Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Bachelorarbeit

Willem-Lennart Draeger

Auswirkungen von Service Meshes auf die Performance von Microservices

Fakultät Technik und Informatik
Studiendepartment Informatik

Faculty of Engineering and Computer Science
Department of Computer Science

Willem-Lennart Draeger

Auswirkungen von Service Meshes auf die Performance von Microservices

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung

im Studiengang Bachelor of Science Angewandte Informatik
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer: Prof. Dr.-Ing. Hamann
Zweitgutachter: Prof. Dr. Sarstedt

Eingereicht am: 17. August 2023

Willem-Lennart Draeger

Thema der Arbeit
Auswirkungen von Service Meshes auf die Performance von Microservices

Stichworte
Service-Mesh, Microservices, Linkerd, Istio, Performance

Kurzzusammenfassung
Diese Bachelorarbeit vergleicht zwei konkrete Service-Mesh-Implementierungen (Linkerd und Istio) bezüglich Latenz und Ressourcenverbrauch. Dazu wird in einem Kubernetes-Cluster eine Test-Applikation deployt und von den Service-Meshes überwacht. Die Test-Applikation wird verschiedenen Testszenarien ausgesetzt um einen Eindruck darüber zu gewinnen wieviel Overhead ein Service-Mesh mit sich bringt.
Willem-Lennart Draeger

Title of the paper
Impact of Service Meshes on the Performance of Microservices

Keywords
service mesh, microservices, Linkerd, Istio, performance

Abstract
This bachelor's thesis compares two specific service mesh implementations (Linkerd and Istio) in terms of latency and resource consumption. For this purpose, a test application is deployed in a Kubernetes cluster and monitored by the service meshes. The test application is subjected to various test scenarios to gain an understanding of the overhead introduced by a service mesh.

Inhaltsverzeichnis

1 Einleitung

1.1 Hintergrund und Motivation

Die moderne Softwareentwicklung hat einen Wandel erlebt, weg von monolithischen Ansätzen hin zu verteilten, auf Microservices basierenden Architekturen. Diese Systeme benötigen effiziente Lösungen für die Orchestrierung und Verwaltung, insbesondere in Cloud-basierten Systemen, wodurch Orchestrierungssysteme an Relevanz gewinnen. Diese Komplexität von Microservice-Architekturen führt zu Herausforderungen in der Kommunikation zwischen Services, der Observability und Reliability. Um mit dieser Komplexität umzugehen, wurden Service Mesh entwickelt. Sie bestehen aus konfigurierbaren Proxies, die für die Kommunikation, Observability, Reliability und Sicherheit der Microservices verantwortlich sind. Service Meshes sollen eine umfassende Networking-Lösung für Microservices sein. Allerdings führen sie durch ihren Architektur auch zu Performance-Overhead. Der Overhead soll in dieser Arbeit genauer untersucht werden. Dazu werden Experimente im Bezug auf Latenz und Ressourcenverbrauch durchgeführt, um einen Eindruck dafür zu erlangen, wie sehr sich der Einsatz eines Service-Meshes auf diese Metriken auswirkt. Um besser zu verstehen warum Service-Meshes entstanden sind, wird im folgenden Kapitel genauer beleuchtet, welche Entwicklung zu Microservice-Architekturen geführt haben.

1.2 Vom Monolithen zur Microservice-Architektur

1.2.1 Monolith

Der Begriff Monolith bezeichnet in der Softwareentwicklung ein Architekturmuster bei dem eine Applikation als zusammenhängende Einheit entwickelt und deployt wird. Dieser Architektur-Stil bringt einige Vorteile mit sich. So ist der Entwicklungsprozess zumindest zu Beginn noch vergleichsweise einfach. Die gesamte Applikation wird in einer Codebase entwickelt, was beispielsweise die Wiederverwendung von Code vereinfacht. Im Gegensatz zu einem verteiltem System, in dem der Code meistens in getrennten Repositories verwaltet wird, kann Code bei einem Monolithen leicht wiederverwendet werden, da es lediglich ein Codebase gibt. Auch

End-To-End-Testing lässt sich bei einem Monolithen leichter umsetzen, da die gesamte Applikation in einem Prozess läuft. Das gleiche gilt für die Fehlersuche [1]. Allerdings bringt dieser Ansatz auch Nachteile mit sich. So kann die wachsende Codebase zu eine Verlangsamung der Entwicklung führen, Der Technologie-Stack der Applikation muss zu Beginn festgelegt werden und auch die Deployment-Zyklen sind schwerer zu koordinieren [1, 2]. Außerdem lassen sich monolithische Anwendungen zwar leicht, aber nicht effizient skalieren. Es kann lediglich eine zweite Instanz deployt werden, um einer erhöhten Nachfrage entgegenzuwirken, selbst wenn die Nachfrage sich nur auf einen kleinen Teil innerhalb der Applikation bezieht. Zudem lassen sich die Ressourcen eines Servers nicht einteilen, was zu einer ineffizienten Nutzung dieser führt [3].

1.2.2 Virtualisierung und Containerisierung

Ein erster Lösungsansatz war die Virtualisierung von Hardware. Hierdurch können mehrere virtuelle Maschinen (VM) auf einem Server deployt werden, auf denen jeweils eine Applikation läuft. Diese Virtualisierung sorgt dafür, dass die Ressourcen der VMs getrennt voneinander verwaltet werden konnten. Ein Nachteil dieses Ansatzes ist allerdings, dass jede VM ihr eigenes Betriebssystem benötigt, wodurch der Ressourcenbedarf größer ist. Außerdem liegen die Startzeiten von VMs oftmals im Minutenbreich, was vergleichsweise langsam ist. Um auch diese Probleme zu lösen, wurden Container entwickelt. Sie ähneln VMs, allerdings benötigen sie kein eigenes Betriebssystem mehr, was sich sowohl auf die Größe der Container-Images als auch deren Startzeit positiv auswirkt [2, 3].

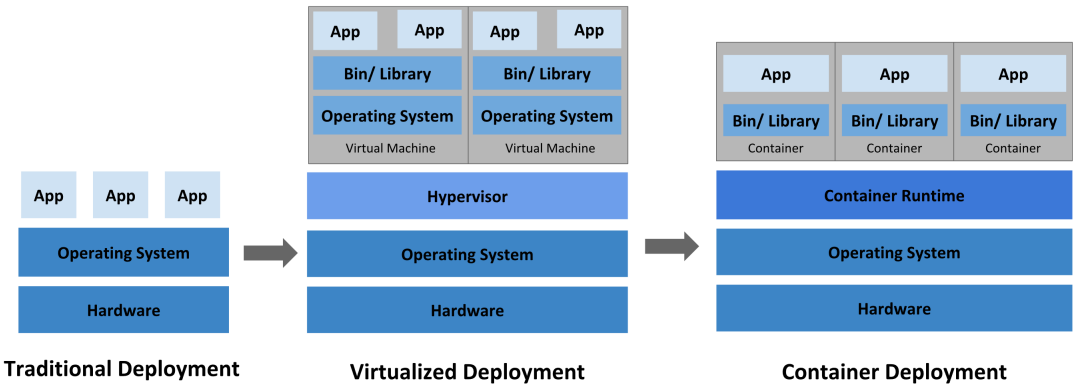

Abbildung 1.1: Entwicklung hin zu Containern als Deployment-Grundlage [3]

1.2.3 Microservices

Microservices brechen mit dem Architekturmuster eines Monolithen, indem sie die Funktionen der Applikation in kleinere Services unterteilen. Dank der im vorherigen Unterabschnitt 1.2.2 aufgeführten Entwicklungen, können diese Services als Container ressourcenschonend auf einem Server deployt werden. Dieser Ansatz versucht die Probleme im Unterabschnitt 1.2.1 zu lösen. Durch die Aufteilung in Services sind Teamverantwortlichkeiten und -grenzen klarer, die Skalierung lässt sich effizienter gestalten, da Services auch einzeln und nicht ausschließlich im Verbund skaliert werden können, Services können deployt werden, ohne sich mit anderen Teams koordinieren zu müssen und der Technologie-Stack eines jeden Services kann individuell gewählt werden. Aber auch Microservice-Architekturen haben Herausforderungen. Insbesondere führt die Unterteilung des Systems in Services zu einer vermehrten Netzwerkkommunikation zwischen den Services, die einerseits langsamer ist, aber vor allem unzuverlässiger sein kann. Die Probleme die in so einem verteilten System allein durch die Netzwerkkommunikation auftreten können sind vielfältig und die initialen intuitiven Annahmen die man trifft oftmals falsch [4]. So können andere Services beispielsweise sehr verzögert oder gar nicht antworten, was das Zusammenspiel der Services erschwert und die Komplexität des Systems erhöht. Oder wie Sam Newman es in seinem Buch 'Monolith to Microservices' [1] ausdrückt:

> "Honestly, microservices seem like a terrible idea, except for all the good stuff."

Weitere Probleme werden in einer Studie beschrieben, die Experten bezüglich ihrer Erfahrung mit Mircoservice-Architekturen befragt hat. Sie berichten von Problemen mit Multi-Repositories und komplizierter Fehlersuche [5]. Die zunehmende Verbreitung von Microservice-Architekturen hat, aufgrund der genannten Probleme, zu einer steigenden Nachfrage nach Lösungen zur Verwaltung der Kommunikation zwischen den Services geführt.

1.3 Service-Meshes

Im Microservice-Kontext sind Service-Meshes als vielversprechende Technologie aufgetaucht, die eine Reihe von Funktionen und Diensten zur Verfügung stellen, um die zuvor genannten Herausforderungen zu bewältigen. Ein Service-Mesh besteht typischerweise aus sogenannten Sidecar-Proxies, die neben jedem Service deployt werden und den Datenverkehr zwischen Services abfangen und verwalten - auch Data Plane genannt - und der Control Plane, die für die Verwaltung der Proxies verantwortlich ist.

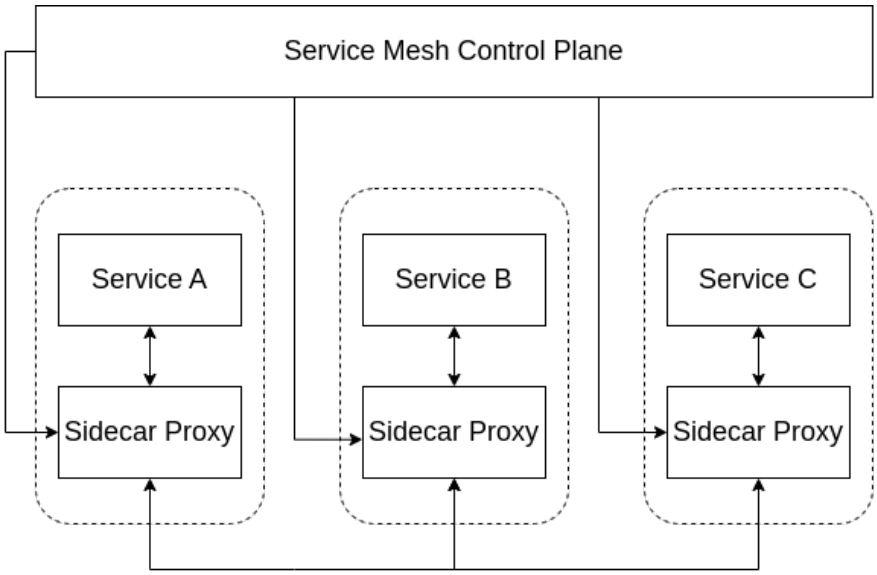

Abbildung 1.2: Service-Mesh-Architektur

Durch die Verwendung eines Service Meshes kann die Komplexität der Netzwerkkommunikation in einer Microservices-Architektur reduziert werden. Es ermöglicht die Trennung nichtfunktionaler Anforderungen wie Observability, Fehlertoleranz und Sicherheit von der Anwendungslogik, indem es Funktionen wie Monitoring, Loadbalancing, Routing und Verschlüsselung zentralisiert und standardisiert.
Zu den Kernaufgaben eines Service-Meshes gehören:

1. Entkopplung der Kommunikationslogik:
   Traditionell wurde die Kommunikationslogik, wie Service Discovery, Loadbalancing und Retry-Mechanismen, innerhalb der Services implementiert. Mit Service-Meshes wird diese Logik von den einzelnen Diensten abstrahiert und von der Mesh-Infrastruktur verwaltet. Das ermöglicht es Entwicklern, sich auf die Geschäftslogik zu konzentrieren, ohne sich um die Details der Kommunikation kümmern zu müssen.

