Marius Jøsok Nesset

Evaluating Performance And Security Characteristics Of Service Mesh Technologies In A Rancher 2.X Environment

Service Mesh Evaluation

Bachelor's project in IT-Operations and Information Security
Supervisor: Erik Hjelmås
May 2021

Bachelor's project

NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Dept. of Information Security and Communication Technology

NTNU
Norwegian University of
Science and Technology

Marius Jøsok Nesset

Evaluating Performance And Security Characteristics Of Service Mesh Technologies In A Rancher 2.X Environment

Service Mesh Evaluation

Bachelor's project in IT-Operations and Information Security
Supervisor: Erik Hjelmås
May 2021

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Dept. of Information Security and Communication Technology

NTNU
Norwegian University of
Science and Technology

Evaluating Performance And Security Characteristics Of Service Mesh Technologies In A Rancher 2.X Environment

Marius Jøsok Nesset

May 20, 2021

Abstract

With today's operations trends tending towards more and more (micro-)services and containers, many feel the need to employ service mesh technologies to manage all these services.

This thesis serves to measure and evaluate the performance and security characteristics of some of these service mesh technologies; Allowing the reader to potentiality use it to aid their decision making on which service mesh to select.

Sammendrag

Dagens IT-drifing trender mot stadig flere (micro-)serviser og konteinere, mange derfor enten vurder-eller allerede har tatt i bruk service mesh teknologier for å handtere alle disse servisene.

Denne bacheloroppgaven ønsker å vere en hjelp i dette valget ved å evaluere ytelse and sikkerhets karakteistiker ved hvisse utvalgte service mesh technologier.

Acknowledgements

First of all I want to thank Thilo Fromm from Kinvolk for their previous work bench marking the performance of Linkerd and Istio; While none of their benchmarks was used for this thesis, they were instrumental in giving me an early understanding of the nature of the problem.

I wanna thank Buypass not only for being an amazing and very vocal client, but also for giving me an jump start on the practical part of this thesis by organizing a private lab-session with one of their engineers, which taught me the basics of Rancher and K3S.

The diagrams in this thesis depicting Kubernetes resources and their relationships, makes use of the open source Kubernetes icon set made by the amazing Kubernetes community; The icon set is released under the Apache-2 License and can be found on the Kubernetes' community Github: `https://github.com/kubernetes/community/tree/master/icons`

I wanna thank NTNU for providing me with an amazing mentor Erik Hjelmås, and for providing me with an large Openstack environment where I could conduct my tests in.

Lastly I wanna thank former Amazon Ring NetOps-team lead, Katie 'Fusl' Holly; for aiding me in the selection process of loadgenerator and for giving me a crash course in Grafana.

Contents

Figures

Tables

Acronyms

ACL Access Control List. 23, 25

CNCF Cloud Native Computing Foundation. 3, 33

CRD Custom Resource Definition. 21, 25, *Glossary:* Custom Resource Definition

RPS Requests Per Second. 6, 13, 14, 19, 33, *Glossary:* Requests Per Second

SMI The Service Mesh Interface Specification. 21, 25, *Glossary:* Service Mesh Interface

Glossary

Custom Resource Definition An extension to kubernetes which defines a new resource kind.. 21

Requests Per Second A metric of how many requests are being sent or received every second, i.e. A request rate of 100 RPS means 100 requests are being sent every second.. 6

Service Mesh Interface A specification defining a set of standard resources for service meshes in Kubernetes. 21

Chapter 1

Introduction

1.1 Task

The client wants a datacenter agnostic assessment and evaluation done of a certain service mesh technologies' performance and security characteristics in a rancher 2.X environment, with focus on:

- Access control policies
  1. Scoping granularity level
  2. Ease of configuration
- Operational overhead
  1. Difficulty learning
  2. Documentation
- Performance
  1. Scaleability
  2. Overhead
  3. Resource usage
- Monitoring
- Mutual TLS
- Maturity

The selection of service mesh technologies to investigate was made by the client and consists of the following three service mesh technologies:

- Rancher's version of Istio
- LinkerD 2
- Traefik Mesh

1.2 Scope and Limitations

Due to the complexity of these systems, this thesis will only be directly evaluating the service mesh technologies themselves. Their components, such as their proxies, are out of scope. However, where it is clear that the feature being evaluated is implemented in the proxy's configuration and not the mesh's, further reference for the reader will be given.

Rancher environment

As this thesis is about evaluating in a Rancher 2.X environment, integration with Rancher is prioritized before all; for example, even though the latest version of Istio at the time of writing is 1.10, the latest supported by Rancher officially is 1.9.3; Thus it is the version of Istio evaluated.

Not an comparison

While this document is meant to be a potential aid in the selection process of a service mesh technology, it is not meant to make that selection for the reader. Therefore, particular emphasis is put on *evaluating* the technologies, not directly *comparing* them towards each other.

Stock configuration

While it is possible to achieve better performance on all the service meshes by tweaking them, it was decided not to tweak the meshes favouring keeping the configuration as close to stock as possible. By keeping the stock configuration, the performance measurements reflect a general use case, not that of a tailor-tuned one.

1.3 Service Mesh Technologies

This section gives a brief description of service mesh technologies, how one works, some relevant terminology, and a brief introduction to each of the selected service mesh technologies.

1.3.1 What Is A Service Mesh Technology

A *service mesh technology* is a dedicated infrastructure layer that controls traffic flow between services. It may also serve to monitor the services health, performance and traffic usage. [1]

Brief terminology

A *service* is an application or part of an application that has its traffic managed by a service mesh technology, for example, a Kubernetes service or pod.

More formally, a distinction between meshed and non-meshed services might be made, where meshed services referrer to those whose traffic is managed by the service mesh technology. Similarly, a distinction between meshed and non-meshed traffic can be made.

The term *service mesh* can depending on the context, refer to the overlaying service mesh technology or the collection of meshed services within a particular domain.

1.3.2 Selected Service Mesh Technologies

This section serves as a brief introduction to each of the service mesh technologies to be investigated, as selected by the client in Section 1.1.

Istio

Istio was announced on the 24th of May 2017 as a joint effort between Google, IBM, and Lyft [2]. It is the most popular service mesh in production, with a 47% market share according to the Cloud Native Computing Foundation (CNCF)'s 2020 survey [3]. It was designed to be extensible [4] and it utilizes an extended version of Envoy as a sidecar proxy [5].

Linkerd

Linkerd is widely considered to be the oldest service mesh [6], it began development in 2015 [7] with version 1.0 releasing on the 25th of April 2017 [8]. It was originally developed by Buoyant but has since changed hands to the CNCF. Like Istio, it has a sidecar proxy-based architecture; however, it utilizes its own micro proxy instead of a pre-existing one [9]. It aims to be lightweight and fast.
Please note that the terms LinkerD and LinkerD 2 are used interchangeably and both refer to the rewritten version 2 of LinkerD known formally as LinkerD 2.

Traefik Mesh

Originally introduced as Maesh by Containous on the 4th of September 2019 [10], and later renamed to its current name Traefik Mesh on the 1st of October 2020 [11] as part of a company wide rebranding [12]. Traefik mesh aims to be a non-invasive service mesh, it achieves this goal by not using sidecar proxies or rewriting IPTables inside of PODs [13], instead it makes its own DNS zone traefik.mesh, which serves as a drop-in replacement for cluster.local [14]; More on how this works is described in Section 1.3.3.

1.3.3 How traffic is meshed

When we say meshed traffic, we mean traffic managed by a service mesh. This management is achieved by routing the traffic through a proxy. In other words, meshed traffic just means traffic being routed through the service mesh's proxy. Within the scope of the service mesh technologies being evaluated, there are two approaches to achieve this, both of which are described below.

An in-depth example of meshed traffic alongside figures visualizing traffic flow can be found in Section 2.3

Istio and LinkerD both use what is called the *sidecar proxy* approach [5] [15], where an admission hook is installed [16]; The hook triggers when a pod belonging to a meshed service is created, and injects an extra container containing a proxy application into the pod, as well as an initialization container which rewrites the IPTables inside the application container to route all traffic through the proxy container [16]. The injected proxy container is referred to as the sidecar proxy, and is where most of the features are implemented as it's where all traffic to and from the pod will now flow through.

As all pods deployed will trigger the admission hook, Istio and Linkerd use a label and an annotation, respectively, on either the pod, service or namespace to denote which pods should be injected into [17]. They both offer a command-line tool to automatically add these annotations/labels, in the form of istioctl and linkerdctl, respectively. Though note that for any pod to get meshed, it must be redeployed as the admission hook is only then fired [17]

Traefik Mesh deploys a stock Traefik Proxy on every host instead of a sidecar proxy alongside every pod [11]. It then creates a new DNS zone called traefik.mesh, which serves as a drop-in replacement for cluster.local, this DNS zone returns records of the traefik proxy deployed on the current host. Note that with this approach no IPTables are rewritten; Thus, it is non-invasive. However, a downside is that it does not support automatic service meshing as it requires you to change the domain of endpoints, being called by the applications to service.namespace.traefik.mesh instead of service.namespace.cluster.local [14].

Chapter 2

Performance

This chapter is dedicated to measuring the performance characteristics of the service mesh technologies. It details the tooling selection, methodology, test environments used to measure various performance characteristics. The results can be found towards the end of each test. Note that that the final evaluation happens in Chapter 4 and that no direct comparison will be drawn here as stated in the limitations found in Section 1.2.

