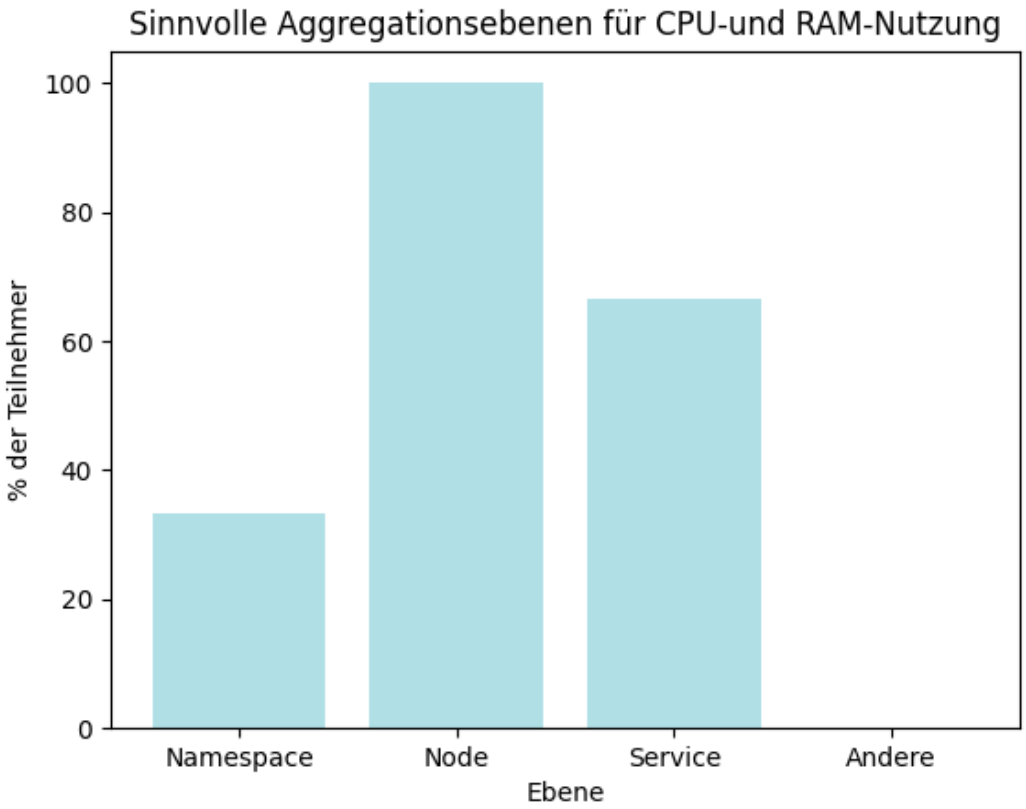

Abb. 5: Anteil der Befragten, die die jeweilige Ebene für CPU- und RAM-Nutzung als sinnvoll erachten

100 % der Befragten geben an, dass CPU- und RAM-Nutzung auf Node-Ebene einen informativen Mehrwert bieten. Noch 66,6 % bescheinigen den Metriken auf Service-Ebene einen informativen Mehrwert und nur 33,3 % auf Namespace-Ebene. Auch hier wurden keine weiteren Aggregationsebenen von den Befragten genannt.

6 Resultierende Implementierungsentscheidungen

Die essenziellen Metriken für Microservice-Monitoring werden bis auf CPU- und RAM-Nutzung bereits vom Linkerd-Service-Mesh abgedeckt. Für die Auslastung bietet das Linkerd-Dashboard momentan nur die „Requests-per-Second“-Metrik an. Da CPU- und RAM-Nutzung in der Literatur sowie im durchgeführten Fragebogen als wichtige Metriken klassifiziert wurden, ist hier der Ansatzpunkt für die Verbesserung des Linkerd-Dashboards zu sehen. Außerdem fehlen noch konkrete Metriken, um die Anforderung „Infrastruktur-Informationen“ bereitzustellen zu befriedigen.

Der Fragebogen hat gezeigt, dass auch Disk-I/O als wichtige Metrik erachtet wird, in der Literatur aber nur wenig auftaucht (vgl. [Thalheim et al. 2017]). Deshalb wird in der vorliegenden Arbeit der Fokus auf die Metriken CPU- und RAM-Nutzung gelegt, jedoch ist Disk-I/O ein möglicher Ansatzpunkt für weitere Verbesserungen in der Zukunft. Um dies mit Sicherheit sagen zu können, sind jedoch noch weitere Untersuchungen nötig.

Als Aggregationsebene für die gewählten Metriken stehen die Service- und Node-Ebene zur Auswahl. In der Literatur werden die Metriken größtenteils für jeden Service gemessen. Des Weiteren werden im Linkerd-Dashboard die vorhandenen Metriken hauptsächlich auf Container (Pod)- oder Service-Ebene dargestellt, mit Namespace als höchster Ebene. Dementsprechend wird die Service-Ebene gewählt, um die Metriken im Prototyp konsistent mit den vorhandenen Metriken zu implementieren.

Da Visualisierungen in Dashboards intensiv genutzt werden und der Nutzen des Dashboards für die Stakeholder dadurch gesteigert wird, wird beschlossen, die Metriken zu visualisieren.

7 Implementierung des Prototyps

Die prototypische Weiterentwicklung des Linkerd-Dashboards dient der Umsetzung der Erkenntnisse aus der Literaturrecherche und dem Fragebogen, um das Linkerd-Dashboard zu verbessern. Das vorliegende Kapitel beschreibt die technische Umsetzung und dessen Ergebnis.