2. Monitoring:
   Service-Meshes bieten Überwachungs- und Monitoring-Funktionen. Sie können Metriken und Traces von Diensten erfassen, um bessere Einblicke in das Verhalten des Systems zu erhalten. Das hilft bei der Fehlersuche, der Leistungsoptimierung und dem Verständnis des allgemeinen Zustands des Systems.

3. Sicherheit und Compliance:
   Service-Meshes können die Sicherheit eines verteilten Systems verbessern. Sie können Funktionen wie Verschlüsselung, Authentifizierung und Autorisierung auf der Kommunikationsebene bereitstellen, um eine sichere Kommunikation zwischen Diensten zu gewährleisten.

4. Traffic-Management und Kontrolle:
   Service-Meshes bieten erweiterte Traffic-Management-Funktionen. Sie können die Weiterleitung von Anfragen steuern, Circuitbreaker implementieren und A/B-Tests oder Canary-Deployments ermöglichen. Diese Funktionen erlauben eine feinere Kontrolle darüber, wie die Daten in einer Applikation fließen, verbessern die Zuverlässigkeit und ermöglichen schrittweise Service-Updates oder Rollbacks.

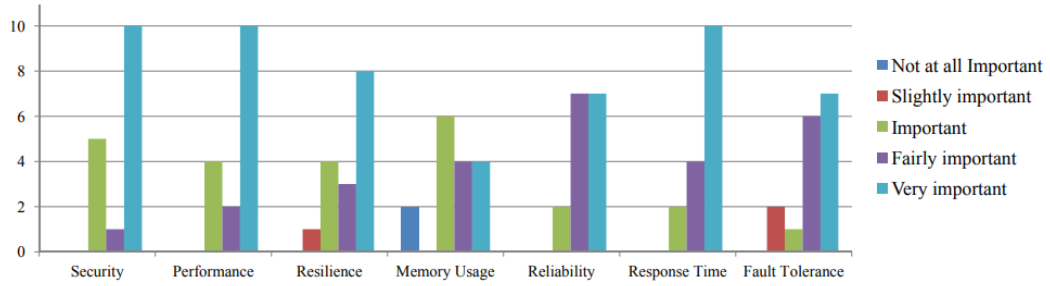

Abbildung 1.3: Herausforderungen und Bedenken im Entwicklungsprozess von Microservice-Architekturen [5]

Wie in der Abbildung 1.3 dargestellt, gehören Sicherheit und Fehlertoleranz zu den zentralen Herausforderungen bei der Entwicklung einer Microservice-Architektur. Diese beiden Anforderungen können mithilfe von Service-Meshes effektiv angegangen werden. Es ist jedoch auch erkennbar, dass die Antwortzeiten eines Systems sowie der Arbeitsspeicherverbrauch, wenn auch in unterschiedlichem Maße, wichtige Aspekte darstellen. Diese Erkenntnis wirft die Frage auf, ob Service Meshes die Herausforderungen in Bezug auf Sicherheit und Fehlertoleranz bewältigen können, ohne dabei zu große negative Auswirkungen auf die Latenz und den Ressourcenverbrauch zu haben.

1.4 Forschungsfragen

Wie zum Ende des vorherigen Abschnitts angedeutet, soll diese Arbeit einen Beitrag dazu leisten, Service Meshes bezüglich ihrer Performance besser vergleichen zu können. Dazu sollen insbesondere zwei Forschungsfragen geklärt werden.

1. Welchen Einfluss hat ein Service-Mesh auf die Latenz einer Applikation?
2. Welchen Einfluss hat ein Service-Mesh auf den Ressourcenverbrauch einer Applikation?

1.5 Abgrenzung der Arbeit

In dieser Arbeit werden mit Istio und Linkerd zwei der bekanntesten Vertreter von Service Meshes verglichen. Allerdings ist die Liste von Service-Meshes [6] lang und dies somit kein allumfassender Vergleich. Außerdem wird lediglich eine kleinere Teilmenge der Feature-Sets aktiviert. Ein anderes Feature-Set kann Einfluss auf die Messwerte nehmen. Weiter werden die Service-Meshes in Verbindung mit einer kleinen Test-Applikation eingesetzt. Eine andere Topologie und Traffic-Patterns können ebenfalls zu anderen Ergebnissen führen.

1.6 Überblick über die Struktur der Arbeit

Die Arbeit ist in sechs Kapitel eingeteilt. Im ersten Kaptiel werden Hintergrund und Motivation dieser Arbeit erläutert. Außerdem werden die Forschungsfragen, die diese Arbeits beantworten soll aufgelistet. Das zweite Kapitel behandelt die Grundlagen, die für das Verständnis von Microservice-Architekturen und Service-Meshes nötig sind. Dabei wird auch auf die beiden Service-Meshes eingegangen, die im Experiment getestet wurden, um deren Unterschiede zu diskutieren. Verwandte Arbeiten werden im dritten Kapitel behandelt. Im vierten Kapitel wird die Methodik für das Experiment genauer erläutert. Kapitel fünf stellt die Ergebnisse vor und Kapitel sechs fasst die Ergebnisse zusammen und macht Vorschläge für weitere Forschungsmöglichkeiten.

2 Grundlagen

In diesem Kapitel wird genauer auf die Technologien eingegangen, die die Grundlage für den Versuchsaufbau bilden.

2.1 Kubernetes

Kubernetes ist eine Open-Source-Plattform zur Verwaltung von Container-Workloads und deren Deployment. Es wurde von Google entwickelt und ist heute eines der am weitesten verbreiteten Orchestrierungstools für Container. Die Hauptaufgabe von Kubernetes besteht darin, auf Containern basierende Applikationen in einer skalierbaren Umgebung zu deployen. Die Konfiguration von Applikationen in Kubernetes erfolgt über sogenannte Deployments. Kubernetes verfügt über viele Tools und Erweiterungen, die die Funktionalität erweitern und an spezifische Anforderungen anpassen. Beispiele hierfür sind Helm, eine Paketverwaltung für Kubernetes-Anwendungen, oder Prometheus, ein Monitoring- und Alerting-System für Kubernetes. Insgesamt bietet Kubernetes eine robuste und skalierbare Plattform zur Verwaltung von Container-Workloads. Es automatisiert viele der komplexen Aufgaben im Zusammenhang mit der Container-Orchestrierung und ermöglicht Anwendungen effizient und zu skalieren. Darüber hinaus gibt es eine Reihe weiterer wichtiger Konzepte und Komponenten in Kubernetes, die es zu erwähnen gilt:

2.1.1 Node

Nodes sind die einzelnen Rechner, die Teil eines Kubernetes-Clusters sind. Ein Kubernetes-Cluster besteht aus mindestens einer Master-Node und einer oder mehreren Worker-Nodes. Die Master-Node ist verantwortlich für die Überwachung, Steuerung und Koordination des Clusters. Sie nimmt Befehle entgegen, plant und verteilt die Ausführung von Containern auf den Worker-Nodes und überwacht den Zustand des Clusters. Die Worker-Nodes sind die eigentlichen Ausführungsumgebungen für die Container. Auf einer Worker-Node wird eine Container-Runtime ausgeführt, die die Container startet und verwaltet (siehe Container Deployment in Abbildung 1.1). Kubernetes verteilt automatisch die Container auf den verfügbaren

Nodes basierend auf den Ressourcenanforderungen und -verfügbarkeit. Insgesamt sind die Nodes die grundlegenden Einheiten eines Kubernetes-Clusters. Sie stellen die Ressourcen für die Ausführung von Containern und ermöglichen durch die Skalierung der Anzahl der Nodes kann die Kapazität des Clusters erhöht und die Ausführung von Anwendungen und Services entsprechend skaliert werden.

2.1.2 Pod

Ein Pod ist die kleinste Einheit in Kubernetes und besteht aus einem oder mehreren Containern, die eng miteinander verbunden sind und auf derselben Node ausgeführt werden. Ein Pod teilt sich Ressourcen wie Netzwerk und Speicher, und ermöglicht die Kommunikation zwischen den Containern innerhalb des Pods.

2.1.3 Labels und Selektoren

Labels sind Schlüssel-Wert-Paare, die den Ressourcen in Kubernetes zugewiesen werden können. Sie dienen zur Gruppierung und Identifizierung von Ressourcen. Sie können verwendet werden um beispielsweise Pods immer auf bestimmten Nodes zu deployen.

2.1.4 Namespace

Namespaces sind virtuelle Cluster innerhalb eines Kubernetes-Clusters. Sie ermöglichen die logische Trennung von Ressourcen. Durch die Verwendung von Namespaces können Anwendungen und Teams voneinander isoliert werden, was die Verwaltung und Überwachung vereinfacht.

2.1.5 Ressourcenbegrenzung

Kubernetes ermöglicht die Definition von Ressourcengrenzen für Pods und Container. Dadurch kann die Ressourcenauslastung gesteuert und vor unkontrollierter Ressourcenverwendung geschützt werden. So können beispielsweise Begrenzungen für CPU, Arbeitsspeicher festgelegt werden.

2.1.6 ConfigMaps und Secrets

ConfigMaps ermöglichen die Trennung von Konfigurationsdaten von den Applikation. Sie können verwendet werden, um Konfigurationsparameter wie Umgebungsvariablen oder Konfi-

gurationsdateien an Pods zu übergeben. Secrets werden verwendet, um sensible Daten wie Passwörter oder Zugangsdaten sicher zu speichern und an Pods weiterzugeben.

2.1.7 Horizontal Pod Autoscaler

Der Horizontal Pod Autoscaler (HPA) ermöglicht die automatische Skalierung von Pods basierend auf der aktuellen Auslastung eines Clusters. Der HPA beobachtet Metriken wie CPU-Auslastung oder Anzahl der eingehenden Anfragen und passt die Anzahl der laufenden Pods dynamisch an. Wenn die Ressourcennutzung hoch ist, skaliert der HPA automatisch die Anzahl der Pods nach oben, um die Last zu bewältigen. Wenn die Ressourcennutzung sinkt, kann der HPA die Anzahl der Pods reduzieren, um Ressourcen freizugeben. Durch die Verwendung von HPAs können Kubernetes-Cluster effizienter genutzt werden, indem sie automatisch auf veränderte Lastbedingungen reagieren. Dies ermöglicht eine bessere Skalierbarkeit und optimale Ressourcennutzung.

2.1.8 Custom Resource Definition

Custom Resource Definitions (CRD) ermöglichen es benutzerdefinierte Ressourcentypen zu erstellen und der Kubernetes-API hinzuzufügen. Standardmäßig bietet Kubernetes eine Reihe von vordefinierten Ressourcentypen wie beispielsweise Pods und Deployments. Diese Ressourcentypen werden von Kubernetes selbst verwaltet. Mit CRDs können neue Ressourcentypen definiert werden, die explizit für eine bestimmte Applikation sind und zusätzliche Informationen und Logik enthalten. Sie können von Operatoren verwendet werden, um spezifische Aktionen auf diese Ressourcen auszuführen.

2.1.9 Operator

Ein Operator ist ein Konzept zur Verwaltung komplexer Anwendungen auf Kubernetes. Er hat spezifisches Wissen über die zu verwaltende Applikation und erleichtert dadurch deren Deployment, Skalierung und Verwaltung. Der Operator-Controller überwacht die benutzerdefinierte Ressource und reagiert auf Änderungen oder Ereignisse, um die gewünschten Zustände der Anwendung zu erreichen. Der Controller enthält die Logik und das Wissen über das Verhalten der Applikation und kann entsprechende Aktionen durchführen, um die Anwendung zu deployen, zu konfigurieren, zu skalieren, zu aktualisieren oder andere verwaltungsbezogene Aufgaben auszuführen. Ein Kubernetes Operator ermöglicht es, komplexe Anwendungen oder Services auf einer höheren Abstraktionsebene zu modellieren und zu verwalten. Statt manuell Applikationen zu konfigurieren und zu deployen, kann mithilfe von Operatoren die

Logik und das Verhalten der Anwendung in den Operator integriert und somit die Verwaltung automatisiert werden. Operatoren können beispielsweise in den Bereichen Datenbanken, Monitoring, Messaging-Systeme oder anderen komplexen Anwendungen eingesetzt werden. Sie ermöglichen eine verbesserte Automatisierung, Skalierung, Aktualisierung und Wartung von Anwendungen auf Kubernetes-Clustern und fördern die Wiederverwendbarkeit von bewährten Konfigurationen und bewährtem Wissen. Um die Verwaltung von Kubernetes-Clustern zu erleichtern, gibt es verschiedene Verwaltungswerkzeuge und Plattformen, die auf Kubernetes aufbauen. Ein Beispiel dafür ist "Kubernetes as a Service", das von verschiedenen Cloud-Anbietern angeboten wird und Deployment und Verwaltung von Kubernetes-Clustern vereinfacht. Im Kontext dieser Arbeit wird beispielsweise der Elastic Kubernetes Service von AWS verwendet, um eine Microservice-Anwendung auf einem Kubernetes-Cluster zu deployen.