2.1 Tooling selection

This section serves to introduce the tools used for measuring the performance characteristics of the service meshes.

2.1.1 Target application

The open-source example application *Emojivoto* by Buoyant was selected to be the target application for the performance tests. It was selected due to its commonality and because it contains both HTTP REST and gRPC services, allowing the performance of both protocols to be tested. You can find the original *Emojivoto* on Github.[1]

Note that the *vote-bot* deployment typically found in *Emojivoto* has been removed in favour of the load generator described in Section 2.1.2. You can find the modified deployment files used for the performance tests on Gitlab.[2]

[1]`https://github.com/BuoyantIO/emojivoto.git`

[2]`https://gitlab.com/DCSG2900/workspace/-/tree/master/apps/emojivoto`

2.1.2 Load generator

In order to measure the performance of something, a stimulus is often required; within the scope of web services, this stimulus often takes the form of a load generator. The selection of which was done based on the following criteria, I came up with based on previous experience:

1. What data they can record.
2. How precise they can record it (particularly response times).
3. The ability to issue requests at a constant Requests per second (RPS) rate.
4. Must not suffer from Coordinated Omission as described in 2.1.2.

The Coordinated Omission Problem

Coordinated Omission is a fault with how some load generators with constant RPS capabilities schedule their requests. It occurs when a load generator only issues new requests after the previous ones have completed, instead of at the point in time needed to maintain the requested RPS rate [18].

This behaviour becomes problematic if response times become higher than the needed request rate, as then the effective RPS rate will become lower than requested [18] resulting in inconsistent data. Worse still is that this behaviour can be masking out lousy performance; since subsequent requests, after a slow one, will wait until the slow request completes, resulting in the severity of periodic slowness being under-reported as fewer requests than usual will be issued in such a situation.

The open source load generator *Vegeta* by Tomás Senart was selected as it fufills all the criteria including not suffering from coordinated omission [19], the source code for *Vegeta* can be found on its Github. [3]

Target Endpoint Selection

The /API/leaderboard endpoint of *Emojivoto*'s web-svc was selected as the target for *Vegeta*, as it causes web-svc to request further information from svc-voting and svc-emoji, these calls are done over GRPC; Thus, a total of three service calls, of which two are GRPC calls are issued by per GET request to the web-svc service's endpoint /API/leaderboard. A visualization of this can be found in Section 2.3.

2.1.3 Monitoring

A monitoring application is needed to record metrics not already recorded by *Vegeta*, of particular concern is the collection of resource usages. While many monitoring solutions exists; The selection of a monitoring solution was greatly reduced by the fact offical Rancher chart for Istio depends on the *rancher-monitoring* chart

[3]`https://github.com/tsenart/vegeta`

[20], resulting in the *rancher-monitoring* chart needing to have been installed regardless.

As *rancher-monitoring* comes with Grafana, Prometheus, node-exporter and prom-operator, no additional monitoring tools were required. The *rancher-monitoring* chart was present for all performance tests for all service meshes. Specifically version 14.5.100 of the *rancher-monitoring* chart is used.

Note that Traefik Mesh and Linkerd 2 both come with their own instances of Prometheus and Grafana, meaning that when testing these meshes two instances of Grafana and Prometheus were present in the cluster.

It is possible to disable these instances and configure Traefik Mesh and Linkerd 2 to use the prometheus found in *rancher-monitoring* directly [21]; However this was not done in favour of running the service meshes as close to stock as possible, it was also deemed to be a none-issue as the impact is negigable due to their relatively low scrape intervals of 10s for Traefik Mesh [22] and 30s for Linkerd 2 [23].

2.2 Testing environment

In order to create a consistent test environment an Openstack HEAT template that automatically deploys the testing environment was developed for this thesis, it can be found alongside other tools made for this thesis on Gitlab.[4]

The template is configured and deployed as an Openstack stack in NTNU's Openstack Environment. In order to ensure a fresh slate before every test, the entire stack is redeployed before every test and every test re-run.

The template deploys a *Manager* node and configures it to host a bind9 DNS server and a K3S version 1.19.7 cluster where Rancher is then installed. The nodes needed by the high availability environment described in Section 2.2.2 are also deployed and configured by the template. The specifications for the *Manager* node are listed in Table 2.1.

Manager Node	
CPU	Dual-vCPU; 2C 1T
RAM	16 GiB
Platform	NTNU Openstack
Flavor	m1.large
OS	Ubuntu 18.04 LTS
Kernel	4.15.0-76-generic

Table 2.1: Manager Node Specifications

[4] `https://gitlab.com/DCSG2900`

Rancher is configured in multicluster mode managing three clusters, the *local* cluster, the high availability *dcsg2900ha* cluster and the low latency dcsg2900ll cluster. The *local* cluster is where rancher itself runs, being the K3S cluster running on the *Manager* node.

The low latency and high availability clusters are specialized test environments running version 1.1.17 of RKE and are both imported into and managed by Rancher. They are described in Section 2.2.1 and Section 2.2.2 respectively. Each test describes in its methodology section which test environment it utilizes.

2.2.1 Low Latency Testing Environment

This section describes the low latency test environment, which makes up the *dcsg2900ll* RKE cluster as stated earlier in Section 2.2. This environment is not designed to accurately simulate a real-world scenario, but instead allows for consistent and accurate latency measurements by removing as many "black box" variables as possible.

The problems with measuring a black box

Idealistically we would be able to describe the performance of all systems in a manner not tied to the variable performance of the medium it is run on; An concrete example of this is expressing the performance of sorting algorithms using Big-O notation. Unfortunately, this approach becomes unfeasible at the complexity level of service mesh technologies. Instead, we resort to essentially measure a black box's latency to process some input in a controlled environment.

However, designing such an environment suitable to measure latencies accurately is a crucial but none trivial task; As many variables can attribute to a system's overall latency, many of which are intermittent and overlooked by many when designing test environments.

For example, the response times between two services not only reflect how fast the services processed the request but also:

- How fast the network infrastructure between the two servers could transmit the data between them.
- What the availability of CPU time was at the time of arrival.
- How fast the CPU could process it and create a response
- Finally, the networking infrastructures time again to send the response.

All of these variables can change over time; Thus, it is crucial to remove as many of these variables as possible in order to get accurate and consistent measurements.

Design decisions behind the low latency environment

Therefore a special environment was needed. This section serves to describe the measures taken to reduce these aforementioned variables as much as possible.

In order to remove latencies associated with networking between nodes running the latency tests, a single node called *latency-test* runs all the processes associated with the test, including monitoring, the test application and load generator. This removes latencies associated with cross-node networking, as all communication between services occurs internally.

To reduce variable processing availability associated with virtualization the *latency-test* node runs directly on bare metal hardware and is not virtualized like the other nodes, the specifications to this node is found in Table 2.3. Using a bare-metal node not only removes the associated overhead with virtualization and hypervisors but also reduces the inherent volatility of processing power that can occur in virtualized environments with shared resources. The hardware node is also underutilized in order to further ensure no resource availability problems.

Table 2.2: Low latency Environment node specifications

Latency Test Node	
CPU	AMD EPYC 7402P 24-Core Processor @ 2.8GHz
RAM	64GB
Platform	Equinix Metal
RKE Roles	Worker
OS	Flatcar Linux 2765.2.3 Stable
Kernel	5.10.32

Table 2.3: Latency-test node specs

2.2.2 High availability environment

For tests less concerned with latencies and more concerned with reflecting operational costs such as resource consumption and utilization, an environment reflecting a more standard setup was needed. A High availability environment per RKE's recommendations [24] is therefore set up. The OpenStack template deploys all nodes in this environment.

The High availability environment consists of three *etcd* nodes, two *control-plane* nodes and four *worker* nodes. Each node only have a single RKE role as recommended for downstream user application clusters [24], the RKE role assigned is the same as their name, i.e etcd nodes have the etcd role. The specs of each node type is found in table 2.4
The worker nodes are where the workload associated with a test are scheduled to; how the test workload is dispersed among these nodes is documented in the tests using this environment.

Etcd Nodes	
CPU	Dual vCPU; 2C 1T
RAM	16 GiB
Platform	NTNU Openstack
Flavor	m1.large
RKE Role	etcd
Count	3
OS	Ubuntu 18.04 LTS
Kernel	4.15.0-76-generic

(a) Specifications for etcd nodes.

Worker Nodes	
CPU	Dual vCPU; 4C 1T
RAM	16 GiB
Platform	NTNU Openstack
Flavor	c1.tiny
RKE Role	worker
Count	4
OS	Ubuntu 18.04 LTS
Kernel	4.15.0-76-generic

(b) Specifications for worker nodes.

Controlplane Nodes	
CPU	Dual vCPU; 2C 1T
RAM	16 GiB
Platform	NTNU Openstack
Flavor	m1.large
RKE Role	controlplane
Count	2
OS	Ubuntu 18.04 LTS
Kernel	4.15.0-76-generic

(c) Specifications for controlplane nodes.

Table 2.4: Node type specifications for high availability environment cluster.

2.3 Visualizing meshed traffic of target

This section serves to visualize the flow of network traffic from *Emojivoto*'s *web-svc* pod when the endpoint /api/leaderboard has been called; being the endpoint selected to be the target for the tests as described in Section 2.1.2. Note that the responses to these calls is not visualized.