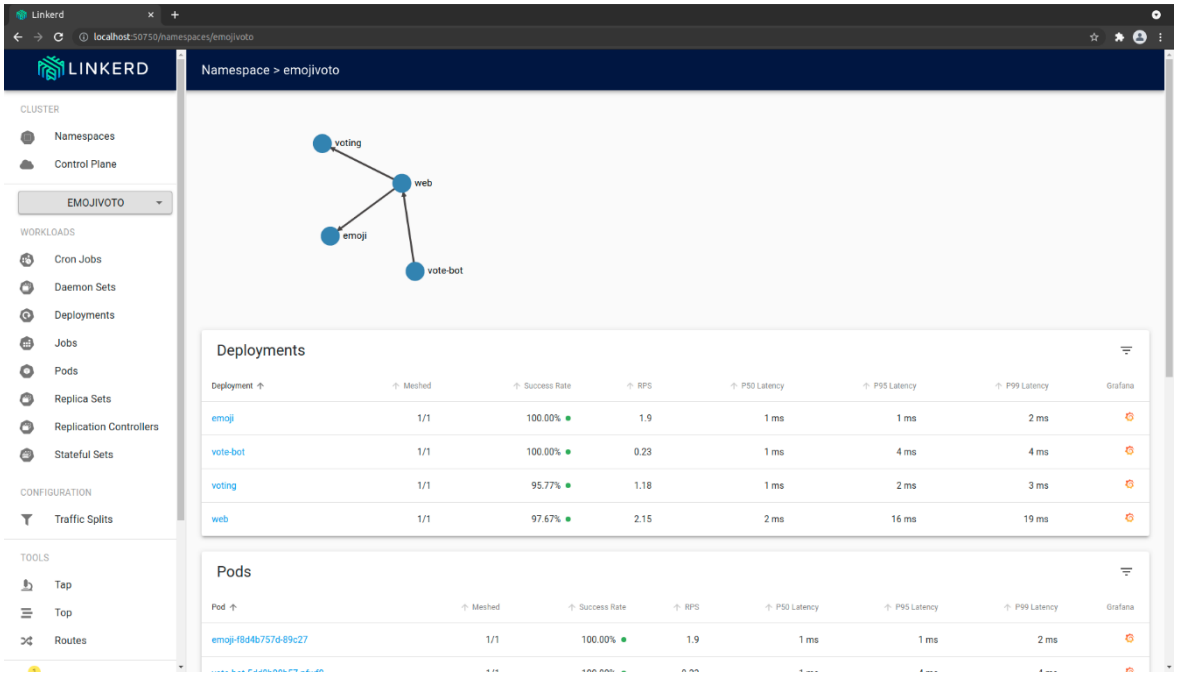

Abb. 6: Linkerd-Dashboard mit Standard-Graph

7.1 Umsetzung

Für die Weiterentwicklung soll der Graph des Dashboards modifiziert werden, um Informationen zu CPU- und RAM-Nutzung bereitzustellen. Hierfür werden die Knoten, welche die einzelnen Services darstellen, je nach Auslastung eingefärbt. Dadurch sind Services mit einem überhöhten Verbrauch schnell zu erkennen und es können gezielt genauere Diagnosen an den betreffenden Services durchgeführt werden. Damit der Graph den tatsächlichen Zustand der Services wiedergibt, liegen der Darstellung die betreffenden Metriken in Echtzeit zugrunde. Die Umsetzung erfolgt in der Programmiersprache ReactJS.