2.2 Prometheus

Prometheus ist ein Open-Source-System für das Monitoring von IT-Systemen. Es wurde ursprünglich von SoundCloud entwickelt und ist inzwischen ein eigenständiges Projekt unter der Leitung der Cloud Native Computing Foundation (CNCF). Prometheus ermöglicht es Metriken einer Applikation zu sammeln und auszuwerten. Das System basiert auf einem Pull-Modell, bei dem es regelmäßig Metriken von den zu überwachenden Services abruft. Diese Metriken können verschiedene Aspekte des Systems wie CPU-Auslastung, Speicherbelegung, Netzwerkdaten und benutzerdefinierte Anwendungsdaten umfassen. Prometheus bietet eine flexible Abfragesprache namens Prometheus Query Language (PromQL), mit der komplexe Abfragen und Aggregationen auf den gesammelten Metriken möglich sind. Konkret wurde Prometheus in dieser Arbeit für die Auswertung des Ressourcenverbrauchs genutzt. Dafür wurden nach jedem Testdurchlauf die durch Prometheus gespeicherten Daten über die CPU- und Arbeitsspeichernutzung heruntergeladen und ausgewertet.

2.3 Jaeger

Jaeger ist ein Tracing-System, das von Uber entwickelt wurde und ist nun ein eigenständiges Projekt unter der Leitung der CNCF. Das Hauptziel von Jaeger besteht darin, Einblicke in die Kommunikationspfade zwischen verschiedenen Komponenten eines verteilten Systems zu gewinnen und Bottlenecks sowie Latenzprobleme zu identifizieren. Tracing ist eine Technik, die verwendet wird, um den Weg eines Requests durch ein System zu verfolgen und Informationen

über die Latenzzeit zu sammeln. Mit Jaeger können Traces von Requests verfolgt und visualisiert werden, um Probleme zu analysieren und die Leistung eines Systems zu verbessern.

Jaeger integriert sich nahtlos mit gängigen Frameworks und Bibliotheken für die Entwicklung von Microservices und unterstützt verschiedene Protokolle und Formate wie HTTP, gRPC und Pub-Sub-Systeme wie Kafka. Es ermöglicht die Instrumentierung von Anwendungen durch das Hinzufügen von Tracing-Code, der Informationen über den Beginn und das Ende einer Anfrage sammelt und an den Jaeger-Collector sendet.

Der Jaeger-Collector empfängt die Traces von den instrumentierten Anwendungen, speichert sie und bereitet sie für die Analyse auf. Anschließend können die gesammelten Traces über die Jaeger-UI visualisiert und untersucht werden. Die UI bietet Funktionen wie Trace-Visualisierung, Fehleranalyse, Abfrage von Tracedaten und die Möglichkeit, Abhängigkeitsgraphen zwischen den Komponenten des Systems zu erstellen.

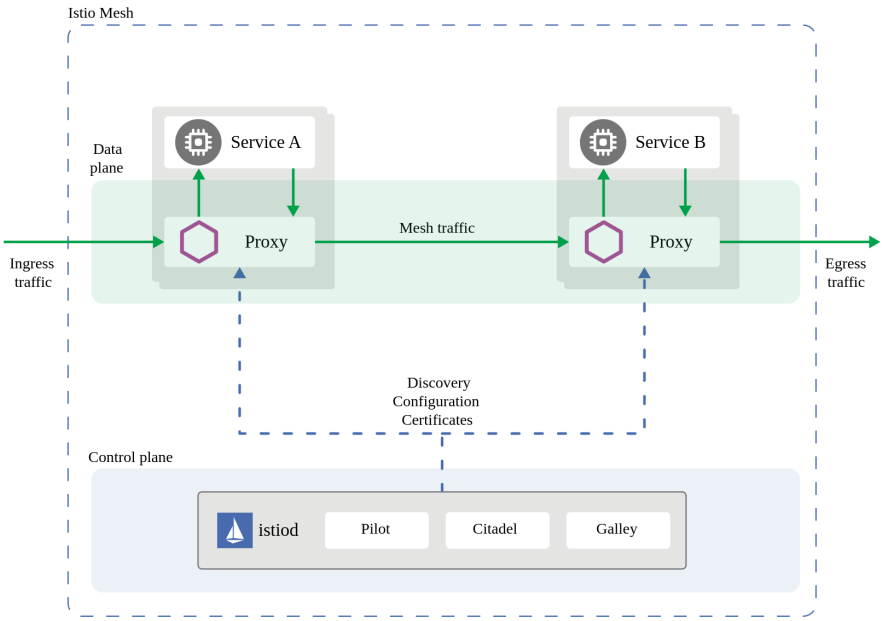

Abbildung 2.1: Isito Architektur [7]

2.4 Istio

Ursprünglich von Google, IBM und Lyft entwickelt, wird Istio jetzt von der CNCF unterstützt. Es bietet eine umfangreiche Funktionalität und verwendet Envoy als Sidecar-Proxy für die Kommunikation zwischen den Services [7]. Ein Vergleich der Abbildungen 1.2 und 2.1 zeigt die Parallelen von Istios Architektur zu der allgemeinen eines Service Meshes. Die Control-Plane ist weiter aufgeschlüsselt in Komponenten, die für die Verwaltung des Meshes nötig sind. So ermöglicht Pilot beispielsweise die Service-Discovery der Sidecar-Proxies, Citadel stellt die Authentifizierung zwischen Services sicher und Galley dient als Abstraktionsschicht zwischen die Istio und der darunterliegenden Plattform [8]. Bis Version 1.4 mussten diese Komponenten noch einzeln deployt werden. Mit Version 1.5 wurde diese Komplexität abstrahiert und alle Komponenten wurden in einer Komponente namens 'Istiod' gebündelt, was den Deployment- und Upgradeprozess erleichtern soll [9].

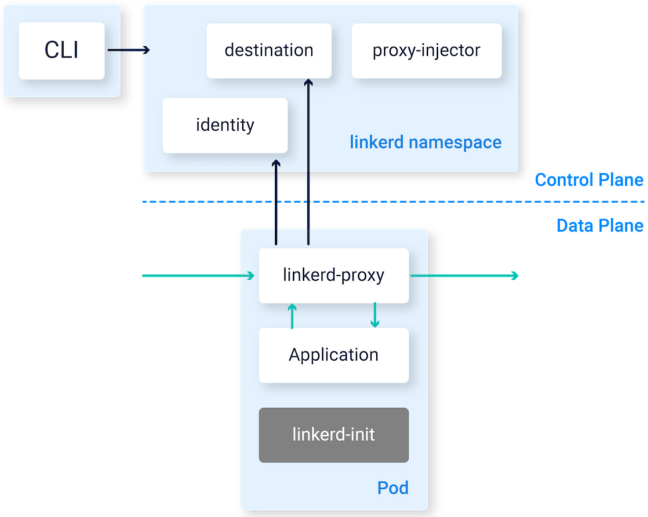

Abbildung 2.2: Linkerd Architektur [10]

2.5 Linkerd

Linkerd wurde von Buoyant entwickelt und ist ebenfalls ein Open-Source-Projekt unter der Leitung der CNCF. Es verwendet den eigens für dieses Service Mesh entwickelten Linkerd2-Proxy, der speziell auf Effizienz und geringen Overhead getrimmt wurde. Auch in dieser Architektur-Übersicht 2.2 lassen sich schnell Parallelen zur Abbildung 1.2 erkennen. Die Control-Plane besteht ähnlich wie bei Isito aus drei Komponenten. Der Destination-Service wird beispielsweise durch die Proxies für Service-Discovery-Informationen genutzt, der Identity-Service stellt die Zertifikate für die Kommunikation zwischen den Services aus und die Proxy-Injector deployt die Sidecar-Proxies sobald ein Pod in Kubernetes erstellt wird [10]. Buoyant hat im Gegensatz zu fast allen anderen Anbietern von Service Meshes nicht auf Envoy als Sidecar-Proxy gesetzt [6]. Was die Frage aufwirft, inwiefern dieser Proxy Einfluss auf Latenz und Ressourcenverbrauch hat.

2.6 Envoy vs Linkerd2-Proxy

In einem Blogpost auf Buoyants Website [11] werden drei größere Punkte diskutiert. Zuerst die Komplexität, die ein Proxy wie Envoy mit sich bringt. Im Gegensatz zum Linkerd2-Proxy, der entwickelt wurde um ausschließlich als Sidecar-Proxy für die Kommunikation zwischen Services zu dienen, ist der Envoy-Proxy vielseitiger einsetzbar. Er kann beispielsweise auch als Ingress- oder Egress-Gateway genutzt werden [12, 13]. Oder wie es abschließend im Blogpost zusammengefasst wird:

> 'Envoy is a Swiss Army knife. Linkerd2-proxy is a needle.'

Ein weiterer Punkt ist die Sicherheit der Proxies. Der Artikel führt einige Quellen an, die zeigen sollen, dass ein Großteil der Sicherheitslücken des Envoy-Proxys auf das Memorymodel der ihm zugrunde liegenden Programmiersparche C++ zurückzuführen sind. Der Linkerd2-Proxy ist in Rust geschrieben und vermeidet durch die Art und Weise wie der Speicher verwaltet wird [14] diese Art von Sicherheitslücken komplett, da sie bereits beim Kompilieren ausgeschlossen werden können [15]. Außerdem wird auch der Ressourcenverbrauch explizit angesprochen. Druch die Spezialisierung auf genau einen Anwendungsfall, konnte der Ressourcenverbrach laut Blogpost sowohl in Bezug auf CPU-Zeit als auch Arbeitsspeichernutzung minimiert werden. Dazu wird auf einen internen Benchmark verwiesen, der im nachfolgenden Kapitel 3 genauer untersucht wird.

3 Verwandte Arbeiten

Da Service Meshes noch eine vergleichsweise junge Technologie ist, gibt es nicht viel Forschung, die sich mit der Performance dieser befasst. Dennoch gibt es einige Online-Ressourcen, die sich mit Performance-Vergleichen verschiedener Service-Meshes beschäftigen. Im folgenden Kapitel werden zwei Experimente genauer betrachtet und ihre Testmethoden analysiert, um eine Grundlage für die eigene Methodik zu schaffen.

3.1 Eric Dahlberg

Eric Dahlberg hat sich 2020 in seiner Masterarbeit mit dem Performancevergleich von Isio und Linkerd auseinandergesetzt. Die Arbeit selbst ist nicht einsehbar, allerdings hat die Firma, die die Arbeit betreute, einen Blog-Artikel [16] verfasst, der die wichtigsten Ergebnisse der Arbeit zusammenfasst. Das Ziel der Arbeit war ein Vergleich der beiden Service-Meshes hinsichtlich ihrer Features, Reife, Nutzerfreundlichkeit und Performance. Da sich diese Bachelorarbeit ausschließlich mit der Performance befasst, wird auch nur auf diesen Teil des Blog-Artikels eingegangen. Um die Performance zu vergleichen wurde ein Cluster mittels der Google Kubernetes Engine mit insgesamt 5 Nodes aufgesetzt, die jeweils über 4 vCPUs und 15GB RAM verfügten. Isito (1.5.0) und Linkerd (2.7.1) wurden in der Konfiguration gestartet, die die 'Get-Started'-Website des Meshes empfiehlt. Die Latenz wurde mittels Apache JMeter gemessen und der Ressourcenverbrauch wurde mit hilfe von Prometheus ermittelt. Als zugrundeliegende Applikation wurde TeaStore [17] gewählt. Um das System unter Last zu setzen, wurden Nutzer simuliert, die jeweils mit fünf RPS das System anfragen. Die Anzahl der Nutzer wurde im Laufe des Experiments stufenweise erhöht (5, 10, 20, 30 , 40). Um zu verhindern, dass Caching die Messergebnisse verfälscht, wurde vor Beginn der Messungen das System für drei Minuten mit 20 Nutzern 'aufgewärmt'. Anschließend wurde für jeweils 14 Minuten pro Stufen das System belastet. Zwischen der Stufen wurde eine Wartezeit von fünf Minuten berücksichtigt, um dem System die Möglichkeit zu geben sich von der vorherigen Belastung zu erholen.