2.3.1 With no service mesh installed

The *web-svc* calls a function in svc/voting and svc/emoji over gRPC, this call is directed towards the service, which then resolves to the pod.

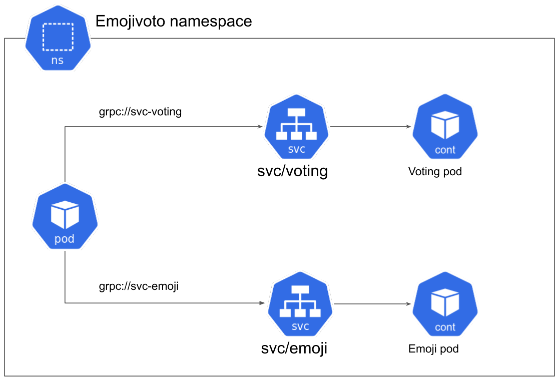


Figure 2.1: Traffic flow from Web pod when not in a service mesh.

2.3.2 With Traefik Mesh installed

Note that the endpoint has been appended with traefik.mesh in order to route the traffic through the mesh as described in Section 1.3.2. They are now resolved to a set of services created by Traefik Mesh which forwards the request to the Traefik proxy pod deployed on each host; The Traefik proxy then forwards it to the real services in emojivoto which forwards it to the destination pods.

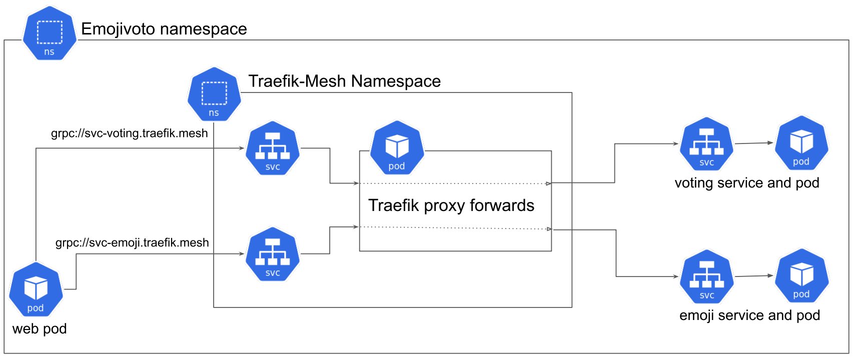
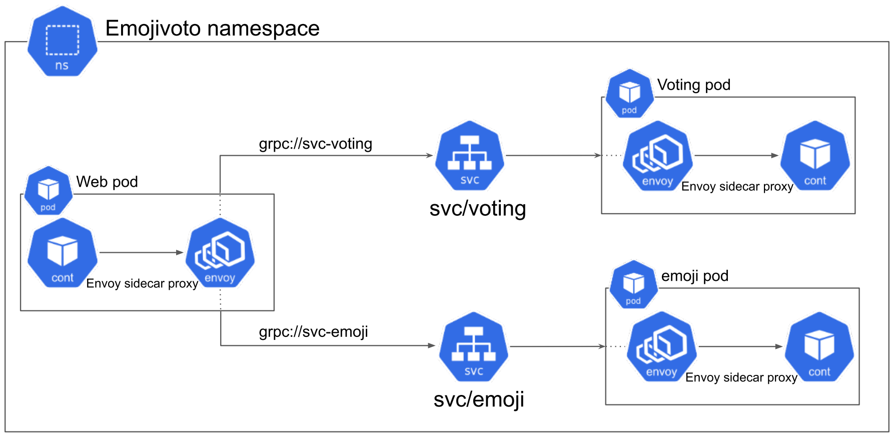


Figure 2.2: Traffic flow from Web pod when using Traefik Mesh.

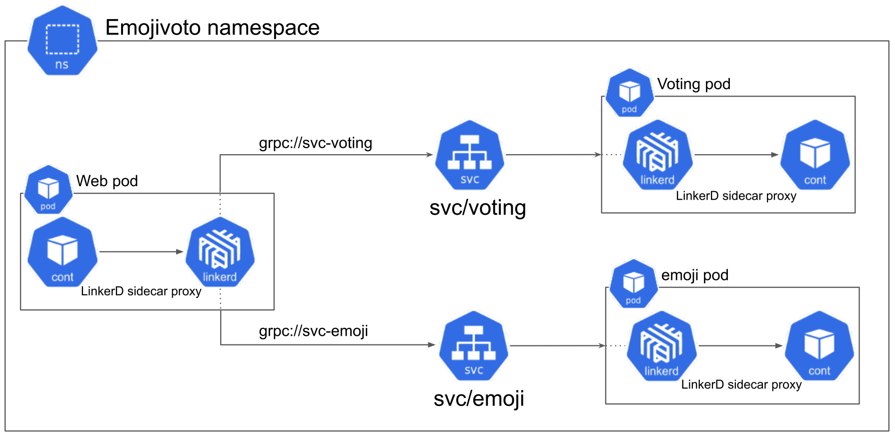
2.3.3 With Istio or Linkerd installed

Since both Linkerd and Istio both use a sidecar proxy, their traffic flow is virtually identical except for which proxy they utilize as their sidecar. Note that since we are not concerned with multiple containers inside the same pod, the visualization of pods has changed to show the containers running inside of them.

Inside the web pod, the web container has had its IP tables rewritten and sends its requests to the sidecar proxy container running along side it. The sidecar proxy then forwards the request to the svc/voting and svc/emoji services, who forward it to the voting and emoji pods, respectively; The request is then fetched by the sidecar containers for these pods, before finally being forwarded to the destination containers.



(a) Istio traffic flow



(b) LinkerD 2 traffic flow

Figure 2.3: Traffic flow from Web pod with sidecar proxy based service meshes.

2.4 Measuring Latencies At Various Constant Loads

As mentioned in 1.3.3, and visualized in 2.3; Traffic flowing through the mesh flows through a proxy, where most of their features are implemented. From a latency standpoint, going through the proxy of these service meshes is not without cost as it not only goes through an extra hop, but time will be spent processing it as well.

Therefore, the latencies introduced by these proxies are an important metric to investigate, as it affects all traffic going through the service mesh. Especially with LinkerD 2 and Istio, due to them having one proxy per pod, if both sides of the connection are part of the service mesh, traffic will flow through 2 proxies and not just one.

2.4.1 Methodology

The low latency environment described in Section 2.2.1 is used for this test. All resources related to the test are deployed to the bare-metal latency-test worker node, including the service mesh being tested. Each service mesh is individually installed, tested and cleaned up; this is repeated three times.

For each test

All the services belonging to the target application described in Section 2.1.1 and the load generator itself described in Section 2.1.2 are meshed.

An automated script *testandfetch.sh* is then used to run the test, it configures *Vegeta* to apply a load starting at 100RPS and increasing in discrete steps every minute by another 100RPS ending with a 2000 RPS run. For each discrete step the results of that run are pulled down before the next one is launched, the target of the load is always that described in Section 2.1.2, being the /api/leaderboard endpoint of the *web-svc* service. The script can be found on Gitlab.[5]

2.4.2 Metrics

What is measured here is the response time of calls to /api/leaderboard at various constant loads. There are two popular ways to display and interpret this data: percentiles and heatmap histograms.

Percentiles

Percentiles are a popular method of interpreting a large data set, they are a form of averages which denotes a value which that X percent of your data is either less or equal to, X being the percentile often prefixed by the capital letter 'P'. [25]

[5]`https://gitlab.com/DCSG2900/workspace/-/blob/master/Latency/testandfetch.sh`

For example, a P99 response time of 2ms means that of 99% of all requests completed within 2ms; alternatively, one could say the 99th percentile of the application is 2ms.

To calculate a precentile Sort your dataset by the metric you wanna calculate the percentile for, then discard the inverse to the percentile being calculated precent of the worst-performing datapoint; the remaining worst data point is now your percentile.

Heatmap

A more modern interpretation is heatmaps; unlike percentiles which discards a set of data and gives you an upper bound for your response times, heatmaps can reveal behavioural relationships in the data set.

A heatmap is essentially a table with an discrete X and Y axis. The X-axis is often either time or increasing RPS load, while the Y-axis is split into ranges often called buckets or bins. The entire dataset is then categorized into the cells where they fit the X and Y-axis criteria. A program was written for this thesis to generate the heatmaps from CSV encoded Vegeta output, and it can be found on Gitlab.[6]

To aid readability the cells are coloured based on a percentage calculated for each cell; the percentage is also shown in parentheses behind the count for each cell.

The percentage is calculated based on the count in that cell and the total count for that discrete step of the X-axis. A grayscale colourmap where perfect black is 0% and perfect white is 100% is used for the cell colour.

> 5 ms	0 (0%)	0 (0%)
5-7 ms	0 (0%)	0 (0%)
3-5 ms	0 (0%)	0 (0%)
1-3 ms	12 (80%)	11 (73%)
< 1 ms	3 (20%)	4 (27%)
	Now -5m	**Now**

(a) Bucket size of 2ms reveals no behavioural pattern.