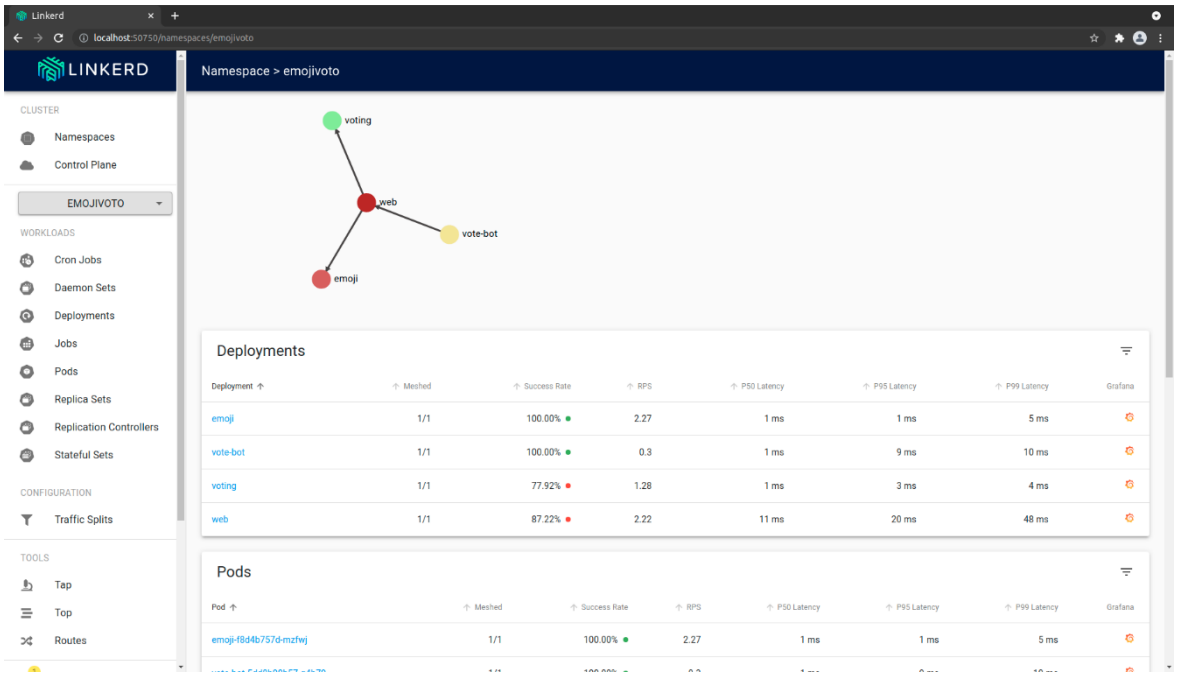

Abb. 7: Linkerd-Dashboard mit modifiziertem Graph

7.1.1 Architektur

Die CPU- und RAM-Nutzung werden mithilfe von zwei PromQL-Anfragen alle fünf Sekunden geholt und im ReactJS-State gespeichert. Mit diesen aktuellen Werten wird dann die prozentuale Auslastung der Services mithilfe der RAM-Limits für den jeweiligen Service berechnet. Die Berechnung der prozentualen CPU-Auslastung findet bereits in der PromQL-Anfrage statt. RAM- und CPU-Limits sind Teil der Yaml-Dateien, mit denen die Services deployt werden und können ebenfalls über PromQL-Abfragen geholt werden. In der `fetchData`-Methode sind alle benötigten PromQL-Abfragen gebündelt. Sie werden genutzt, um die aktuellen Metriken in den ReactJS-State zu laden.

Weiterführend wird dann bei jeder Aktualisierung der Metriken geprüft, ob die Auslastung unter 50 %, zwischen 50 % und 80 % oder über 80 % liegt und die Knoten der einzelnen Services werden dementsprechend eingefärbt. Hierbei wird immer nach der höchsten Auslastung eingefärbt, das heißt ist die CPU-Nutzung > 50 % und RAM-Nutzung < 50 % wird der Service-Knoten gelb eingefärbt.

- CPU oder RAM <= 50 % = grün
- CPU oder RAM > 50 % und < 80 % = gelb
- CPU oder RAM >= 80 % = rot
- CPU und RAM >= 80 % = dunkelrot

Diese Funktionalität wird in der Methode `updatesvcMetrics` realisiert, die die Helfermethode `createMetricDstructure` verwendet, um die einzelnen Rückgaben der PromQL-Abfragen in einem Dictionary zu bündeln und die Einheiten der Metriken anzupassen (siehe Anhang C)

Um zu erreichen, dass die Funktionen immer wieder aufgerufen werden, solange das Dashboard geöffnet ist, werden in der `ComponentDidMount`-Methode Intervalle erstellt, die die betreffenden Funktionen in einem definierten Intervall immer wieder automatisch ausführen.

Damit beim Schließen des Dashboards die Funktionen nicht endlos weiter ausgeführt werden, werden die Intervalle in der Methode `ComponentWillUnmount` gelöscht.

8 Evaluation des Prototyps

Ziel der Evaluation ist es, die Forschungsfrage F3 zu beantworten und im Zuge dessen die Qualität des Prototyps zu evaluieren sowie mögliche Ansätze für Verbesserungen herauszuarbeiten.

8.1 Umsetzung

Zur Evaluation des Prototyps wird ein halbstrukturiertes Experteninterview mit dem Teamleiter der DevOps-Abteilung von Relaxdays GmbH durchgeführt (siehe Anhang D). Hierfür wird erfragt, ob die neu eingefügten Metriken sinnvoll sind und welche neuen Einsichten durch sie gewonnen werden können, welche Tätigkeiten und Diagnosen sie erleichtern beziehungsweise ermöglichen, wie die Verwendung von Echtzeit-Metriken bewertet wird und ob es weitere implementierungswürdige Metriken für das Dashboard gibt.

Weiterführend wird die Visualisierung der Metriken evaluiert und untersucht, welche Verbesserungen vorgenommen werden könnten.

Abschließend wird erfragt, ob der Prototyp gegenüber dem Standard Linkerd-Dashboard bevorzugt und in der Produktion Verwendung finden wird.

8.2 Ergebnisse

Die Metriken CPU- und RAM-Auslastung sind sinnvoll, da die grundlegende Auslastung des Systems mit diesen überwacht werden kann. Besonders für Ressourcen mit Hardcap wie RAM ist die Metrik wichtig, da bei Überschreitung des allokierten RAM der betroffene Container neu gestartet wird. Bei Softcap-Metriken wie CPU ist eine zeitweise Überschreitung der zugeordneten Ressourcen weniger kritisch, sollte aber auch überwacht werden, um gegebenenfalls Anpassungen an den betroffenen Services durchzuführen. Dies ist damit zu begründen, dass die Container mit Softcap nicht neugestartet werden, ihre Reaktionszeit sich aber deutlich verschlechtert. Aufgrund dieser Aspekte werden die beiden Metriken als wertvoll für das Microservice-Monitoring klassifiziert.

Im Allgemeinen erleichtert die Visualisierung das Verständnis des Systems und ermöglicht so eine schnellere Einarbeitung neuer Mitarbeiter. Durch die Erweiterung der Visualisierung können nun Probleme besser lokalisiert werden, wodurch ein Ansatzpunkt für Fehlerbehebungen geboten wird. Dadurch werden das Debugging beziehungsweise die Fehlerbehebung vereinfacht.