	CPU	RAM	Latenz(75. Perzentil)
no-mesh	2226 mCPU	9749Mb	18ms
linkerd	4386 mCPU	10947Mb	49ms
istio	5132 mCPU	14152Mb	42ms

Tabelle 3.1: Dahlbergs Ergebnisse bei 20 Nutzern [16]

Auf genaue Zahlen bezüglich Latenz und Ressourcen wird im Blog-Artikel nicht detailliert eingegangen. Es werden lediglich die Latenzen im 75. Perzentil verglichen, die Istio eine Erhöhung um 133% und Linkerd eine um 175% zuschreiben. Die CPU-Ressourcen wurden durch Istio um zusätzliche 131% und durch Linkerd um 97% belastet. Der Arbeitsspeicher des Systems wurden 45% mehr durch Istio und 11% mehr durch Linkerd genutzt. Als Ergebnis wird festgehalten, dass Istio zwar eine geringere Latenz erzielen kann, dies allerdings nur durch eine erhöht CPU- und Arbeitsspeicher-Nutzung erreicht. Somit komme Istio für Applikationen in Frage, die mehr Wert auf Antwortzeiten legen, wohingegen Linkerd sich besser für Systeme mit weniger Ressourcen eigne.
Der Blog-Artikel lässt einige Details aus. So ist beispielsweise nicht klar zu erkennen, welche Funktionalitäten der Service-Meshes aktiviert wurden. Außerdem wurde der Ressourcenverbrauch clusterweit gemessen. Hier wäre eine genauere Aufteilung in Data- und Control-Plane hilfreich, um festzustellen welche Komponenten der Service-Meshes den Ressourcenverbrauch verursachen.

3.2 Thilo Fromm

Thilo Fromm hat die wohl umfangreichsten Lasttests durchgeführt. Diese wurden ebenfalls in einem Blog-Artikel beschrieben. Hierbei sollte ein Szenario simuliert werden, bei dem das zugrundeliegende System zwar stark belastet wurde, die Antwortzeiten aber immer noch in einem akzeptablen Rahmen lagen. Getestet wurden dabei die Applikation ohne Service-Mesh, mit Linkerd und mit Isio als Service-Mesh. Bei der ersten Veröffentlichung des Artikels wurde dabei sowohl für Linkerd als auch Istio die Konfiguratiuon genutzt, die der jeweilige Anbieter auf seiner 'Get-Started'-Website angegeben hat. Leser haben angemerkt, dass Istios Konfiguration nicht für Performance-Tests geeignet sei, weshalb ein weiterer Testlauf mit einem angepassten Istio-Profil durchgeführt wurde. Die Details dieser Anpassungen sind für diese Arbeit nicht relevant, da in Fromms Experiment Istio in der Version 1.1.6 verwendet wurde. Wie im Abschnitt zu Istio 2.4 erläutert, wurde die Control-Plane von Istio mit Version 1.5 vereinfacht. Die Änderungen im Artikel beziehen sich auf eine Komponente, die ab Version

1.5 nicht mehr Teil der Control-Plane war. Weitere Anpassungen der Konfigurationen die sich sowohl auf Linkerd als auch Istio beziehen sind, dass auf Ingress-Gateways verzichtet und Tracing deaktiviert wurde. Es wird auch erwähnt, dass die von Istio angebotene Prometheus-Instanz nicht genutzt wurde. Allerdings geht aus dem Artikel nicht hervor, welche Alternative eingesetzt wurde. Als Grundlage für das Experiment diente ein Cluster mit fünf Worker-Nodes, die jeweils mit 48 vCPUs und 64GB RAM ausgestattet waren. Die Tests wurden pro Konfiguration auf zwei identischen Clustern jeweils zwei Mal durchgeführt. Jede Konfiguration wurde also vier mal getestet. Zu Beginn eines jeden Durchlaufs wurde zufällig eine der Nodes als Load-Generator ausgewählt. Auf den übrigen Nodes wurde die Applikation emojivoto deployt. Bei jedem Druchlauf wurde zuerst das Service-Mesh und anschließend die Applikation installiert. Daraufhin wurde der Load-Generator gestartet und über ein Zeitfenster von 30 Minuten Testdaten gesammelt. Abschließend wurden die Testdaten heruntergeladen und Service-Mesh und Applikation wieder gelöscht. Dies wurde für jede Konfigurationen (kein Service-Mesh, Linkerd, Istio und Istio 'tuned') wiederholt.

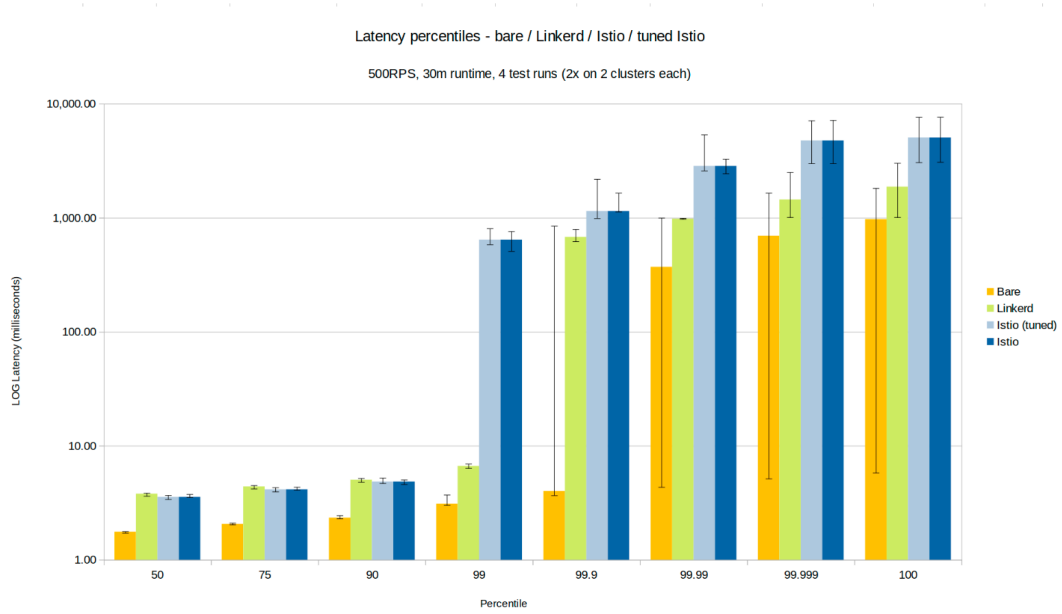

Abbildung 3.1: Fromms Ergebnisse bei 500 RPS (Latenz) [18]

Die Ergebnisse von Fromm 3.1 bezüglich der Latenz zeigen, dass Linkerd in den höheren Perzentilen niedrigere Werte erreicht als Istio. Dieses Ergebnis ist unabhängig von der Konfiguration in der Istio getestet wurde.

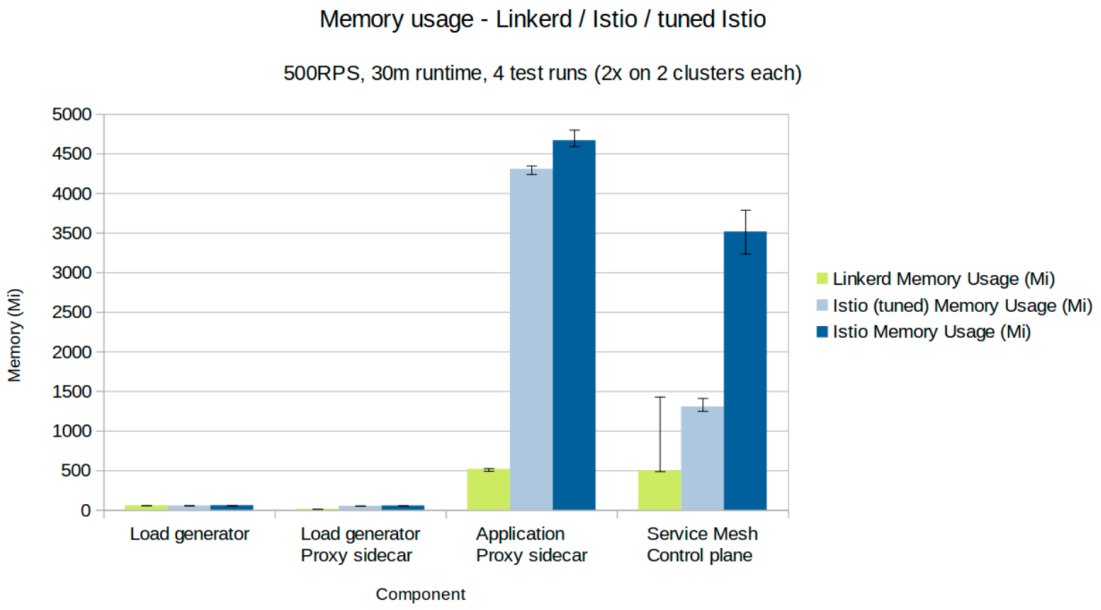

Abbildung 3.2: Fromms Ergebnisse bei 500 RPS (Arbeitsspeicher) [18]

Die CPU- und Arbeitsspeicher-Ergebnisse bilden jeweils den Median, sowie den höhsten und niedrigsten Wert ab, der in den vier Durchläufen gemessen wurde. Der Ausreißer beim Arbeitsspeicherverbrauch der Control-Plane für Linkerd ist laut Artikel auf eine Prometheus-Instanz zurückzuführen, die in einem Durchlauf ungewöhlich viel Arbeitsspeicher verbraucht hat. Der gleiche Ablauf wurde nochmal mit 600 RPS durchgeführt, wobei Istio, im Gegensatz zu Linkerd den geforderten Durchsatz nicht mehr erreichen konnte. Istios Antwortzeiten lagen dabei teils im Minutenbereich. Der Artikel weißt aber auch darauf hin, dass dies kein Szenario ist, dass tatsächlich auftritt, da das Cluster schon frühre skaliert 2.1.7 hätte. Auch dieser Artikel lässt einige Details aus. Insbesondere ist die Konfiguration der Service-Meshes nicht klar. So wird einerseits erwähnt, dass bei Istio auf die Prometheus-Instanz verzichtet wurde, aber bei Linkerd die mitgelieferte Prometeus-Instanz genutzt wurde. Was in einem Testdurchlauf zu einem erhöhten Ressourcenverbrauch geführt hat. Es ist nicht ersichtlich welche Alternative bei Istio eingesetzt wurde.