> 2.5 ms	0 (0%)	0 (0%)
2-2.5 ms	7 (46%)	6 (40%)
1.5-2 ms	0 (0%)	0 (0%)
1-1.5 ms	5 (33%)	5 (33%)
< 1 ms	3 (20%)	4 (27%)
	Now -5m	**Now**

(b) Bucket size of 0.5ms reveals an behavioural pattern.

Table 2.5: Example showing how bucket size selection can mask data

The range covered by a bucket or bin is called the bucket size, and it should be constant between all buckets except for the last and first buckets, which catch the bounding extremities.

[6] `https://gitlab.com/DCSG2900/heatmap-generator`

The selection of bucket size is important as it can hide or reveal relationships in the dataset.

For example, a behaviour where every other request took an extra 1ms to respond due to a problem with the cache only living for a short while, can easily be revealed with a small enough bucket size as seen in Table 2.5b, but masked by a larger bucket size as seen in Table 2.5a.

2.5 Results

As the heatmaps are to large to display horizontally, they have been split in two and are displayed vertically. Please see Table 2.7, Table 2.8 and Table 2.9 for the results displayed as heatmaps. P95 and P99 percentiles are depicted in Table 2.6 and Figure 2.4 respectively.

Table 2.6: P95 Latencies in table

rps	**traefik**	**linkerd**	**istio**
200	717μs	1086μs	1354μs
400	681μs	1063μs	1362μs
600	634μs	879μs	1239μs
800	622μs	876μs	1259μs
1000	627μs	868μs	1323μs
1200	697μs	875μs	1466μs
1400	636μs	943μs	1525μs
1600	660μs	989μs	1479μs
1800	732μs	1023μs	1626μs

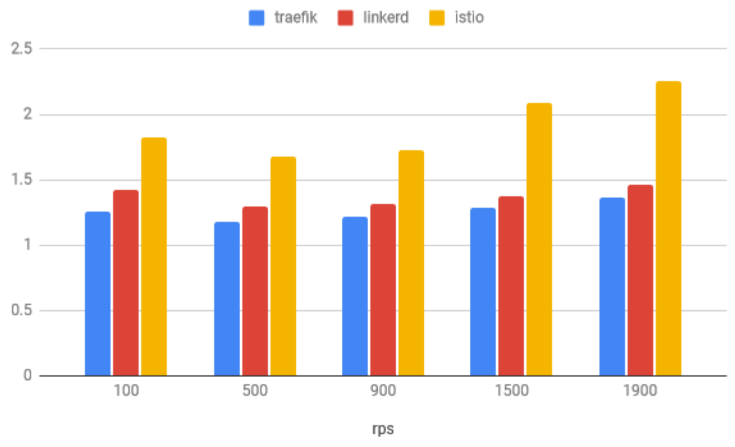


Figure 2.4: P99th response times for various RPS rates.

	100 RPS	200 RPS	300 RPS	400 RPS	500 RPS	600 RPS	700 RPS	800 RPS	900 RPS	1000 RPS
+INF	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)
2.0 *ms*	37 (0.6%)	37 (0.3%)	61 (0.3%)	77 (0.3%)	82 (0.2%)	98 (0.2%)	135 (0.3%)	168 (0.3%)	190 (0.3%)	280 (0.4%)
1.4 *ms*	488 (8.4%)	409 (3.5%)	1040 (5.9%)	851 (3.6%)	819 (2.8%)	1021 (2.9%)	1157 (2.8%)	1509 (3.2%)	1733 (3.3%)	2253 (3.8%)
1.3 *ms*	798 (13.7%)	404 (3.4%)	1654 (9.4%)	1075 (4.6%)	498 (1.7%)	297 (0.8%)	332 (0.8%)	364 (0.7%)	926 (1.7%)	742 (1.2%)
1.2 *ms*	1355 (23.3%)	899 (7.7%)	2391 (13.7%)	1491 (6.4%)	1419 (4.8%)	963 (2.7%)	1031 (2.5%)	1281 (2.7%)	1337 (2.5%)	16012 (27.4%)
1.1 *ms*	2809 (48.4%)	5061 (43.5%)	7731 (44.3%)	9526 (40.9%)	15747 (54.0%)	13461 (38.6%)	16538 (40.6%)	18288 (39.3%)	22233 (42.3%)	23522 (40.3%)
1.0 *ms*	315 (5.4%)	4806 (41.3%)	4559 (26.1%)	10220 (43.9%)	10573 (36.2%)	19004 (54.5%)	21489 (52.8%)	24869 (53.4%)	26020 (49.6%)	15473 (26.5%)
900 *μs*	0 (0.0%)	1 (0.0%)	3 (0.0%)	4 (0.0%)	0 (0.0%)	2 (0.0%)	1 (0.0%)	7 (0.0%)	3 (0.0%)	2 (0.0%)
800 *μs*	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)
700 *μs*	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)
600 *μs*	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)
500 *μs*	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)
400 *μs*	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)
300 *μs*	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)
MIN	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)

(a) Heatmap showing response times from Istio and varying loads from 100PRS to 1000RPS.

	1100 RPS	1300 RPS	1400 RPS	1500 RPS	1600 RPS	1700 RPS	1800 RPS	1900 RPS	2000 RPS
+INF	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)
2.0 *ms*	343 (0.5%)	618 (0.8%)	741 (0.9%)	1099 (1.2%)	849 (0.9%)	1160 (1.1%)	2109 (2.0%)	2234 (2.0%)	5464 (4.7%)
1.4 *ms*	2531 (3.9%)	14225 (19.2%)	30319 (37.1%)	16889 (19.3%)	5096 (5.5%)	8366 (8.4%)	13205 (12.6%)	15690 (14.2%)	33529 (28.8%)
1.3 *ms*	1602 (2.5%)	14002 (18.9%)	15252 (18.7%)	43118 (49.3%)	3828 (4.1%)	10117 (10.2%)	28160 (26.9%)	32626 (29.5%)	24435 (21.0%)
1.2 *ms*	7903 (12.3%)	38126 (51.6%)	6108 (7.4%)	18629 (21.3%)	23121 (25.0%)	25736 (25.9%)	21665 (20.7%)	40351 (36.5%)	44297 (38.1%)
1.1 *ms*	49501 (77.4%)	6743 (9.1%)	17268 (21.1%)	5030 (5.7%)	38475 (41.6%)	32591 (32.8%)	29076 (27.8%)	13102 (11.8%)	7457 (6.4%)
1.0 *ms*	2013 (3.1%)	119 (0.1%)	11833 (14.5%)	2615 (2.9%)	20938 (22.6%)	21161 (21.3%)	10353 (9.9%)	6353 (5.7%)	987 (0.8%)
900 *μs*	7 (0.0%)	0 (0.0%)	0 (0.0%)	1 (0.0%)	0 (0.0%)	8 (0.0%)	1 (0.0%)	3 (0.0%)	0 (0.0%)
800 *μs*	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)
700 *μs*	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)
600 *μs*	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)
500 *μs*	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)
400 *μs*	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)
300 *μs*	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)
MIN	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)

(b) Heatmap showing response times from Istio and varying loads from 1100PRS to 2000RPS.

Table 2.7: Heatmaps of Istio showing response buckets at loads from 100RPS to 2000RPS.

+INF	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)
2.0 *ms*	11 (0.1%)	18 (0.1%)	24 (0.1%)	27 (0.1%)	37 (0.1%)	44 (0.1%)	45 (0.1%)	59 (0.1%)	64 (0.1%)	69 (0.1%)
1.4 *ms*	53 (0.8%)	71 (0.5%)	181 (1.0%)	125 (0.5%)	120 (0.4%)	143 (0.3%)	165 (0.3%)	210 (0.4%)	275 (0.5%)	291 (0.4%)
1.3 *ms*	39 (0.6%)	57 (0.4%)	232 (1.2%)	102 (0.4%)	129 (0.4%)	148 (0.4%)	184 (0.4%)	224 (0.4%)	299 (0.5%)	354 (0.5%)
1.2 *ms*	206 (3.4%)	115 (0.9%)	964 (5.3%)	240 (1.0%)	230 (0.7%)	177 (0.4%)	220 (0.5%)	276 (0.5%)	186 (0.3%)	263 (0.4%)
1.1 *ms*	480 (8.0%)	274 (2.2%)	2025 (11.2%)	472 (1.9%)	157 (0.5%)	105 (0.2%)	145 (0.3%)	158 (0.3%)	203 (0.3%)	196 (0.3%)
1.0 *ms*	790 (13.1%)	626 (5.2%)	2424 (13.4%)	743 (3.1%)	242 (0.8%)	144 (0.4%)	151 (0.3%)	172 (0.3%)	220 (0.4%)	264 (0.4%)
900 *µs*	1299 (21.6%)	1306 (10.9%)	3421 (19.0%)	1286 (5.3%)	644 (2.1%)	584 (1.6%)	586 (1.3%)	671 (1.4%)	575 (1.0%)	668 (1.1%)
800 *µs*	2119 (35.3%)	5476 (45.7%)	6073 (33.8%)	8700 (36.3%)	11133 (37.1%)	14326 (39.8%)	17119 (40.8%)	19429 (40.5%)	18171 (33.7%)	18487 (30.8%)
700 *µs*	991 (16.5%)	4033 (33.6%)	2620 (14.5%)	12258 (51.1%)	17251 (57.6%)	20253 (56.3%)	23299 (55.5%)	26706 (55.7%)	33894 (62.8%)	39276 (65.6%)
600 *µs*	1 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	2 (0.0%)	0 (0.0%)	1 (0.0%)	3 (0.0%)
500 *µs*	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)
400 *µs*	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)
300 *µs*	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)
MIN	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)
	100 RPS	**200 RPS**	**300 RPS**	**400 RPS**	**500 RPS**	**600 RPS**	**700 RPS**	**800 RPS**	**900 RPS**	**1000 RPS**

(a) Heatmap showing response times from Linkerd and varying loads from 100PRS to 1000RPS.