Die Verwendung von Echtzeit-Metriken für die Visualisierung wird als sehr gut empfunden. Durch die Darstellung des aktuellen Datenverkehrs werden Anhaltspunkte für momentane, aber auch kommende Probleme generiert.

Es werden keine weiteren Metriken als implementierungswürdig klassifiziert, mit der Begründung, dass die vorhandenen Metriken ausreichen, um Microservice-Cluster zu überwachen und zu viele Metriken eher schaden als nützen.

Die Visualisierung der Metriken ist intuitiv und recht gelungen, jedoch gibt es auch einige Verbesserungsvorschläge. Textüberlagerungen sollten direkt verhindert werden und nicht erst manuell durch Auseinanderziehen des Graphen behoben werden müssen. Des Weiteren wird eine Legende gewünscht, die die Wertebereiche der Einfärbungen wiedergibt.

Wenn diese Verbesserungsvorschläge eingebaut werden, würde die DevOps-Abteilung den Prototyp gegenüber dem Standard-Dashboard bevorzugen und in der Produktivumgebung einsetzen.

Insgesamt ist der Prototyp gelungen, jedoch müssen an einigen Stellen kleine Nachbesserungen vorgenommen werden, bevor dieser in der Produktion eingesetzt werden würde. Die ausgewählten Metriken für die Weiterentwicklung sind sinnvoll, bieten einen informativen Mehrwert und ermöglichen neue Einsichten in das System. Auch die Darstellung der Metriken ist im Kern gelungen, jedoch müssen hier noch die zuvor angesprochenen kleineren Anpassungen vorgenommen werden.

9 Diskussion

CPU- und RAM-Nutzung, Antwortzeit, Workload, Fehlerrate und Service-Interaktion sind eine Metrik-Menge, mit der Microservice-Architekturen effektiv überwacht werden können. Diese deckt alle von Haselböck und Weinreich postulierten Anforderungen an Systeme zum Microservice-Monitoring ab (vgl. [Haselböck/Weinreich 2017]). Deshalb ist diese Metrik-Menge ideal zum Monitoring von Microservice-Systemen und bestärkt nochmal die Ergebnisse von Mayer und Weinrich (vgl. [Mayer/Weinreich 2017]). Die Darstellung der Metriken wird üblicherweise mit Dashboards realisiert, wobei hauptsächlich Visualisierungen zum Einsatz kommen.

CPU- und RAM-Auslastung sind eine sinnvolle Erweiterung des Linkerd-Dashboards, da diese in der Literatur sowie im Fragebogen als wichtige Kennzahlen deklariert und alle anderen als essenziell eingestuften Metriken bereits vom Linkerd-Dashboard abgedeckt werden. Zudem ermöglichen sie Infrastruktur-Informationen, wodurch das Linkerd-Dashboard nun alle Anforderungen an Systeme zum Microservice-Monitoring erfüllt (vgl. [Haselböck/Weinreich 2017]).

Die Evaluation des Prototyps ergab, dass der Informationsgehalt der ausgewählten Metriken positiv bewertet wird. Diese ermöglichen neue Einsichten in das System und erleichtern Tätigkeiten im DevOps-Tätigkeitsfeld wie Fehlerbehebungen. Es werden keine weiteren Metriken als implementierungswürdig angesehen, was die Annahme aus praktischer Sicht noch einmal bestätigt, dass die zuvor genannte Metrik-Menge ideal zum Microservice-Monitoring ist. Die gewählte Visualisierung der Metriken wird als gelungen eingestuft.

Die vorliegende Studie weist einige Einschränkungen auf. Zum einen wurden der explorative Fragebogen sowie die Evaluation des Prototyps nur mit Mitarbeitern von Relaxdays GmbH durchgeführt. Dies kann die Generalisierbarkeit der Ergebnisse des Fragebogens und der Evaluation möglicherweise beeinträchtigen. Zum anderen war die Fragebogenstichprobe zu klein. Außerdem haben zwei Personen den Fragebogen nur teilweise ausgefüllt, was bei einer so kleinen Stichprobe besonders kritisch ist. Dementsprechend ist die Aussagekraft des Fragebogens als beschränkt einzustufen.

Im Allgemeinen wäre es sinnvoll, mehr Forschung im Hinblick auf die Praxis von Microservice-Monitoring durchzuführen. Also insbesondere den Fokus auf alle notwendigen Metriken zu legen und nicht nur einige isoliert zu untersuchen. Dadurch könnte die vorliegende Studie validiert und ihre Ergebnisse weiter gestützt werden.

Literaturverzeichnis

[Aceto et al. 2013] Aceto, G., Botta, A., Donato, W. de, Pescapè, A., Cloud monitoring: A survey, in: Computer Networks, 57 (2013) 9, S. 2093–2115.

[Baresi/Guinea 2013] Baresi, L., Guinea, S., Event-Based Multi-level Service Monitoring, 2013 IEEE 20th International Conference on Web Services, IEEE, 2013, S. 83–90.

[Benbernou et al. 2008] Benbernou, S., Cavallaro, L., Hacid, M.S., Kazhamiakin, R., Kecskemeti, G., Poizat, J.L., Silvestri, F., Uhlig, M., Wetzstein, B., State of the Art Report, Gap Analysis of Knowledge on Principles, Techniques and Methodologies for Monitoring and Adaptation of SBAs (2008).