Es gibt noch eine Vielzahl [19, 20, 21, 22, 23, 24] weiterer Experimente, die Linkerd und Istio miteinander vergleichen. Allerdings sind die Artikel in vielen Fällen vergleichsweise alt, genau wie der vom Fromm, oder es ist nicht klar welche Konfiguration verwendet wurde. Eine weitere Auflistung würde also keinen signifikanten Betrag zu dieser Arbeit leisten. Dennoch

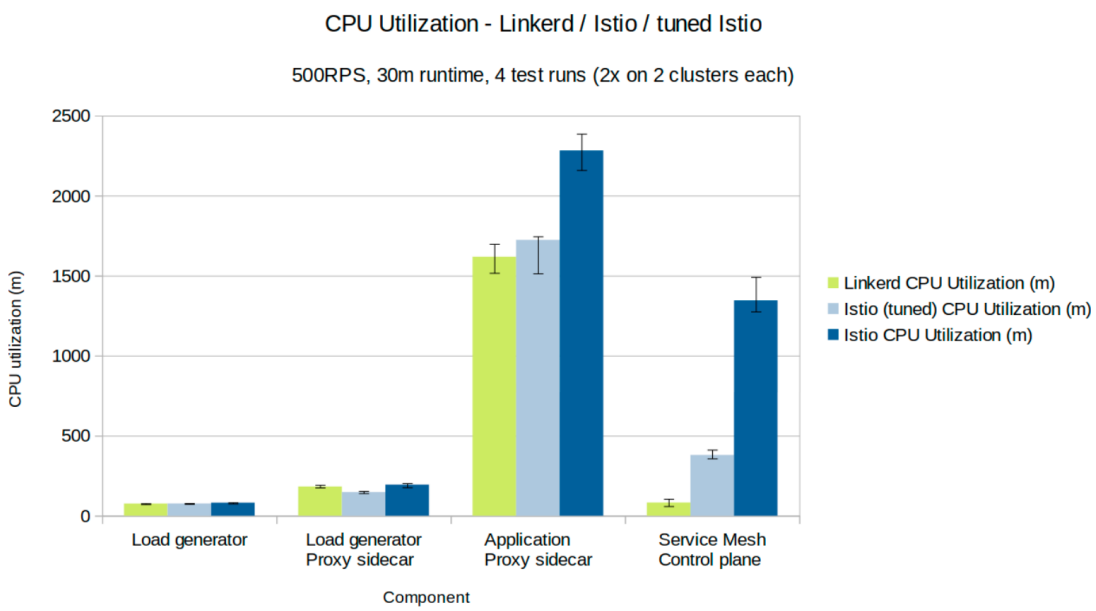

Abbildung 3.3: Fromms Ergebnisse bei 500 RPS (CPU) [18]

lassen sich aus den genannten Experimenten gute Grundlagen für den Aufbau des Experiments dieser Arbeit ableiten. Welche das sind, wird im nächsten Kapitel genauer erläutert.

4 Methodik

In diesem Kapitel wird darauf eingegangen wie der Aufbau der Testumgebung aussieht und welche Aspekte durch diesen Aufbau berücksichtigt werden sollen.

4.0.1 Test-Cluster

Die Test-Cluster beschreibt die Umgebung, in der die Test-Applikation deployt wird. Aufgrund der Vertrautheit mit den Produkten, wurde Amazon Web Services (AWS) als Cloudprovider für dieses Experiment ausgewählt. Speziell wurde Elastic Kubernetes Service (EKS) verwendet, um eine Kubernetes-Umgebung bereitzustellen, in der die Service Meshes getestet werden können. Das Cluster besteht aus drei Nodes vom Instanztyp c3.2xlarge. Die Datenbanken der Applikation werden mithilfe von RDS (Instanztyp db.r5.xlarge) realisiert. Das EKS-Cluster verwendet die Kubernetes-Version 1.26.

4.0.2 Test-Applikation

Die Test-Applikation ist ein vertikaler Schnitt einer breiteren Microservice-Architektur. Es beschreibt den Großteil einer Nutzerverwaltung und besteht aus vier Microservices, die im Folgenden kurz erläutert werden.

1. Api-Gateway
   Ist zentrale Startpunkt eines jeden Requests an das System. Von hier werden die Request an die cluster-internen Mircoservices weitergeleitet.

2. Customers
   Ist für die Verwaltung allgemeiner Kundendaten wie Name, Vorname, E-Mail-Addresse, etc. verantwortlich.

3. Fundingsources
   Ist für die Verwaltung verschiedener Bezahlmethoden der Kunden verantwortlich.

4. Tags
   Ist für die Verwaltung von Tags verantwortlich. Diese Tags bieten zusätzliche Informa-

tionen über verschiedene Entitäten in System. Im Kontext dieses Testsystems bieten die Tags Informationen über den Kunden, die spezifischen Prozessen benötigt werden.

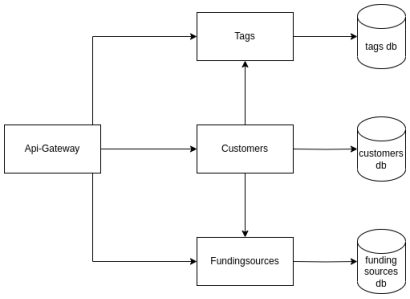

Abbildung 4.1: Architektur des Testsystems

4.1 Anforderungen an die Testumgebung

Wie abschließend in Kapitel 3 über verwandte Arbeiten erwähnt, lassen sich aus den genannten Experimenten Grundlagen ablesen, die zu einer guten Vergleichbarkeit der Service-Meshes führen. Diese Grundlagen werden im Folgenden näher erläutert.

1. Load-Generator: Die Last sollte innerhalb des Clusters erzeugt werden. Der Internetverkehr würde für zusätzliche Latenz sorgen [25] und somit ein erhebliches Rauschen in den Messdaten erzeugen.

2. Caches: Die Caches des Systems sollten 'aufgewärmt' werden. Dadurch wird sichergestellt, dass nach einem Neustart des Systems die Messdaten nicht nach oben verfälscht werden.

3. Ingress: Es sollte kein Ingress-Gateway genutzt werden. Zwar hat Istio ein Ingress-AddOn, das genau wie die Sidecar-Proxies auf dem Envoy-Proxy basiert, allerdings trifft das nicht auch auf Linkerd zu. Wie im Abschnitt zum Vergleich der Proxies 2.6 erläutert, wird der Linkerd2-Proxy ausschließlich als Sidecar-Proxy verwendet. Stattdessen bietet Linkerd die Möglichkeit eine Vielzahl von Ingress-Controllern zu integrieren [26]. Die Auswahl des Ingress-Anbieters könnte also eine Auswirkung auf die Messergebnisse haben.

4. Feature-Set: Die Feature-Sets sollten gleich sein. Ein Mehr an Features bedeutet auch immer Mehraufwand für das Service-Mesh. Da die Konfiguration der 'Get-Started'-

Website der beiden Meshes nicht gut erkennen lässt, welche Features aktiviert sind, muss einen andere Deploymentstrategie gefunden werden, um sicherzustellen, dass die gleichen Features aktiviert sind.

5. Implementierung der Feature-Sets: Auch wenn Istio und Linkerd Features wie Tracing in Form von Jaeger 2.3 oder Monitoring in Form von Prometheus 2.2 anbieten, ist es wichtig sicherzustellen, dass auch hier die Konfiguration vergleichbar ist. Die Anzahl der gesammelten Traces oder das Intervall in dem Services nach Monitoring-Informationen abgefragt werden, kann auf Latenz und Ressourcenverbrauch Einfluss nehmen. Es muss also eine Möglichkeit gefunden werden die Konfiguration der Features vergleichbar zu halten.

6. Trennung der Verantwortlichkeiten: Der Ressourcenverbrauch kann mittels Prometheus bis auf Container-Ebene gemessen werden. Um hier genaue Ergebnisse zu erhalten, ist es also nicht zwingend nötig den Load-Generator, die Applikation und andere Services auf separaten Nodes 2.1.1 zu deployen. Allerdings könnte es zu Rauschen bei den Latenz-Messungen führen, da unter Umständen die selbe CPU Threads abarbeitet, die für Load-Generierung und Bearbeitung der Requests an die Applikation zuständig ist. Es muss also sichergestellt sein, dass sowohl der Load-Generator als auch die Applikation auf getrennten CPUs laufen, um zu verhindern, dass sie sich gegenseitig die CPU-Ressourcen streitig machen.

Der Punkt bezüglich Caches bedarf keiner besonderen technischen Lösung, der Messzeitraum wird erweitert und die ersten Minuten werden nicht mit ausgewertet. Das gleiche gilt für das Ingress-Gateway. Es wird keins deployt. Im Gegensatz dazu benötigen die anderen Anforderungen technische Lösungen, die im folgenden Abschnitte erläutert werden.

4.2 Tools und Technologien

4.2.1 k6

Bei k6 handelt es sich um ein Open-Source-Tool zur Durchführung von Lasttests für Webanwendungen. Es bietet die Möglichkeit, die Leistungsfähigkeit von Anwendungen zu überprüfen, indem es simulierten Traffic erzeugt und die Reaktion des Systems auf Lastszenarien überwacht.

k6 Operator

Der k6-Operator (Unterabschnitt 2.1.9) ist speziell für die Verwendung von k6 in Kubernetes-Clustern entwickelt worden. Er ermöglicht es, k6-Tests innerhalb eines Kubernetes-Clusters zu orchestrieren und auszuführen. Er automatisiert die Bereitstellung, Skalierung und Überwachung von k6-Testinstanzen in einer Kubernetes-Umgebung. Dadurch können k6-Tests als CRDs (Unterabschnitt 2.1.8) in Kubernetes definiert werden. Diese Ressourcen beschreiben die Testkonfiguration, wie z.B. die zu testende Anwendung, die Skriptdatei, die gewünschte Last oder andere Parameter. Der Operator überwacht diese Custom-Ressourcen und erstellt, aktualisiert oder löscht k6-Testinstanzen basierend auf den spezifizierten Konfigurationen. Der k6-Operator löst somit die Anforderung, dass die Latenzen nur innerhalb des Clusters erzeugt und gemessen werden sollten.

4.2.2 Helm

Bei Helm handelt es sich um ein Open-Source-Projekt zur Verwaltung von Anwendungen in Kubernetes-Clustern. Es ermöglicht die Definition von Anwendungen und deren Abhängigkeiten in Helm-Charts. Ein Helm-Chart ist eine Sammlung von Dateien, die die notwendigen Ressourcen und Konfigurationen enthalten, um eine Anwendung in Kubernetes zu deployen [27]. Mittels Helm lassen sich die Anforderungen bezüglich Feature-Sets und deren Implementierung angehen. Die in Helm-Charts angegebenen Default-Werte können überschrieben werden. Istio und Linkerd lassen sich so genauer konfigurieren. Außerdem können so auch externe Services eingebunden werden, anstatt die mitgelieferten AddOns zu nutzen. Konkret kann eine externe Prometeus-Instanz genutzt werden, um den Ressourcenverbrauch der Service-Meshes zu messen.

4.2.3 Flux

Flux ist ein Open-Source-Tool um CI/CD-Workflows in Kubernetes-Clustern zu realisieren. Es ermöglicht Applikationen und Konfigurationen als Kubernetes-Ressourcen zu definieren und in Git-Repositories zu speichern. Anschließend synchronisiert Flux diese Definitionen mit dem Kubernetes-Cluster und stellt sicher, dass der tatsächliche Zustand des Clusters dem gewünschten Zustand entspricht. Es überwacht die Git-Repositorys auf Änderungen und synchronisiert die geänderten Ressourcen automatisch mit dem Cluster [28]. Insbesondere bietet Flux auch eine Integration mit Helm-Charts für die Verwaltung von Anwendungen [29]. Zwar wurde Flux in dieser Arbeit auch genutzt, um die Test-Applikation in im Test-Cluster zu deployen, allerdings war der primäre Treiber die Integration mit Helm-Charts. So konnten

die Service-Meshes in Flux definiert und konfiguriert werden. Dadurch wurde sichergestellt, dass auch bei wiederholten Testläufen und nach mehreren Wechseln zwischen den konkreten Implementierungen die Konfiguration der Meshes die selbe bleibt.

4.2.4 Kubernetes Taints und Affinities

Um den letzten Punkt der Anforderungen bezüglich Trennung der Verantwortlichkeiten anzugehen, bietet Kubernetes Möglichkeiten (Unterabschnitt 2.1.3) sicherzustellen, dass Pods auf bestimmten Nodes deployt werden. Durch die Anwendung von Taints und Affinities wird forciert, dass die Applikation, der Load-Generator und die sonstige Infrastruktur immer auf getrennten Nodes laufen.