+INF	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)
2.0 *ms*	95 (0.1%)	98 (0.1%)	100 (0.1%)	136 (0.1%)	142 (0.1%)	161 (0.1%)	232 (0.2%)	198 (0.1%)	215 (0.1%)
1.4 *ms*	374 (0.5%)	426 (0.5%)	450 (0.5%)	567 (0.6%)	621 (0.6%)	898 (0.9%)	1044 (1.0%)	1228 (1.1%)	1393 (1.2%)
1.3 *ms*	302 (0.4%)	354 (0.4%)	526 (0.6%)	472 (0.5%)	571 (0.6%)	488 (0.5%)	620 (0.6%)	751 (0.6%)	720 (0.6%)
1.2 *ms*	409 (0.6%)	417 (0.5%)	326 (0.4%)	525 (0.6%)	657 (0.7%)	785 (0.8%)	735 (0.7%)	664 (0.6%)	1024 (0.9%)
1.1 *ms*	274 (0.4%)	265 (0.3%)	404 (0.5%)	439 (0.5%)	602 (0.6%)	798 (0.8%)	997 (0.9%)	1234 (1.1%)	1142 (1.0%)
1.0 *ms*	254 (0.3%)	522 (0.7%)	535 (0.6%)	642 (0.7%)	809 (0.9%)	1306 (1.3%)	1218 (1.1%)	1814 (1.6%)	4136 (3.6%)
900 *µs*	836 (1.2%)	768 (1.0%)	1706 (2.1%)	9914 (11.8%)	3456 (3.8%)	7309 (7.6%)	18090 (17.7%)	16598 (15.4%)	14878 (13.0%)
800 *µs*	20868 (31.6%)	22084 (30.7%)	23883 (30.6%)	31291 (37.3%)	60843 (67.7%)	56354 (58.8%)	40715 (40.0%)	61809 (57.3%)	78653 (69.1%)
700 *µs*	42456 (64.4%)	46916 (65.2%)	49904 (64.1%)	39844 (47.5%)	22075 (24.5%)	27667 (28.8%)	38136 (37.4%)	23475 (21.7%)	11594 (10.1%)
600 *µs*	1 (0.0%)	3 (0.0%)	5 (0.0%)	2 (0.0%)	33 (0.0%)	29 (0.0%)	0 (0.0%)	0 (0.0%)	1 (0.0%)
500 *µs*	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)
400 *µs*	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)
300 *µs*	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)
MIN	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)
	1100 RPS	**1200 RPS**	**1300 RPS**	**1400 RPS**	**1500 RPS**	**1600 RPS**	**1700 RPS**	**1800 RPS**	**1900 RPS**

(b) Heatmap showing response times from Linkerd and varying loads from 1100PRS to 2000RPS.

Table 2.8: Heatmaps of Linkerd showing response buckets at loads from 100RPS to 2000RPS.

	100 RPS	200 RPS	300 RPS	400 RPS	500 RPS	600 RPS	700 RPS	800 RPS	900 RPS	1000 RPS
+INF	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)
2.0 *ms*	9 (0.1%)	13 (0.1%)	15 (0.0%)	15 (0.0%)	31 (0.1%)	33 (0.0%)	32 (0.0%)	43 (0.0%)	46 (0.0%)	50 (0.0%)
1.4 *ms*	29 (0.4%)	32 (0.2%)	73 (0.4%)	96 (0.4%)	106 (0.3%)	127 (0.3%)	149 (0.3%)	188 (0.3%)	221 (0.4%)	228 (0.3%)
1.3 *ms*	12 (0.2%)	35 (0.2%)	36 (0.2%)	53 (0.2%)	64 (0.2%)	69 (0.1%)	91 (0.2%)	98 (0.2%)	135 (0.2%)	153 (0.2%)
1.2 *ms*	21 (0.3%)	26 (0.2%)	55 (0.3%)	72 (0.3%)	84 (0.2%)	110 (0.3%)	134 (0.3%)	188 (0.3%)	158 (0.2%)	221 (0.3%)
1.1 *ms*	13 (0.2%)	42 (0.3%)	47 (0.2%)	87 (0.3%)	86 (0.2%)	148 (0.4%)	182 (0.4%)	193 (0.4%)	223 (0.4%)	209 (0.3%)
1.0 *ms*	15 (0.2%)	25 (0.2%)	47 (0.2%)	77 (0.3%)	94 (0.3%)	143 (0.3%)	187 (0.4%)	217 (0.4%)	239 (0.4%)	262 (0.4%)
900 *μs*	18 (0.3%)	27 (0.2%)	52 (0.2%)	46 (0.1%)	96 (0.3%)	141 (0.3%)	159 (0.3%)	177 (0.3%)	212 (0.3%)	271 (0.4%)
800 *μs*	184 (3.0%)	92 (0.7%)	261 (1.4%)	95 (0.3%)	131 (0.4%)	106 (0.2%)	126 (0.3%)	108 (0.2%)	187 (0.3%)	248 (0.4%)
700 *μs*	656 (10.9%)	433 (3.6%)	1889 (10.4%)	429 (1.7%)	379 (1.2%)	222 (0.6%)	241 (0.5%)	211 (0.4%)	328 (0.6%)	397 (0.6%)
600 *μs*	1678 (27.9%)	1968 (16.4%)	4973 (27.6%)	2334 (9.7%)	2450 (8.1%)	1766 (4.9%)	1751 (4.1%)	1851 (3.8%)	1998 (3.7%)	2013 (3.3%)
500 *μs*	2758 (45.9%)	7051 (58.7%)	8956 (49.7%)	16455 (68.5%)	22320 (74.4%)	25999 (72.2%)	29382 (69.9%)	33893 (70.6%)	37877 (70.1%)	40886 (68.1%)
400 *μs*	607 (10.1%)	2256 (18.8%)	1596 (8.8%)	4241 (17.6%)	4159 (13.8%)	7136 (19.8%)	9566 (22.7%)	10833 (22.5%)	12376 (22.9%)	15061 (25.1%)
300 *μs*	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	1 (0.0%)
MIN	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)

(a) Heatmap showing response times from Traefik Mesh and varying loads from 100PRS to 1000RPS.

	1100 RPS	1200 RPS	1300 RPS	1400 RPS	1500 RPS	1600 RPS	1700 RPS	1800 RPS	1900 RPS	2000 RPS
+INF	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)
2.0 *ms*	53 (0.0%)	1205 (1.6%)	70 (0.0%)	94 (0.1%)	78 (0.0%)	117 (0.1%)	117 (0.1%)	117 (0.1%)	125 (0.1%)	175 (0.1%)
1.4 *ms*	305 (0.4%)	407 (0.5%)	435 (0.5%)	495 (0.5%)	516 (0.5%)	600 (0.6%)	732 (0.7%)	793 (0.7%)	861 (0.7%)	959 (0.7%)
1.3 *ms*	180 (0.2%)	150 (0.2%)	193 (0.2%)	239 (0.2%)	271 (0.3%)	275 (0.2%)	237 (0.2%)	321 (0.2%)	402 (0.3%)	440 (0.3%)
1.2 *ms*	255 (0.3%)	218 (0.3%)	288 (0.3%)	292 (0.3%)	380 (0.4%)	387 (0.4%)	434 (0.4%)	446 (0.4%)	460 (0.4%)	587 (0.4%)
1.1 *ms*	307 (0.4%)	272 (0.3%)	362 (0.4%)	416 (0.4%)	463 (0.5%)	497 (0.5%)	554 (0.5%)	672 (0.6%)	701 (0.6%)	725 (0.6%)
1.0 *ms*	290 (0.4%)	260 (0.3%)	485 (0.6%)	540 (0.6%)	632 (0.7%)	670 (0.6%)	756 (0.7%)	823 (0.7%)	872 (0.7%)	1035 (0.8%)
900 *μs*	301 (0.4%)	316 (0.4%)	412 (0.5%)	586 (0.6%)	703 (0.7%)	734 (0.7%)	753 (0.7%)	886 (0.8%)	1008 (0.8%)	1054 (0.8%)
800 *μs*	312 (0.4%)	286 (0.3%)	493 (0.6%)	446 (0.5%)	500 (0.5%)	580 (0.6%)	700 (0.6%)	830 (0.7%)	951 (0.8%)	979 (0.8%)
700 *μs*	459 (0.6%)	405 (0.5%)	450 (0.5%)	428 (0.5%)	506 (0.5%)	471 (0.4%)	511 (0.5%)	722 (0.6%)	901 (0.7%)	1116 (0.9%)
600 *μs*	1898 (2.8%)	2881 (4.0%)	1844 (2.3%)	1836 (2.1%)	2217 (2.4%)	2227 (2.3%)	2371 (2.3%)	2612 (2.4%)	2466 (2.1%)	2659 (2.2%)
500 *μs*	46183 (69.9%)	51210 (71.1%)	52117 (66.8%)	55402 (65.9%)	57241 (63.6%)	61004 (63.5%)	64823 (63.5%)	67530 (62.5%)	70993 (62.2%)	77736 (64.7%)
400 *μs*	15457 (23.4%)	14390 (19.9%)	20851 (26.7%)	23227 (27.6%)	26492 (29.4%)	28437 (29.6%)	30012 (29.4%)	32249 (29.8%)	34260 (30.0%)	32535 (27.1%)
300 *μs*	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	1 (0.0%)	1 (0.0%)	1 (0.0%)	0 (0.0%)	1 (0.0%)	0 (0.0%)
MIN	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)

(b) Heatmap showing response times from Traefik Mesh and varying loads from 1100PRS to 2000RPS.