[Beyer et al. 2016] Beyer, B., Jones, C., Petoff, J., Murphy, N.R., Site reliability engineering, O'Reilly, 2016.

[Cinque et al. 2018] Cinque, M., Della Corte, R., Iorio, R., Pecchia, A., An Exploratory Study on Zeroconf Monitoring of Microservices Systems, 2018 14th European Dependable Computing Conference (EDCC), IEEE, 2018, S. 112–115.

[Delgado et al. 2004] Delgado, N., Gates, A.Q., Roach, S., A taxonomy and catalog of runtime software-fault monitoring tools, in: IEEE Transactions on Software Engineering, 30 (2004) 12, S. 859–872.

[Dragoni et al. 2016] Dragoni, N., Giallorenzo, S., Lafuente, A.L., Mazzara, M., Montesi, F., Mustafin, R., Safina, L., Microservices: yesterday, today, and tomorrow, 2016.

[Haselböck/Weinreich 2017] Haselböck, S., Weinreich, R., Decision Guidance Models for Microservice Monitoring, 2017 IEEE International Conference on Software Architecture Workshops (ICSAW), IEEE, 2017, S. 54–61.

[Larrucea et al. 2018] Larrucea, X., Santamaria, I., Colomo-Palacios, R., Ebert, C., Microservices, in: IEEE Software, 35 (2018) 3, S. 96–100.

[Ma et al. 2019] Ma, S.-P., Liu, I.-H., Chen, C.-Y., Lin, J.-T., Hsueh, N.-L., Version-Based Microservice Analysis, Monitoring, and Visualization, 2019 26th Asia-Pacific Software Engineering Conference (APSEC), IEEE, 2019, S. 165–172.

[Mansouri-Samani/Sloman 1993] Mansouri-Samani, M., Sloman, M., Monitoring distributed systems, in: IEEE Network, 7 (1993) 6, S. 20–30.

[Mayer/Weinreich 2017] Mayer, B., Weinreich, R., A Dashboard for Microservice Monitoring and Management, 2017 IEEE International Conference on Software Architecture Workshops (ICSAW), IEEE, 2017, S. 66–69.

[Morgan 2017] Morgan, W., What's a service mesh? And why do I need one?, 2017, URL: https://linkerd.io/2017/04/25/whats-a-service-mesh-and-why-do-i-need-one/, gelesen am 07.08.2021.

[Morgan 2021] Morgan, J., Introduction to the service mesh—the easy way, 2021, URL: https://linkerd.io/2021/04/01/introduction-to-the-service-mesh/, gelesen am 07.08.2021.

[Noor et al. 2019] Noor, A., Jha, D.N., Mitra, K., Jayaraman, P.P., Souza, A., Ranjan, R., Dustdar, S., A Framework for Monitoring Microservice-Oriented Cloud Applications in Heterogeneous Virtualization Environments, 2019 IEEE 12th International Conference on Cloud Computing (CLOUD), IEEE, 2019, S. 156–163.

[o. V. o. D.] o. V., Architecture, o. D., URL: https://linkerd.io/2.10/reference/architecture/, gelesen am 01.09.2021.

[Okoli/Schabram 2010] Okoli, C., Schabram, K., A Guide to Conducting a Systematic Literature Review of Information Systems Research, in: SSRN Electronic Journal (2010).

[Ostler 2021] Ostler, U., Microservices verbreiten sich schnell, 2021, URL: https://www.datacenter-insider.de/microservices-verbreiten-sich-schnell-a-992899/, gelesen am 13.10.2021.

[Pina et al. 2018] Pina, F., Correia, J., Filipe, R., Araujo, F., Cardroom, J., Nonintrusive Monitoring of Microservice-Based Systems, 2018 IEEE 17th International Symposium on Network Computing and Applications (NCA), IEEE, 2018, S. 1–8.

[Saralaya/D'Souza 2013] Saralaya, S., D'Souza, R., A Review of Monitoring Techniques for Service Based Applications, 2013 2nd International Conference on Advanced Computing, Networking and Security, IEEE, 2013, S. 96–101.

[Thalheim et al. 2017] Thalheim, J., Rodrigues, A., Akkus, I.E., Bhatotia, P., Chen, R., Viswanath, B., Jiao, L., Fetzer, C., Sieve: Actionable Insights from Monitored Metrics in Microservices, 2017.

[Wang et al. 2020] Wang, Z., Xia, Y., Sun, C., Cheng, L., Research on Microservice Application Performance Monitoring Framework and Elastic Scaling Mode, in: Journal of Physics: Conference Series, 1617 (2020), S. 12048.

[Wolff 2017] Wolff, E., Microservices, Addison-Wesley, Pearson, Boston, New York, London, Munich, 2017.

[Wolff 2018a] Wolff, E., Das Microservices-Praxisbuch, dpunkt.verlag, Heidelberg, 2018.

[Wolff 2018b] Wolff, E., Microservices, 2. Aufl., dpunkt.verlag, Heidelberg, 2018.