4.3 Ablauf der Performance-Tests

Im folgenden wird der Ablauf der Performance-Tests beschrieben. Die App-Node beschreibt dabei die Node auf der die eigentliche Applikation läuft und die k6-Node die auf der der Load-Generator läuft. Durch manuelle Load-Tests wurde eine Obergrenze von 500 RPS ermittelt, bei der alle Konfigurationen akzeptable Antwortzeiten liefern und nicht zu Timeouts der Requests führen. Dennoch wurden verschiedene Laststufen (25, 50, 100, 250, 500) getestet, um die Service-Meshes auch bei moderater Last vergleichen zu können. Dabei sah ein Durchlauf wie folgt aus:

1. Deployment des Service-Meshes auf der App-Node (überspringen für Referenztest ohne Service-Mesh)
2. Deployment der Applikation auf der App-Node
3. Deployment des Load-Generators auf der K6-Node
4. Abwarten bis der Test beendet ist
5. Herunterladen der Test-Daten (Latenz)
6. Wiederholen von Schritt 3-5 bis alle Laststufen getestet wurden
7. Herunterladen der Test-Daten (Ressourcenverbrauch)
8. Entfernen des Service-Meshes (wenn vorhanden) und Applikation von der App-Node
9. Wiederhole Schritt 1-8 für alle Testkonfigurationen (kein Service-Mesh, Linkerd, Istio)

5 Ergebnisse

In diesem Kapitel wird auf die Ergebnisse der Experimente eingegangen. Dazu werden zuerst die Latenztests betrachtet und miteinander verglichen. Anschließend werden die Ergebnisse über den Ressourcenverbrauch betrachtet.

5.1 Latenz

In den Grafiken sind die Messergebnisse über die Zeit abgetragen, sowie die Perzentile im Vergleich zueinander. Da es insbesondere bei einer höheren Anzahl an Request pro Sekunde zu starken Ausschlägen bei der Latenzmessung gekommen ist, wurde davon abgesehen die Maximalwerte mit in die Grafik einzubinden. Sie sind aber in den entsprechenden Tabellen zu finden. Die Ergebnisse bei 25 RPS zeigen einen ersten Trend bei der Latenz. Linkerd ist bis zum 99.9 Perzentil schneller als Istio. Lediglich die Maximalwerte weichen vom Trend ab. Die Ergebnisse der anderen Laststufen zeichnen ein ähnliches Bild. Allerdings sind insbesondere bei den höheren Laststufen immer wieder Latenzspitzen zu erkennen. Bei 500 RPS weisen diese zumindest ohne Service-Mesh und bei Linkerd eine gewisse Regelmäßigkeit auf. Woher diese Spitzen kommen konnte nicht festgestellt werden. Da es sich bei der Applikation um Spring-Boot-Anwendungen handelt, lag die Vermutung nahe, dass der Garbage-Collector Einfluss auf die Messwerte nimmt. Allerdings konnte keine Korrelation zwischen den Latenzspitzen und den Momenten in denen der Garbage-Collector arbeitet hergestellt werden. Allgemein lässt sich sagen, dass die Service-Meshes einen konstanten Latenz-Overhead verursachen, der unabhängig von der Laststufe ist. Die Ergebnisse zeige ein umgekehrtes Bild zu denen von Dahlberg (siehe Tabelle 3.1), wo Linkerd höhere Latenzen verursacht hat.

Tabelle 5.1: Perzentile bei 25 Requests pro Sekunde

	min	p25	p50	p75	p95	p99	p999	max
no-mesh	6.741	7.543	8.055	8.628	9.711	14.803	27.296	48.872
linkerd	8.933	9.711	10.292	11.109	12.189	16.053	29.630	74.514
istio	9.942	10.754	11.523	12.213	13.506	16.717	31.047	44.253

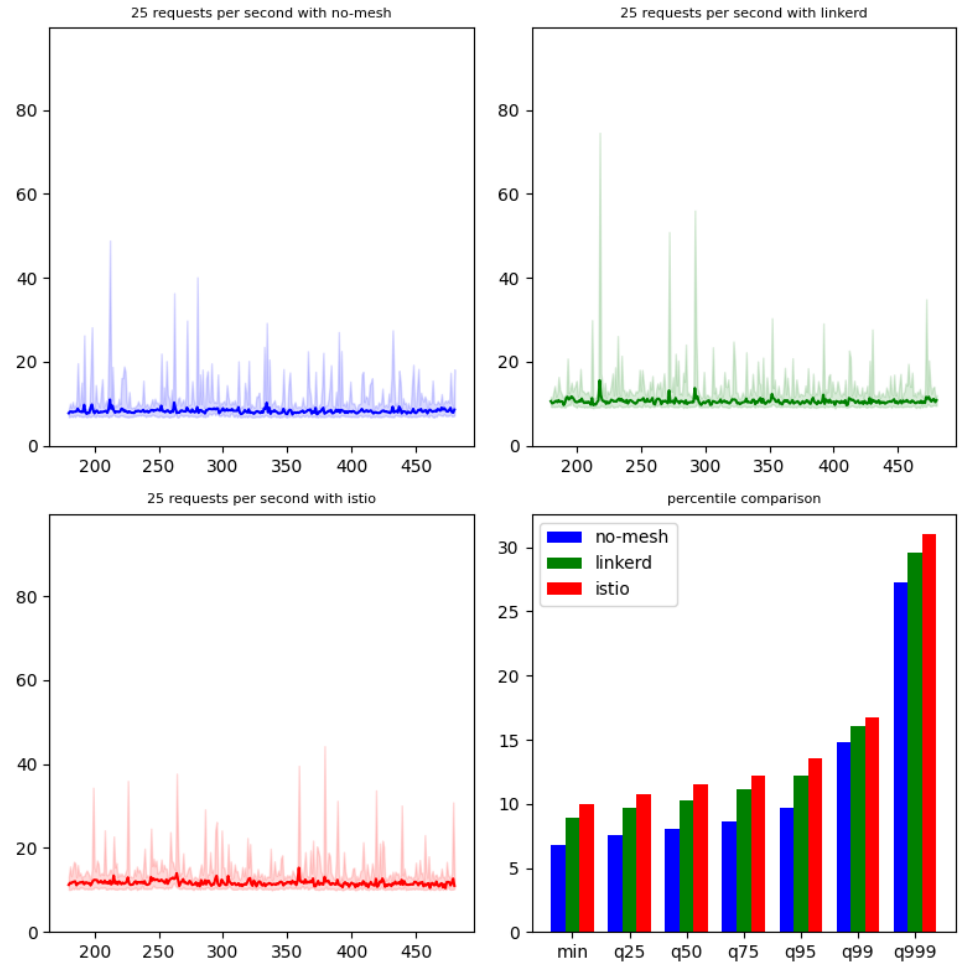

Abbildung 5.1: Vergleich bei 25 Requests pro Sekunde

Tabelle 5.2: Perzentile für 50 Requests pro Sekunde

	min	p25	p50	p75	p95	p99	p999	max
no-mesh	6.578	6.975	7.193	7.520	8.604	13.237	23.757	161.576
linkerd	8.901	9.858	10.638	11.421	12.634	17.768	32.188	138.194
istio	9.720	10.271	10.585	11.138	12.439	15.421	30.563	75.917

	min	p25	p50	p75	p95	p99	p999	max
no-mesh	6.521	6.867	7.034	7.300	8.633	13.727	25.482	136.908
linkerd	8.791	9.276	9.537	9.981	11.606	16.385	26.479	49.604
istio	9.533	10.093	10.320	10.705	12.253	15.509	25.560	38.398

Tabelle 5.3: Perzentile bei 100 Requests pro Sekunde

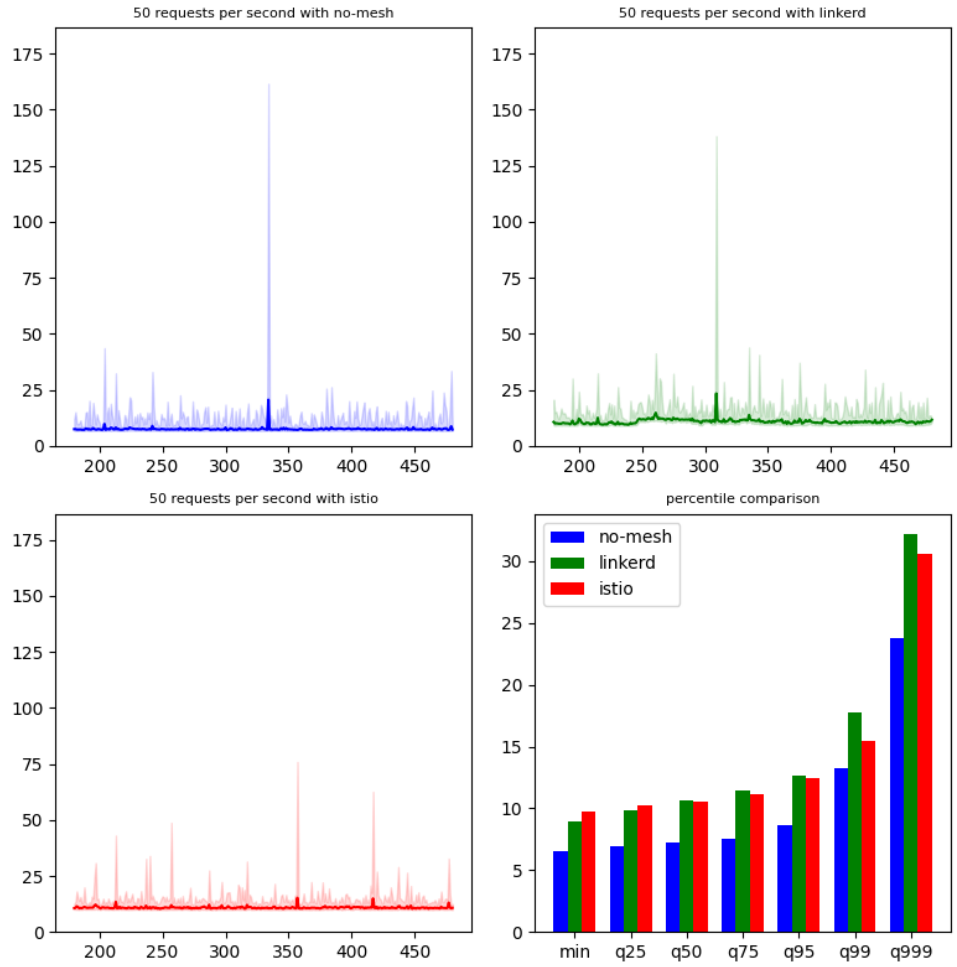

Abbildung 5.2: Vergleich bei 50 Requests pro Sekunde

Tabelle 5.4: Perzentile bei 250 Requests pro Sekunde

	min	p25	p50	p75	p95	p99	p999	max
no-mesh	6.430	6.822	6.935	7.161	9.254	15.331	40.083	150.396
linkerd	9.184	9.909	10.127	10.465	12.802	19.048	56.213	182.506
istio	10.080	11.089	11.344	11.721	13.850	18.384	30.753	68.750

Tabelle 5.5: Perzentile bei 500 Requests pro Sekunde

	min	p25	p50	p75	p95	p99	p999	max
no-mesh	6.601	7.089	7.251	7.550	10.715	18.255	120.040	210.278
linkerd	10.062	11.844	12.397	13.587	20.893	34.928	137.306	368.166
istio	11.468	14.232	15.043	16.485	21.990	36.627	168.935	392.038