Table 2.9: Heatmaps of Traefik Mesh showing response buckets at loads from 100RPS to 2000RPS.

2.6 Resource usage

An important factor to consider when evaluating any technology is its cost, here we are specifically interested in its resource usage as that will be a driving factor for scaleability and operation costs. This test serves to measure the resource requirements of the service mesh technologies, in particular their proxies as that is what will scale the most with load.

2.6.1 Methodology

To depict a real world scenario the high availability environment described in Section 2.2.2 is utilized for this test, the target application and loadgenerator are also deployed in a high availability manner; Specifically a pod for each of *Emojivoto*'s components: Web, voting and emoji pods are run on each worker node, the same is true for the *Vegeta* loadgenerator described in Section 2.1.2; In total 4 instances of vegeta, web, emoji and voting alongside the service mesh being tested exists in the cluster. Each service mesh is individually installed, tested and cleaned up.

A test run starts with configuring each of the vegeta instances to generate a load of 200 RPS (thus a total of 800 RPS, since there are 4 replicas each sending 200 each) for 20 minutes towards the target endpoint described in Section 2.1.2. Prometheus collects resource usage information during the test from all containers including the sidecar proxies and Traefik's proxies, these are then displayed in Grafana. Once the CPU usage has stabilized (on average around 4 minutes into the run), the following data is recorded for each (sidecar-)proxy:

- Average CPU quote over 10 minutes.
- Highest CPU in the same 10 minute frame.
- Highest Ram usage.

2.6.2 Results

Traefik mesh

IT seems that Traefik Mesh distributed the load mainly between node 2 and node 4, while barely utilizing node 1. Node 2 was sporadically used. Its worth noting that Traefik Mesh had the lowest total resource usage of all the service meshes.

Table 2.10: Resource usage of Traefik Mesh

Proxy	Ram	CPU Max	CPU Avg
Node 1	18.23 MiB	0.175	0.15
Node 2	21.18 MiB	2.02	2.02
Node 3	19.77 MiB	0.846	0.38
Node 4	20.61 MiB	1.85	1.81
Total	79.79 MiB	4.89	4.36

Istio

Istio is clearly very memory hungry, having upwards of a GiB in total memory usage, 10 times more then the other meshes. It also does not seem to have done a great job load balancing with its default behaviour.

Table 2.11: Resource usage of Istio

Sidecar	Ram	CPU Max	CPU Avg
Web 1	63.75 MiB	0.617	0.613
Web 2	69.28 MiB	0.507	0.504
Web 3	68.72 MiB	0.519	0.507
Web 4	244.32 MiB	1.23	1.21
Voting 1	53.37 MiB	0.137	0.133
Voting 2	55.31 MiB	0.136	0.134
Voting 3	55.14 MiB	0.406	0.403
Voting 4	54.98 MiB	0.166	0.163
Emoji 1	57.27 MiB	0.312	0.306
Emoji 2	57.55 MiB	0.386	0.382
Emoji 3	68.03 MiB	0.984	0.944
Emoji 4	60.04 MiB	0.304	0.300
Vegeta 1	43.32 MiB	0.120	0.117
Vegeta 2	43.01 MiB	0.144	0.141
Vegeta 3	45.45 MiB	0.350	0.345
Vegeta 4	44.47 MiB	0.126	0.120
Total:	1084 MiB	6.444	6.322

Linkerd

Linkerd seems to have done the best when it comes to balancing the load across the services, especially with the web service, it used barely over 100MiB in total of Ram, and had very consistent CPU utilization.

Table 2.12: Resource usage of Linkerd

Sidecar	**Ram**	**CPU Max**	**CPU Avg**
Web 1	9.41 MiB	0.725	0.706
Web 2	9.27 MiB	0.723	0.710
Web 3	9.01 MiB	0.700	0.696
Web 4	9.28 MiB	0.688	0.685
Voting 1	4.73 MiB	0.137	0.130
Voting 2	4.91 MiB	0.0978	0.0955
Voting 3	4.83 MiB	0.128	0.120
Voting 4	4.95 MiB	0.122	0.119
Emoji 1	4.95 MiB	0.501	0.483
Emoji 2	4.99 MiB	0.485	0.481
Emoji 3	4.80 MiB	0.559	0.545
Emoji 4	4.91 MiB	0.461	0.448
Vegeta 1	8.39 MiB	0.271	0.266
Vegeta 2	7.41 MiB	0.0981	0.0951
Vegeta 3	7.50 MiB	0.0963	0.0940
Vegeta 4	7.71 MiB	0.0867	0.0852
Total	107.05 MiB	5.879	5.759

Chapter 3

Security

In this chapter i will discuss how certain security features are implemented in the service meshes, and how they are configured. A common trend that will be seen is that istio has its own Custom Resource Definition (CRD) for security called *security.istio.io* while Traefik Mesh and Linkerd 2 both mainly use the Service Mesh Interface (SMI) specification to configure security options.

3.1 Mutual TLS

Using mutual TLS (mTLS) is an essential step of zero-trust networking. It allows both sides of a communication tunnel to authenticate each other, and therefore mTLS is sometimes referred to as mutual authentication. When we talk about mTLS within the scope of service meshes, we generally mean automatic mTLS. The service mesh automatically upgrades plaintext traffic to tls and automatically mTLS to authenticate the services as they talk together.

We might want to force communication to require mTLS, and this is referred to as mTLS enforcement.

Generally, we want to use mTLS as much as possible; however, it does introduce a delay and an inevitable resource cost. Thus it might be advantageous in very low resource/latency environments/applications to turn mTLS by default off. There are also cases where mTLS would be excessive; for example, it generally does not make sense to have mTLS on port 443 of a web server that serves traffic over HTTPS already, particularly for ingress traffic from the outside world.

3.1.1 Traefik Mesh

Traefik mesh does not at the time of writing support mutual TLS [26]. However, as of version 1.4, Traefik Mesh uses stock Traefik Proxy as its underlying proxy [11], Traefik Proxy does support mTLS [27] so that would be a possible route to go if mTLS is needed.

Please note that as Traefik proxy is outside the scope of this thesis, I have not explored this possible route but felt it was still worth a mention.

3.1.2 Istio

Istio lets you control mTLS through its **PeerAuthentication** resources, which defines what traffic is allowed/denied from being tunneled through the sidecar [28]. As mTLS is implemented in the sidecar this allows you to specify the mTLS level and requirement of your workloads, the following 4 modes of mTLS is supported:

1. **UNSET**: Mode is inherited from parent.
2. **DISABLE**: Disables mTLS and tunneling.
3. **PERMISSIVE**: (Effective default [29]) Opertunistic mTLS with no enforcement.
4. **STRICT**: mTLS enforcement, connection must be TLS with a valid client cert present.

Istio has an impressive scope feature for this. The peer authentication resource is only applied to workloads within the namespace the **PeerAuthentication** resource is in, with one important and helpful exception: *if applied to the istio-root namespace, it applies globally* [28].

If a *WorkloadSelector* is specified the policy is only applied to matching workloads, if no workload selector is used, the policy applies to all workloads within the namespace. Istio even allows you to set a mode of operation on a port level using optional the *portLevelMtls* field.

Istio, by default uses an autogenerated self-signed root cert; you can configure Istio to use any specific cert or to use a secret store such as Hashicorp Vault [30].

3.1.3 Linkerd 2

Linkerd 2 supports automatic mTLS through its identity component [31]. Currently, mTLS cannot be turned off globally without turning the identity component off as well, and this also disables features such as tap, which rely on identity [32].

There are some inject flags supported by Linkerd 2 that allow you to manage what traffic is routed through the proxy which in turn manages what traffic has identity (and thus eligible for mTLS) [33] [34], please see the following table 3.1.

Table 3.1: Arguments to control Linkerd 2 proxy

Inject cmd arg	Annoatation	Description
–skip-inbound-ports	config.linkerd.io/skip-inbound-ports	Skip the proxy on the specified ports when trafic is inbound, and send directly to application.
–skip-outbound-ports	config.linkerd.io/skip-outbound-ports	Skip the proxy on the specified ports when traffic is outbound.
–require-identity-on-inbound-ports	config.linkerd.io/proxy-require-identity-inbound-ports	Require inbound traffic on the specified ports to have an valid identity.

There is no official mTLS enforcement in Linkerd 2 [35], requiring identity on an inbound port is the closest you will get to mTLS enforcement.