Anhang

Anhang A: Literaturrecherche

Treffer der jeweiligen Suchanfragen

microservice monitoring: 10.600 (Google Scholar); 293 (IEEE Xplore)

microservice monitoring requirements: 8410 (Google Scholar); 56 (IEEE Xplore)

microservice monitoring metrics: 5420 (Google Scholar); 26 (IEEE Xplore)

microservice metrics: 7840 (Google Scholar); 81 (IEEE Xplore)

microservice dashboard: 2800 (Google Scholar); 8 (IEEE Xplore)

Finales Ergebnis der Literaturrecherche

A Dashboard for Microservice Monitoring and Management, Mayer, B.; Weinreich, R., 2017

An Exploratory Study on Zeroconf Monitoring of Microservices Systems, Cinque, M.; Della Corte, R.; Iorio, R.; Pecchia, A., 2018

A Framework for Monitoring Microservice-Oriented Cloud Applications in Heterogeneous Virtualization Environments, Noor, A.; Jha, D. N.; Mitra, Karan; Jayaraman, P. P.; Souza, A.; Ranjan, R.; Dustdar, S., 2019

A Review of Monitoring Techniques for Service Based Applications, Saralaya, Sridevi; D'Souza, Rio, 2013

A taxonomy and catalog of runtime software-fault monitoring tools, Delgado, N.; Gates, A. Q.; Roach, S., 2004

Decision Guidance Models for Microservice Monitoring, Haselböck, S.; Weinreich, R., 2017

Event-Based Multi-level Service Monitoring, Baresi, L.; Guinea, S., 2013

Monitoring distributed systems, Mansouri-Samani, M.; Sloman, M., 1993

Nonintrusive Monitoring of Microservice-Based Systems, Pina, F.; Correia, J.; Filipe, R.; Araujo, F.; Cardroom, J., 2018

Research on Microservice Application Performance Monitoring Framework and Elastic Scaling Mode, Wang, Z.; Xia, Y.; Sun, C.; Cheng, L., 2020

Sieve: Actionable Insights from Monitored Metrics in Microservices, Thalheim, J.; Rodrigues, A.; Akkus, I. E.; Bhatotia, P.; Chen, R.; Viswanath, B.; Jiao, L.; Fetzer, C., 2017

Site reliability engineering How Google runs production systems, Beyer, B.; Jones, C.; Petoff, J.; Murphy, Niall R., 2016

State of the Art Report, Gap Analysis of Knowledge on Principles, Techniques and Methodologies for Monitoring and Adaptation of SBAs, Benbernou, S.; Cavallaro, L.; Hacid, M. S.; Kazhamiakin, R.; Kecskemeti, G.; Poizat, J. L.; Silvestri, F.; Uhlig, M.; Wetzstein, B., 2008

Version-Based Microservice Analysis, Monitoring, and Visualization, Ma, S.-P.; Liu, I.-H.; Chen, C.-Y.; Lin, J.-T.; Hsueh, N.-L., 2019

Anhang B: Explorativer Fragebogen

Exploration von sinnvollen Metriken zum effektiven Microservice-Monitoring

Seite 1

Hey Team DevOps,
ich schreibe momentan an meiner Bachelorarbeit und möchte untersuchen, mit welchen Metriken sich die Visualisierung des Linkerd-Dashboards verbessern lässt, sodass die Microservice Monitoring-Kapazitäten des Dashboards weiter ausgebaut werden. Hierzu habe ich bereits eine Literaturrecherche durchgeführt und möchte nun noch zusätzlich die Sichtweise von Praktikern in meine Bewertung einfließen lassen, um das Dashboard um die richtigen Metriken zu erweitern.

Hierzu würde ich von euch gerne erfahren, welche Metriken ihr als besonders wichtig für das Microservice-Monitoring erachtet.

Angaben zur Person

Berufsbezeichnung

Alter

Seite 2

Was sind wichtige Metriken für das effektive Microservice-Monitoring? *

mit kurzer Begründung

Welche 3 Metriken erachtest du als besonders wichtig, für ein effizientes Monitoring von Microservice-Architekturen? *

	Bezeichnung	Aggregationsebene (mehrere möglich)
1. Metrik		
2. Metrik		
3. Metrik		

Abb. 8: Explorativer Fragebogen 1/2

Auf welchen Aggregationsebenen bieten Metriken einen sinnvollen informativen Mehrwert?

Mehrfachauswahl möglich

- ☐ Namespace
- ☐ Node
- ☐ Service
- ☐ _____

Seite 3

Sind CPU- und RAM-Nutzung relevante Metriken zum Monitoring von Microservice-Architekturen? *

- ◯ ja
- ◯ nein

Wenn ja, wofür sind sie besonders hilfreich in der Praxis?

Auf welchen Aggregationsebenen bieten CPU- und RAM-Nutzung einen sinnvollen informativen Mehrwert?

Mehrfachauswahl möglich

- ☐ Namespace
- ☐ Node
- ☐ Service
- ☐ _____

» Umleitung auf Schlussseite von Umfrage Online

Abb. 9: Explorativer Fragebogen 2/2

Anhang C: Implementierung des Prototyps

```
1.  fetchData() {
2.
    fetch(`http://localhost:9090/api/v1/query?query=sum(container_memory_working_set_bytes
    {namespace="${this.props.namespace}", container!="linkerd-proxy"}) by (pod)`)
3.        .then(response => response.json())
4.        .then(response => this.setState({ servRam: response }))
5.        .catch(err => console.error(err));
6.
7.
    fetch(`http://localhost:9090/api/v1/query?query=sum(rate(container_cpu_usage_seconds_t
    otal{image!="", namespace="${this.props.namespace}", container!="linkerd-proxy"}[1m]))
    by(pod)`)
8.        .then(response => response.json())
9.        .then(response => this.setState({ servCpu: response }))
10.       .catch(err => console.error(err));
11.
12.
    fetch(`http://localhost:9090/api/v1/query?query=sum(container_spec_memory_limit_bytes{
    namespace="${this.props.namespace}", container!="linkerd-proxy"}) by (pod)`)
13.       .then(response => response.json())
14.       .then(response => this.setState({ servMemLimit: response }))
15.       .catch(err => console.error(err));
16.   }
17.
```