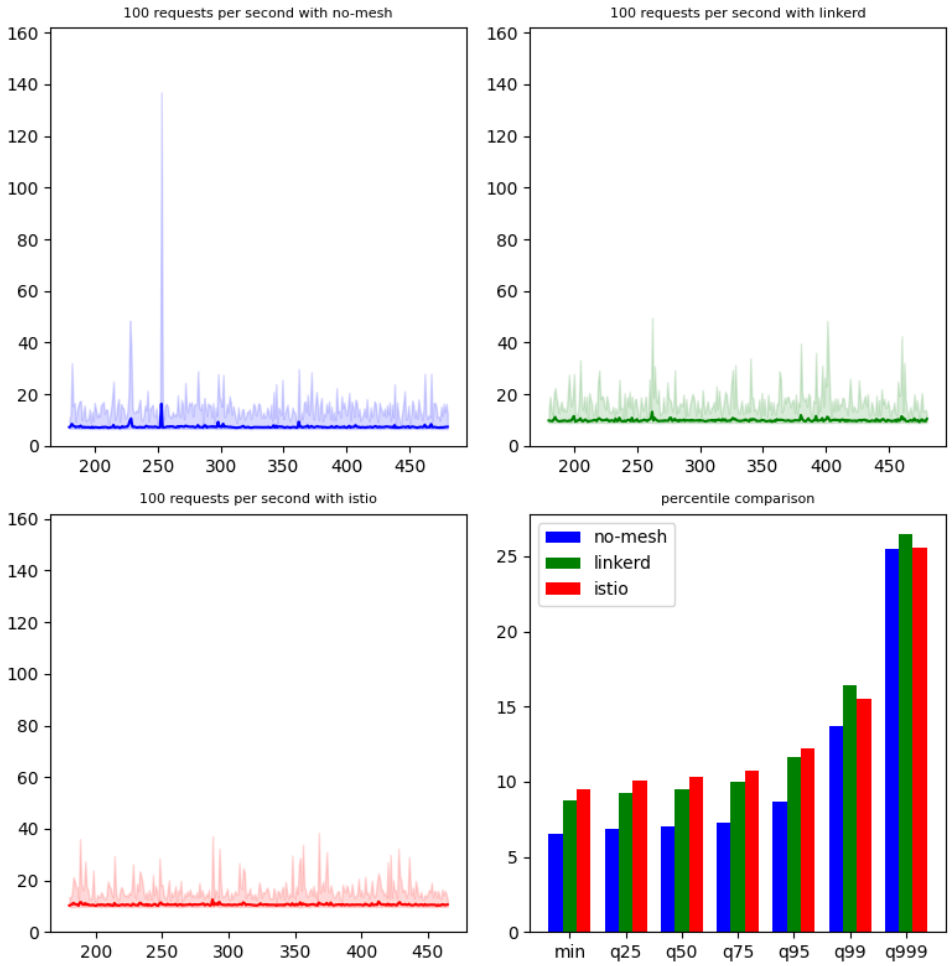

Abbildung 5.3: Vergleich bei 100 Requests pro Sekunde

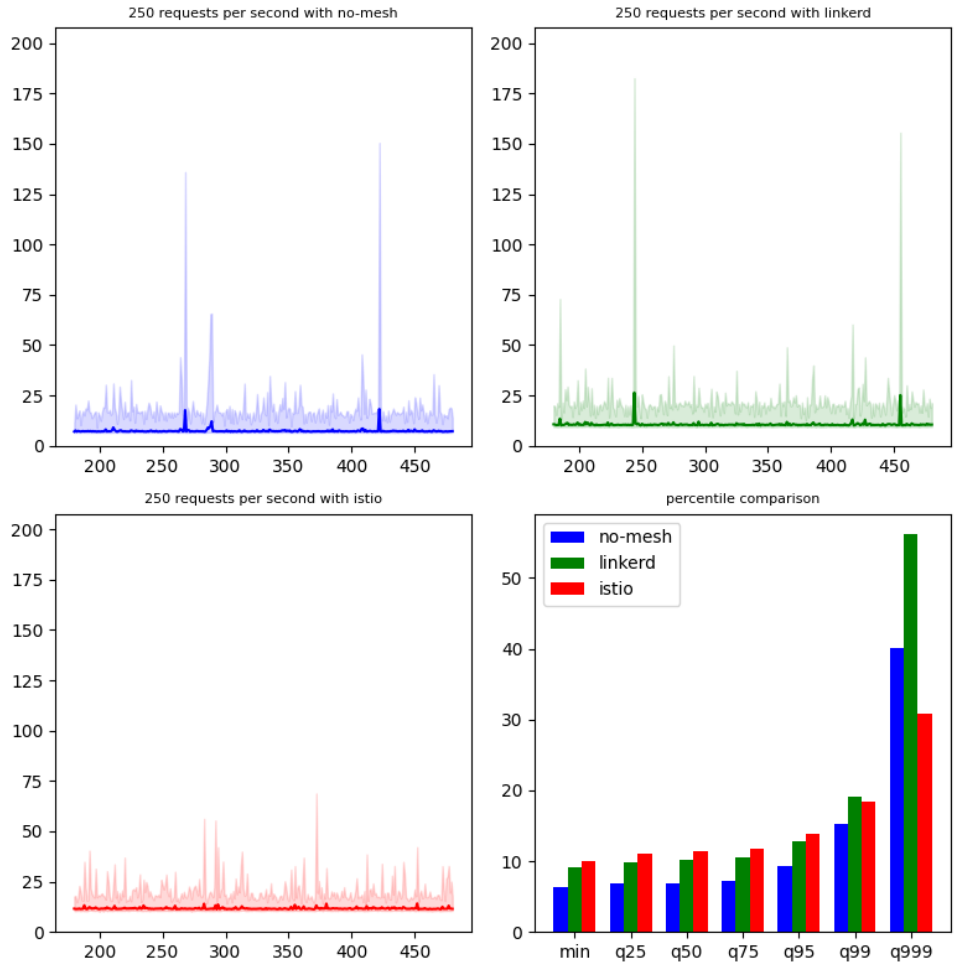

Abbildung 5.4: Vergleich bei 250 Requests pro Sekunde

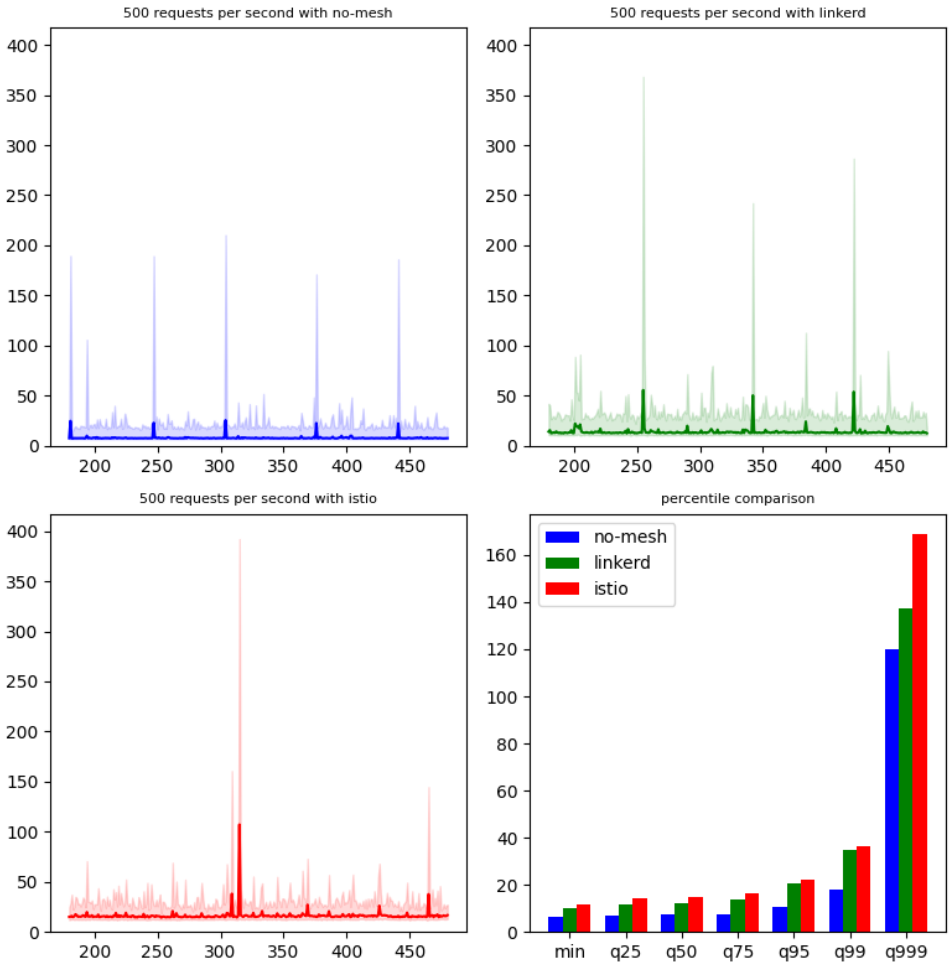

Abbildung 5.5: Vergleich bei 500 Requests pro Sekunde

Tabelle 5.6: Ressourcenverbrauch bei 25 RPS

	min	avg	median	max
cpu data plane istio	0.089	0.118	0.120	0.124
cpu data plane linkerd	0.064	0.091	0.094	0.095
cpu control plane istio	0.005	0.006	0.006	0.008
cpu control plane linkerd	0.004	0.005	0.005	0.006
memory data plane istio	188.5	188.7	188.7	188.9
memory data plane linkerd	26.6	26.9	26.9	27.1
memory control plane istio	110.2	111.5	111.0	113.8
memory control plane linkerd	141.0	141.3	141.3	142.1

5.2 Ressourcenverbrauch

Exemplarisch sind hier die Ergebnisse für 25 und 500 RPS aufgeführt, um den Unterschied zwischen niedriger und hoher Last darzustellen. Auf den Ressourcenverbrauch der Control-Plane nimmt die Last scheinbar keinen Einfluss. Die genutzte CPU-Zeit ist zwischen beiden Service-Meshes ähnlich, wobei Linkerd minimal weniger Ressourcen nutzt. In absoluten Zahlen ist dies aber nicht ausschlaggebend für den Ressourcenverbrauch der Meshes. Auch der Arbeitsspeicherverbrauch der Control-Plane scheint weitestgehend unbeeinflusst von der Last auf dem System. Interessant ist allerdings, dass Linkerd mehr Arbeitsspeicher für seine Control-Plane benötigt. Das Experiment von Fromm [18] zeigt, dass Linkerd einen sehr viel geringen Arbeitesspeicherverbrauch aufweist und auch die später von Buoyant veröffentlichen Benchmarks [22, 23], die das gleiche Setup nutzen, zeigen einen signifikanten Unterschied zugunsten von Linkerd. Die Ergebnisse der Data-Plane zeigen, dass Linkerd erheblich weniger Arbeitsspeicher verbraucht, was vermutlich auf die Unterschiede in der Architektur der Proxies zurückzuführen ist (siehe Abschnitt 2.6). Bei niedriger Last verbrauchen Istios Proxies fast sieben mal soviel Arbeitsspeicher. Bei hoher Last ist es immerhin noch gut doppelt soviel. Die CPU-Ressourcen der Data-Plane werden durch Istio bei niedriger Last um ungefähr 30% und bei hoher Last um ca. 34% mehr belastet.