Linkerd 2, like Istio generates a cert by default and supports bringing your own certificates. The automatic certificate will expire after 365 days and require manual reissuing [36], alternatively automatic control-plane certificate rotation can be configured [37] using third party services such as Vault or cert-manager. The only requirement for a third-party solution to function is the ability to write to Kubernetes secrets of type kubernetes.io/tls [38].

3.2 Access Control

We often want to limit access to certain parts of our infrastructure to only a select few identities, this is where access control comes in handy. This section covers what scoping features and actions are supported by Access Control List (ACL) in the different service meshes.

3.2.1 Istio

Istio uses the **AuthorizationPolicy** resource to define access control between workloads [39], the scoping support is identical to that of **PeerAuthentication** described in 3.1.2, except for the fact there is no per port override. We now instead have rules, which allows us to within the same policy define several mini scopes depending on the source, destination and conditionals. Its worth noting that UDP traffic will bypass the proxy and therefore any **AuthorizationPolicy** [40]

Requests are matched based on the rules in the policy; the matching is OR-based, meaning that a request will ignore the other rules once matched to one. If no rules are specified, the policy applies to all requests within the scope. All fields of the rules are optional, and all accept multiple values for sources, destinations and conditionals. Strings in rules support a form of regex, for example "/api/*/hello" will match "/api/1/hello" and "/api/2/hello" but not "/api/hello".

There are four supported actions Allow (default) and Deny actions define whether to allow or deny matched requests, respectively. They also inversely define the behaviour of non matched requests within the same scope; an Authorization-Policy with its action set to Deny will implicitly allow non matched requests. The AUDIT action marks matched requests to be logged. It does not have any effect on whether requests are allowed or not. The AUDIT action requires a supported plugin to be installed, and at the time of writing, the only supported plugin is the StackDriver plugin [39].

Lastly, the **CUSTOM** action allows extensions to evaluate the matched requests, and they are evaluated before the native actions Allow and Deny.

You can specify source on a granular level There are five optional fields that positively match requests; based on the criteria described in the table 3.2. There is also an accompanying negative field for each positive field(prefixed by not) that disallow matched sources (i.e. negative field of IpBlocks is notIpBlocks).

Table 3.2: Optional positive match fields for source

Field name	Matches to	Description
principals	source.principal attribute	List of allowed identities such as service accounts.
requestPrincipals	request.auth.principal attribute	A list of allowed request identities i.e "iss/sub" claims.
namespaces	source.namespace attribute	List of Kubernetes namespaces the request is allowed to originate from.
ipBlocks	source.ip attribute	List of allowed IP addresses and CIDR notated IP Address blocks for the request to originate from.
remoteIpBlocks	X-Forwarded-For header or proxy protocol	Same as ipBlocks but using X-Forwarded-For header or similar for other proxy protocols, usually this is for remote end users.

Within fields, matches are OR-Based, while the source requires at least one match per specified positive field. Meaning for a request to match, it must match at least one of the specified values in each field, specified in the source and none of the negative fields. Fields not specified its ignored during evaluation.

Destinations can also be specified on a granular level they function identity to sources with how they match requests and how they have a negative field per positive fields, the only difference is the fields they have as seen in table 3.3

Table 3.3: Optional positive match fields for destination

Field name	Matches to	Description
hosts	request.host attribute	HTTP only, the target host / domain.
ports	destination.port attribute	Allowed destination ports for the request.
methods	request.method attribute	HTTP only, the allowed REST request methods, i.e GET, POST, ect.
paths	request.url_path	For HTTP its the path url segement, for gRPC its the fully qualified domain of the method.

Conditional matching is supported conditionals have a required field **key** which specifies where to get the value from; it then matches against a set of allowed or disallowed values. For example, you can specify to the request needs a valid JWT token issued by a vendor of choice.

3.2.2 LinkerD

LinkerD 2 has authorization policies planned for version 2.11 [41]. The current version of LinkerD at time of writing is 2.10; thus, I could not verify LinkerD 2's authorization capabilities.

3.2.3 Traefik Mesh

Traefik Mesh provides access control as an opt-in feature, meaning access control is not supported by default [42]. To opt-in one has to set the ACL flag to true during installation [42], enabling what Traefik refferres to as ACL mode [43]. When enabled all traffic through the service mesh is implicitly denied [44], and support for version v1alpha2 [45] of the Traffic Access Control(access.smi-spec.io) SMI component is enabled [43]; It contains a single CRD called TrafficTarget, which associates a set of rules to a destination and set of sources excplicitly allowing matching traffic. TrafficTarget resource requires the destination, atleast one rule and atleast one source to be set [46].

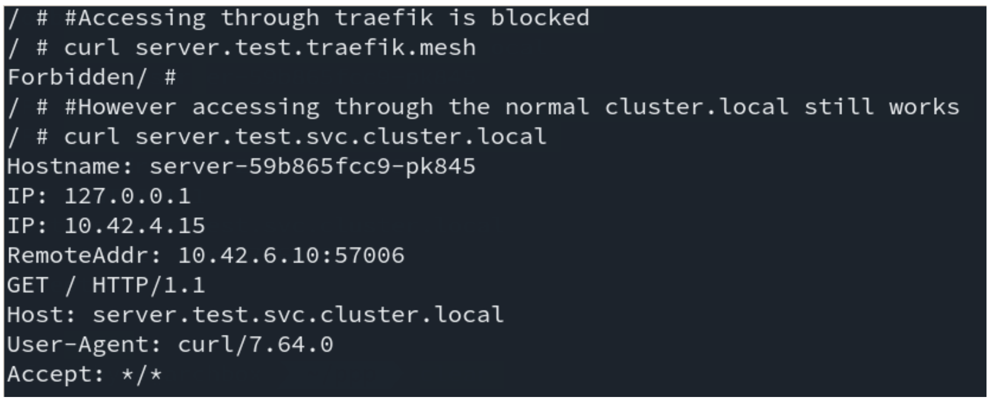
The destination and sources are service identities currently the only way to provision service identities are through Kubernetes ServiceAccounts, the SMI specification is planning to support other ways to provision service identities at a later date.

Rules are evaluated in an AND-based relationship and are defined through the Traffic Spec(specs.smi-spec.io) SMI component, at the time of writing Traffic Mesh supports version v1alpha3 of this component. This version has two CRDs called HTTPRouteGroup and TCPRoute, for defining HTTP and TCP traffic respectively. A HTTPRouteGroup consists of one or multiple routes, each route can define a name, a set of allowed REST methods, a set of regexes for allowed Headers and a regex for its path. When a HTTPRouteGroup is used as a rule in a TrafficTarget all routes defined in the group are allowed in a OR-Based relationship, meaning traffic must only match one of the routes. It is possible to select which routes must match by specifying the optional matches field.

Note that UDP traffic was only first supported in version v1alpha4 of the Traffic Spec component; Thus its not possible to use UDP within the mesh as its not possible to write a rule allowing it.

Its possible to specify which port the rules should apply to through the optional field **ports** on the destination. By default rules apply to all ports and protocols of the destination. This means that to for example allow only traffic on port 443 and 80, two TrafficTarget resources are required. This issue will likely be resolved once Traefik Mesh supports v1alpha4 of the Traffic Spec component, as it allows specify ports for TCPRoutes and UDPRoutes.

Non-meshed traffic is allowed to meshed services due to its non-invasive design, traffic that does not opt-in to go through the proxy is not effected by access control as seen in Figure 3.1. It is therefore important to disallow non-meshed traffic through other means for security critical pods.



```
/ # #Accessing through traefik is blocked
/ # curl server.test.traefik.mesh
Forbidden/ #
/ # #However accessing through the normal cluster.local still works
/ # curl server.test.svc.cluster.local
Hostname: server-59b865fcc9-pk845
IP: 127.0.0.1
IP: 10.42.4.15
RemoteAddr: 10.42.6.10:57006
GET / HTTP/1.1
Host: server.test.svc.cluster.local
User-Agent: curl/7.64.0
Accept: */*
```

Figure 3.1: Figure showing a meshed service being able to bypass Traefik mesh's access control by using the normal cluster.local DNS.

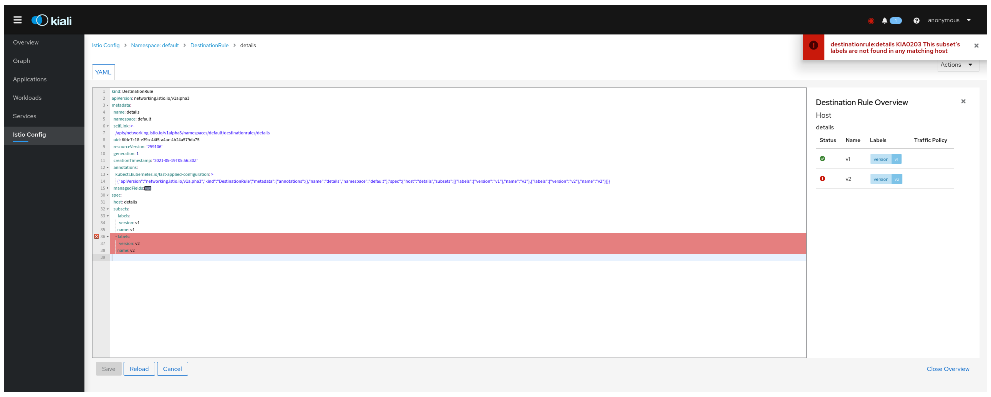
3.3 Monitoring

As mentioned in Section 2.1.3 all of the service meshes either depend on or provide their own Prometheus by default. They also all virtually provide the same "golden" metrics meaning its little point in comparing the metrics recorded. Therefore this section instead serves to briefly describe the various dashboards and documentation surrounding their monitoring capabilities.