Listing 1: fetchData-Methode mit PromQL-Abfragen der gewünschten Metriken

```
1.  updatesvcMetrics() {
2.      const tmp = this.createMetricDstructure();
3.      this.setState({ svcMetrics: tmp });
4.
5.      _map(this.state.svcMetrics, d => {
6.        if (((parseFloat(d.RamValue) / parseFloat(d.MemLimit)) <= 0.5) || (d.CpuValue <=
    50.0)) {
7.          d3.select(`#${d.pod}`)
8.            .attr('fill', 'LightGreen');
9.        }
10.       if ((((parseFloat(d.RamValue) / parseFloat(d.MemLimit)) > 0.5) &&
    ((parseFloat(d.RamValue) / parseFloat(d.MemLimit)) < 0.8)) || (d.CpuValue > 50.0 &&
    d.CpuValue < 80.0)) {
11.         d3.select(`#${d.pod}`)
12.           .attr('fill', 'Khaki');
13.       }
14.       if (((parseFloat(d.RamValue) / parseFloat(d.MemLimit)) >= 0.8) || (d.CpuValue >=
    80.0)) {
15.         if (((parseFloat(d.RamValue) / parseFloat(d.MemLimit)) >= 0.8) && (d.CpuValue
    >= 80.0)) {
16.           d3.select(`#${d.pod}`)
17.             .attr('fill', 'FireBrick');
18.         } else {
19.           d3.select(`#${d.pod}`)
20.             .attr('fill', 'IndianRed');
21.         }
22.       }
23.     });
24.   }
25.
```

Listing 2: updatesvcMetrics-Methode

```
1.    createMetricDstructure() {
2.      const servRamVar = [];
3.      const servCpuVar = [];
4.      const servMemLimVar = [];
5.      _map(this.state.servRam.data.result, d => {
6.        servRamVar.push({
7.          pod: this.prettyPodNames(d.metric.pod),
8.          RamValue: parseFloat(d.value[1] / 1000000).toFixed(2),
9.        });
10.     });
11.     _map(this.state.servCpu.data.result, d => {
12.       servCpuVar.push({
13.         pod: this.prettyPodNames(d.metric.pod),
14.         CpuValue: parseFloat(d.value[1] * 100).toFixed(2),
15.       });
16.     });
17.     _map(this.state.servMemLimit.data.result, d => {
18.       servMemLimVar.push({
19.         pod: this.prettyPodNames(d.metric.pod),
20.         MemLimit: parseFloat(d.value[1] / 1000000).toFixed(2),
21.       });
22.     });
23.
24.     const cpuRamMerged = servRamVar.map(t1 => ({ ...t1, ...servCpuVar.find(t2 =>
    t2.pod === t1.pod) }));
25.     const Mergedcomplete = cpuRamMerged.map(t1 => ({ ...t1, ...servMemLimVar.find(t2
    => t2.pod === t1.pod) }));
26.
27.     return Mergedcomplete.sort((a, b) => (a.pod > b.pod ? 1 : -1));
28.   }
29.
```

Listing 3: Methode createMetricDstructure zur Zusammenführung der Metriken

```
1.    prettyPodNames(input) {
2.      this.tmp = input.slice(0, input.lastIndexOf('-'));
3.      return this.tmp.slice(0, this.tmp.lastIndexOf('-'));
4.    }
```

Listing 4: prettyPodNames Helfermethode

```
this.fetchData();
this.interval = setInterval(() => { this.fetchData(); }, 5000);
this.intervalUpdate = setInterval(() => { this.updatesvcMetrics(); }, 7000);
```

Listing 5: Erstellung der Intervalle in der ComponentDidMount-Methode

```
  componentWillUnmount() {
    clearInterval(this.interval);
    clearInterval(this.intervalUpdate);
  }
```

Listing 6: Löschen der Intervalle in der Methode ComponentWillUnmount

Anhang D: Experteninterview mit einem DevOps-Teamleiter zur Evaluation des Prototyps

Sind CPU und RAM eine sinnvolle Erweiterung der Dashboard-Metriken?

Ja sind sie, da wir im Kubernetes-Umfeld unterscheiden zwischen Ressourcen die Hard- und Soft-gecapt werden. Und dazu zählen eben genau diese beiden CPU und RAM. Das heißt CPU ist ein soft-Limit und kann Spike-weise mehr CPUs nutzen, als ihm zugeteilt sind. Das ist also alles gut, sollte man aber trotzdem im Auge behalten. Beim RAM ist es so, du kannst im Nachgang nicht einfach mehr RAM allokieren, das heißt, wenn der Pod Gefahr läuft, an seine RAM-Grenze zu stoßen, geht er out of memory und stirbt und wird neu gestartet. Ja, und das kann man damit sehr gut im Auge behalten.

Jetzt hast du ein bisschen vorgegriffen, aber das ist auch nicht schlimm. Ich habe jetzt noch die zweite Frage, sind jetzt Insights möglich, die vorher nicht möglich waren? Und wenn ja, welche wären das? Hast du da noch etwas hinzuzufügen?

Also die beiden sind auf jeden Fall erst mal Grundlage und sehr viel Wert. Was ich jetzt nur noch mal… Durchsatz war ja jetzt schon mit drin oder?