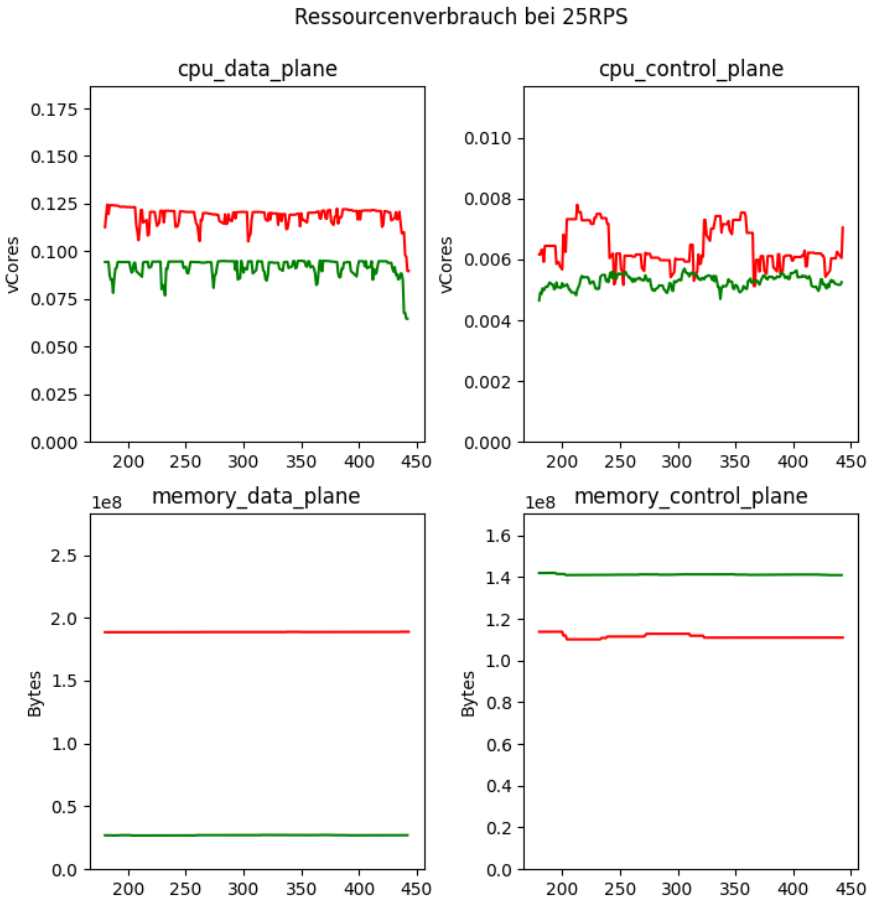

Abbildung 5.6: Ressourcenverbrauch bei 25 RPS

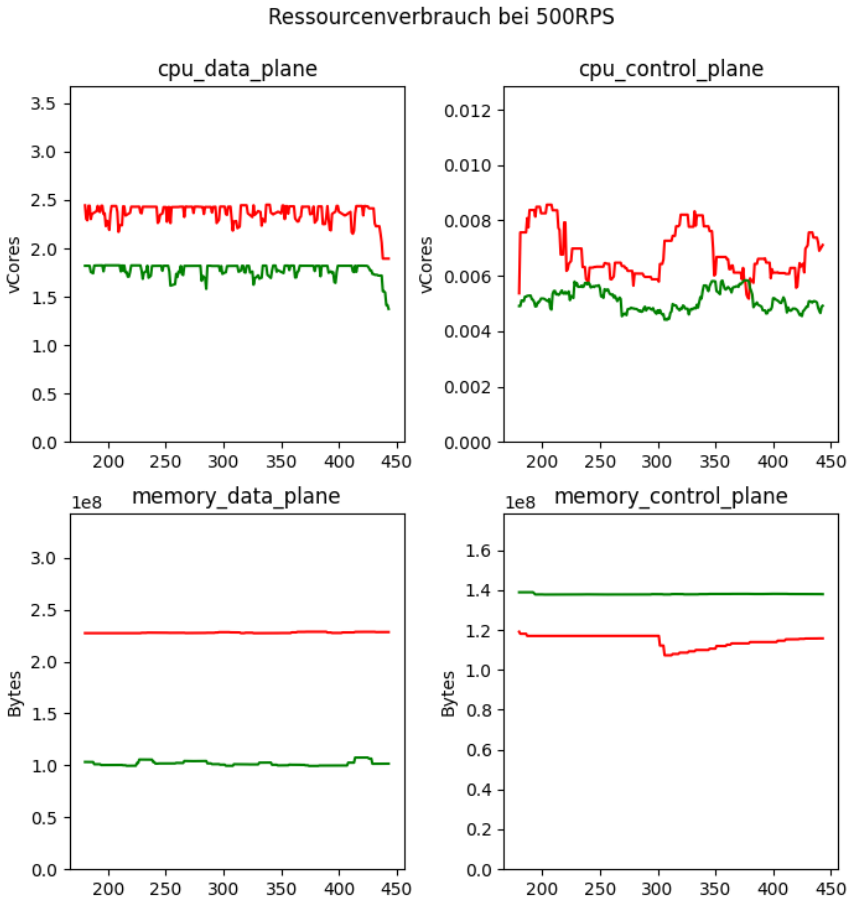

Abbildung 5.7: Ressourcenverbrauch bei 500 RPS

Tabelle 5.7: Ressourcenverbrauch bei 500 RPS

	min	avg	median	max
cpu data plane istio	1.894	2.363	2.400	2.451
cpu data plane linkerd	1.375	1.778	1.819	1.825
cpu control plane istio	0.005	0.007	0.006	0.009
cpu control plane linkerd	0.004	0.005	0.005	0.006
memory data plane istio	227.4	227.9	227.8	228.8
memory data plane linkerd	99.4	101.7	101.1	107.5
memory control plane istio	107.2	114.6	115.7	119.1
memory control plane linkerd	137.7	137.9	137.9	138.9

6 Schlussfolgerung

Die Messergebnisse zeigen, dass Linkerd sowohl bei der Latenz als auch Ressourcenverbrauch vorne liegt. Allerdings sind die Unterschiede nicht so groß wie die, die durch Buoyant gemessen wurden [22, 23]. Der Ressourcenverbrauch der Control-Planes der beiden Service-Meshes ist nahezu identisch. Die Unterschiede in der Data-Plane sind vermutlich auf die Unterschiede der Proxies zurückzuführen (vgl. Unterabschnitt 2.6). Da bei einer wachsenden Anzahl an Microservices aber auch die Anzahl der Proxies steigt, ist die Data-Plane der Teil des Service-Mesh, der am meisten von Optimierung profitiert. Die Ergebnisse des Experiments geben einen ersten Anhaltspunkt wie viel Overhead ein Service-Mesh in Sachen Latenz und Ressourcenverbrauch mit sich bringt und wie die Unterschiede zwischen konkreten Implementierungen ausfallen können. Zu berücksichtigen ist hier, dass das Feature-Set minimal war. Es wurde lediglich mTLS aktiviert, um die Kommunikation zwischen den Services zu sichern. Wie in Abschnitt 1.5 erwähnt, kann ein anderes Feature-Set zu anderen Ergebnissen führen.

6.1 Ausblick

Wie zuvor erwähnt, bieten Service-Meshes eine Vielzahl von Features (vlg Abschnitt 1.3). Deren Einfluss auf die Performance kann genauer untersucht werden. Auch Unterschiede der Payloads können Einfluss nehmen [30]. Die Payloads in diesem Experiment waren vergleichsweise klein. Größere Payloads im Mega- oder sogar Gigabyte-Bereich zu untersuchen ist eine weitere Möglichkeit Service-Meshes genauer zu untersuchen. Skalierung ist ein weiterer Punkt der untersucht werden kann. Die Ergebnisse dieses Experiments nutzen keines der Features die Kubernetes zur Verfügung stellt, um die Applikation zu skalieren. Ohne Skalierung lassen sich zwar die theoretisch erreichbaren Performance-Werte messen, allerdings würde die Applikation üblicherweise ab einer gewissen Last horizontal skalieren (Unterabschnitt 2.1.7). Welchen Einfluss ein Service-Mesh auf diesen Skalierungsprozess hat, ist ein weiterer Bereich der genauer untersucht werden kann. Abschließend sind auch noch Service-Mesh-Architekturen zu nennen, die mit dem Sidecar-Modell brechen und versuchen einen Teil der Funktionalität die ein Service-Mesh bietet mittels extended Berkeley Packet Filter (eBPF) in die Kernel-Ebene

zu verschieben. Hierdurch soll die Performance von Service-Meshes weiter verbessert werden [31, 32].

Abbildungsverzeichnis

Tabellenverzeichnis

Literaturverzeichnis

[1] Sam Newman. *Monolith to microservices: evolutionary patterns to transform your monolith*. O'Reilly Media, 2019.

[2] Anjali Khatri and Vikram Khatri. *Mastering Service Mesh: Enhance, secure, and observe cloud-native applications with Istio, Linkerd, and Consul*. Packt Publishing Ltd, 2020.

[3] Deployment timeline. https://kubernetes.io/docs/concepts/overview/. Accessed: 2023-06-22.

[4] Peter Deutsch. The eight fallacies of distributed computing. *URL: http://today. java. net/jag/Fallacies. html*, 1994.

[5] Javad Ghofrani and Daniel Lübke. Challenges of microservices architecture: A survey on the state of the practice. *ZEUS*, 2018:1–8, 2018.

[6] Service mesh comparison. https://servicemesh.es/. Accessed: 2023-06-15.

[7] Architecture. https://istio.io/latest/docs/ops/deployment/architecture/. Accessed: 2023-06-15.

[8] Lee Calcote and Zack Butcher. *Istio: Up and running: Using a service mesh to connect, secure, control, and observe*. O'Reilly Media, 2019.

[9] Istioupdate. https://istio.io/latest/news/releases/1.5.x/announcing-1.5/. Accessed: 2023-06-16.

[10] Architecture. https://linkerd.io/2.13/reference/architecture/. Accessed: 2023-06-16.

[11] Why linkerd doesn't use envoy. https://linkerd.io/2020/12/03/why-linkerd-doesnt-use-envoy/#fnref:6. Accessed: 2023-06-15.

[12] Istio ingress control. https://istio.io/latest/docs/tasks/traffic-management/ingress/ingress-control/. Accessed: 2023-06-15.

[13] Istio egress control. https://istio.io/latest/docs/tasks/traffic-management/egress/egress-control/. Accessed: 2023-06-15.

[14] Rust ownership. https://doc.rust-lang.org/book/ch04-01-what-is-ownership.html. Accessed: 2023-06-15.

[15] Under the hood of linkerd's state-of-the-art rust proxy, linkerd2-proxy. https://linkerd.io/2020/07/23/under-the-hood-of-linkerds-state-of-the-art-rust-proxy-linkerd2-proxy/. Accessed: 2023-06-15.

[16] Benchmarking istio 1.5.0 and linkerd 2.7.1 (master thesis). https://elastisys.com/benchmarking-istio-linkerd-erik-dahlberg-master-thesis/. Accessed: 2023-06-22.

[17] Jóakim von Kistowski, Simon Eismann, Norbert Schmitt, André Bauer, Johannes Grohmann, and Samuel Kounev. TeaStore: A Micro-Service Reference Application for Benchmarking, Modeling and Resource Management Research. In *Proceedings of the 26th IEEE International Symposium on the Modelling, Analysis, and Simulation of Computer and Telecommunication Systems*, MASCOTS '18, September 2018.

[18] Performance benchmark analysis of istio and linkerd. https://kinvolk.io/blog/2019/05/performance-benchmark-analysis-of-istio-and-linkerd/. Accessed: 2023-06-23.

[19] Benchmarking istio & linkerd cpu. https://medium.com/@michael_87395/benchmarking-istio-linkerd-cpu-c36287e32781. Accessed: 2023-06-23.

[20] Benchmarking istio & linkerd cpu at scale. https://medium.com/@michael_87395/benchmarking-istio-linkerd-cpu-at-scale-5f2cfc97c7fa. Accessed: 2023-06-23.

[21] Service mesh performance evaluation — istio, linkerd, kuma and consul. https://medium.com/elca-it/service-mesh-performance-evaluation-istio-linkerd-kuma-and-consul-d8a89390d630. Accessed: 2023-06-23.

[22] Benchmarking linkerd and istio. https://linkerd.io/2021/05/27/linkerd-vs-istio-benchmarks/. Accessed: 2023-06-23.

[23] Benchmarking linkerd and istio: 2021 redux. https://linkerd.io/2021/11/29/linkerd-vs-istio-benchmarks-2021/. Accessed: 2023-06-23.

[24] Yehia Elkhatib and Jose Povedano Poyato. An evaluation of service mesh frameworks for edge systems. In *International Workshop on Edge Systems, Analytics and Networking (EdgeSys)*. ACM, May 2023.

[25] Toke Høiland-Jørgensen, Bengt Ahlgren, Per Hurtig, and Anna Brunstrom. Measuring latency variation in the internet. In *Proceedings of the 12th International on Conference on emerging Networking EXperiments and Technologies*, pages 473–480, 2016.

[26] Handling ingress traffic. https://linkerd.io/2.13/tasks/using-ingress/. Accessed: 2023-06-23.

[27] Using helm. https://helm.sh/docs/intro/using_helm/. Accessed: 2023-06-23.

[28] Core concepts. https://fluxcd.io/flux/concepts/. Accessed: 2023-06-23.

[29] Flux for helm users. https://fluxcd.io/flux/use-cases/helm/. Accessed: 2023-06-23.

[30] Performance and scalability. https://istio.io/v1.16/docs/ops/deployment/performance-and-scalability/. Accessed: 2023-06-23.

[31] Mohammad Reza Saleh Sedghpour and Paul Townend. Service mesh and ebpf-powered microservices: A survey and future directions. In *2022 IEEE International Conference on Service-Oriented System Engineering (SOSE)*, pages 176–184. IEEE, 2022.

[32] Cilium service mesh - thomas graf, isovalent. https://www.youtube.com/watch?v=mpwTkm53YTY. Accessed: 2023-06-23.

Hiermit versichere ich, dass ich die vorliegende Arbeit ohne fremde Hilfe selbständig verfasst und nur die angegebenen Hilfsmittel benutzt habe.

Hamburg, 17. August 2023 Willem-Lennart Draeger