Ja.

Ja genau, das sind eigentlich die Wichtigsten zum monitoren**.**

Das zielt jetzt wahrscheinlich auch auf die 4 golden Metrics ab oder?

Ja genau, also es macht halt keinen Sinn…letztlich ist es ja so, du kannst dir 100 Metriken ausgeben lassen, du prüfst die sowieso nicht alle nach und wirst betriebsblind. Also das ist schon ok.

Ok, die dritte Frage wäre, welche Tätigkeiten und Diagnosen werden durch den Prototyp vereinfacht? Jetzt vielleicht auch noch mal im Hinblick auf das Debugging.

Genau... also genau, aufgrund der Visualisierung fällt es einem Mitarbeiter erst mal leicht, gerade in so einem hochdynamischen Umfeld wie bei uns zu blicken und zu verstehen, welcher Service wie mit welchem kommuniziert. Das fehlt ja bisher gänzlich. Das heißt, ich

kann Leute leichter in den Prozess onboarden, die Leute verstehen das System leichter und auch als Entwickler kann ich das System leichter debuggen, weil ich seh ja in dem Moment wo es hängt. Und so würde ich halt… ohne eine Visualisierung gehe ich halt die ganze Kette durch und habe gar keinen Ansatzpunkt, den kriege ich damit ja schon mal.

Jetzt noch mal eine Frage zu den Echtzeit-Metriken, findest du, dass die Verwendung von Echtzeit-Metriken gut ist oder wäre da eine bessere Alternative irgendwie möglich? Also jetzt auch gerade bezogen auf den Graphen.

Genau. Also ich find das sehr gut, weil man wie schon gesagt sieht, wo gerade viel Traffic reingeht oder wo es sich aufstaut oder welcher Container zu Problemen führen könnte aufgrund von verschiedenen Ressourcen. Das ist sehr wertvoll an der Stelle.

Ok, gibt es weitere Metriken, die du noch als implementierungswürdig ansiehst?

Lass uns kurz nochmal besprechen was ist jetzt drin. Also wir haben CPU, RAM, die zwei, wir haben den Durchsatz

Genau Success Rate und Latenz.

Ja, das sind eigentlich die Wichtigsten. Ja also wie du schon sagst, die vier goldenen und das reicht auch an der Stelle, um zu evaluieren, hab ich Probleme, könnte ich zeitnah in Probleme laufen. Ja genau, also das sollte ausreichen.

Alles klar, dann würde ich jetzt zu der Visualisierung an sich kommen. Wie findest du die Darstellung an sich? Findest du die gelungen und ist sie intuitiv und verständlich?

Sie ist intuitiv verständlich. Sie ist ganz gut gelungen. Was mir jetzt negativ aufgefallen ist, ist, dass wir die Textüberlagerungen haben. Also da sehe ich… Ich kann mir das im Nachhinein natürlich auseinanderziehen, aber wenn ich das so aufmache, sehe ich erst mal, dass die Knoten sich so überlappen und dadurch der Text überlappt ist. Da würde ich mir für später halt wünschen, dass das halt nicht passiert. Ansonsten finde ich das, wenn man das auseinanderzieht, ist das ne gute Übersicht, ja.

Ok, das wäre jetzt eigentlich genau die siebte Frage gewesen. Also ob du Änderungswünsche hast oder Vorschläge zur weiteren Verbesserung. Also zum einen dann natürlich den Graphen noch mal auseinanderziehen beziehungsweise schon auseinandergezogen darstellen. Vielleicht noch irgendwas?

Wichtig ist natürlich, ich weiß jetzt nicht, ob es wirklich so ist, die Konsistenz der Darstellung. Damit ziele ich auf die Farben ab, also dass die dann auch immer gleich sein sollten. Also wenn die Grenzwerte erreicht werden, dass es dann auch bei jedem Graphen korrekt dargestellt wird und auch immer nach diesem Schema.

Ja, das ist so

Hattest du ne Legende drin?

Ne.

Das wäre vielleicht noch ganz hilfreich, weil hättest du jetzt ein Ampelsystem, ist es klar, sage ich mal. Aber selbst da wüsste ich den Wertebereich bei Gelb nicht. So, und jetzt haben wir ja schon mit Hellrot noch vier Farben. Das heißt, ich kenne die Wertebereiche nicht, ab wann springt der über und das wäre für den Betrachter, glaube ich hilfreich zu wissen.

Gut und dann jetzt noch mal eine abschließende Frage. Würdet ihr den Prototyp in der Praxis einsetzen oder gegenüber dem Vanilla-Dashboard bevorzugen?

Wenn der Prototyp mit den eben angesprochenen Erweiterungen kommt, dann ja, dann würde ich den dem Vanilla-Board gegenüber bevorzugen.

Alles klar, das wars dann eigentlich auch schon. Jetzt wäre noch Zeit für weitere Anmerkungen von dir, falls du noch irgendwas hast.

Nö, also eigentlich nicht. Ich finde gut, was du da gemacht hast, ja. Das ist sehr hilfreich an der Stelle. Ansonsten hätte ich eigentlich nix, nö.

Alles klar, ja dann vielen Dank!

Eigenständigkeitserklärung

Urban, Dario

Name, Vorname

Ich versichere, dass ich die Bachelorarbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

Darüber hinaus versichere ich, dass die elektronische Version der Bachelorarbeit mit der gedruckten Version übereinstimmt.

Leipzig, 05.11.2021

Ort, Datum

Unterschrift