3.3.1 Istio

Note that as Grafana technically belongs to its dependency *rancher-monitoring* it wont be talked about in this section. By default Rancher's Istio 1.9.3 chart only comes with the *Kali* dashboard, however *Jaeger* can also be optionally enabled in the options of the chart.

The Kali dashboard allows you to not only view what services are meshed, but also the relationship between them. It also allows you to view your configuration and will point out configuration errors and suggestions to your as seen in Figure 3.2. It can also generate a diagram of your traffic flow, Figure 3.3 depicts such a diagram generated when sending a load to the target /api/leaderboard endpoint as described in Section 2.1.2.



destinationrule:details KIA0203 This subset's labels are not found in any matching host

```
kind: DestinationRule
apiVersion: networking.istio.io/v1alpha3
metadata:
 name: details
 namespace: default
 selfLink: >-
  /apis/networking.istio.io/v1alpha3/namespaces/default/destinationrules/details
 uid: 6fde7c18-e39a-44f5-a4ac-4b24a579da75
 resourceVersion: '259106'
 generation: 1
 creationTimestamp: '2021-05-19T05:56:30Z'
 annotations:
  kubectl.kubernetes.io/last-applied-configuration: >
   {"apiVersion":"networking.istio.io/v1alpha3","kind":"DestinationRule","metadata":{"annotations":{},"name":"details","namespace":"default"},"spec":{"host":"details","subsets":[{"labels":{"version":"v1"},"name":"v1"},{"labels":{"version":"v2"},"name":"v2"}]}}
 managedFields:
spec:
 host: details
 subsets:
  - labels:
    version: v1
   name: v1
  - labels:
    version: v2
   name: v2
```

Destination Rule Overview

Host

details

Status	Name	Labels	Traffic Policy
✓	v1	version v1	
!	v2	version v2	

Figure 3.2: Kali dashboard highlighting a configuration issue in a Destination-Route config.

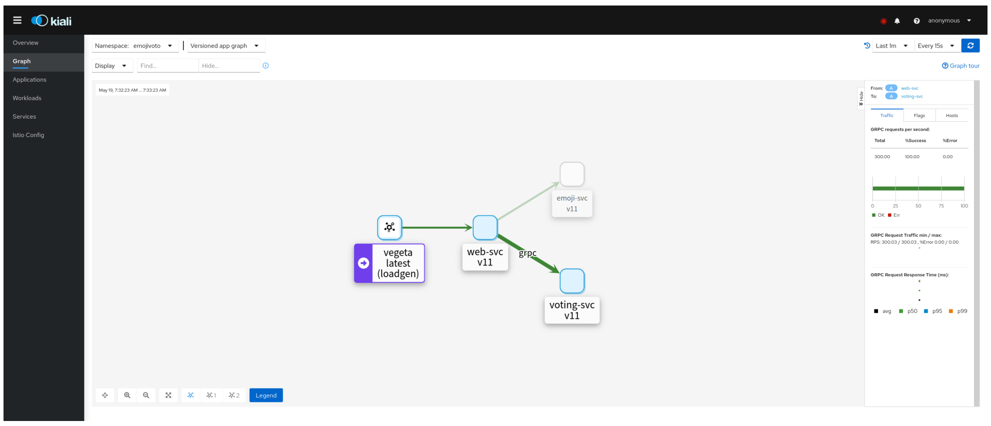


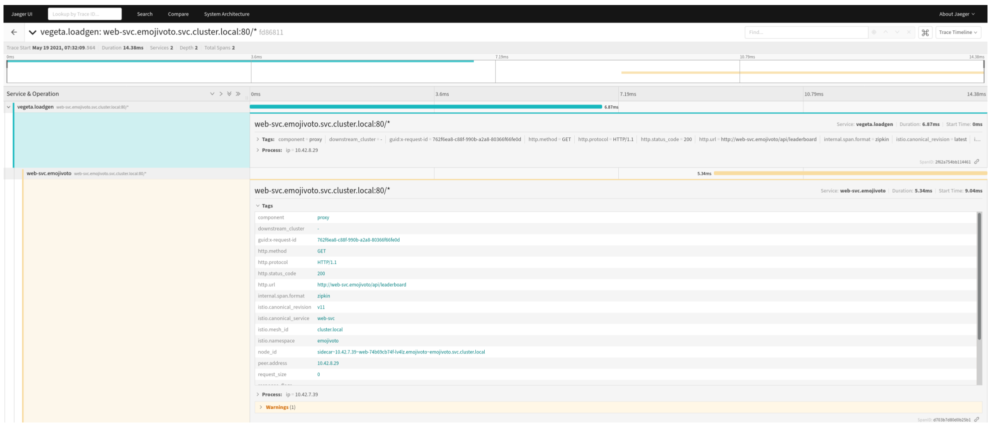
Figure 3.3: Kali dashboard showing traffic flow generated by /api/leaderboard.

Access logs

Envoy can generate access logs for each request, or selectively through a AUDIT action rule as described earlier in Section 3.2.1. There is however no central location by to access these logs, by default they are not sent anywhere. The format of the access logs can be customized [47], and Istio has an tutorial on configuring Envoy to log to stdout, in their documentation.[1].

[1]`https://istio.io/latest/docs/tasks/observability/logs/access-log/`

Jaeger allows you to access the distributed traces recorded by Envoy, these are far more detailed than Kali's traffic flow graph, an example of a trace can be seen in Figure 3.4. In addition to Jaeger Istio also supports: Zipkin, Lightstep, and Datadog, as tracing backends. [48]



vegeta.loadgen: web-svc.emojivoto.svc.cluster.local:80/* fd86811

Trace Start **May 19 2021, 07:32:09**.564 | Duration **14.38ms** | Services **2** | Depth **2** | Total Spans **2**

vegeta.loadgen web-svc.emojivoto.svc.cluster.local:80/*

web-svc.emojivoto.svc.cluster.local:80/*

Service: **vegeta.loadgen** | Duration: **6.87ms** | Start Time: **0ms**

Tags: component = proxy | downstream_cluster = - | guid:x-request-id = 762f6ea8-c88f-990b-a2a8-80366f66fe0d | http.method = GET | http.protocol = HTTP/1.1 | http.status_code = 200 | http.url = http://web-svc.emojivoto/api/leaderboard | internal.span.format = zipkin | istio.canonical_revision = latest | i...

Process: ip = 10.42.8.29

SpanID: 2f62a754bb114461

web-svc.emojivoto web-svc.emojivoto.svc.cluster.local:80/*

web-svc.emojivoto.svc.cluster.local:80/*

Service: **web-svc.emojivoto** | Duration: **5.34ms** | Start Time: **9.04ms**

Tags

component	proxy
downstream_cluster	-
guid:x-request-id	762f6ea8-c88f-990b-a2a8-80366f66fe0d
http.method	GET
http.protocol	HTTP/1.1
http.status_code	200
http.url	http://web-svc.emojivoto/api/leaderboard
internal.span.format	zipkin
istio.canonical_revision	v11
istio.canonical_service	web-svc
istio.mesh_id	cluster.local
istio.namespace	emojivoto
node_id	sidecar~10.42.7.39~web-74b69cb74f-lv4lz.emojivoto~emojivoto.svc.cluster.local
peer.address	10.42.8.29
request_size	0

Process: ip = 10.42.7.39

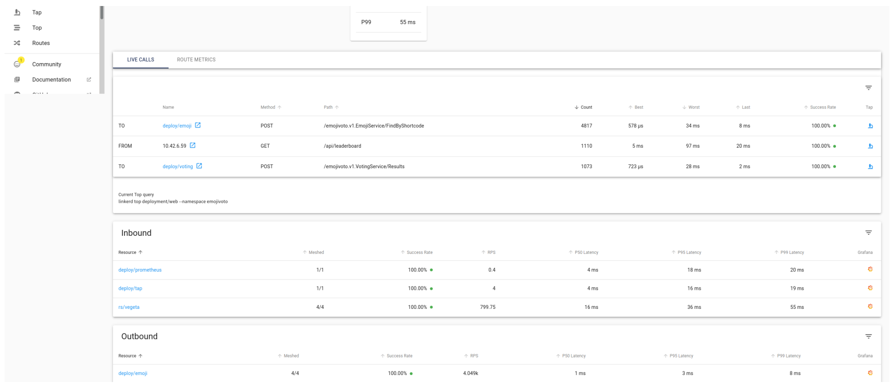
Warnings (1)

SpanID: d703b7d80d0b25b1

Figure 3.4: Jaeger showing an trace.

3.3.2 Linkerd

Linkerd has split its monitoring and telemetry features into the Viz extension; It not only contains Grafana and Prometheus, but also Linkerd Viz's dashboard and tap [49]. Note that by default the Prometheus instance only stores metrics for 6 hours [50], though this is configurable. Linkerd can also be configured to use rancher-monitoring's Grafana and Prometheus instance as stated earlier in Section 2.1.3.



P99	55 ms

	Name	Method	Path	Count	Best	Worst	Last	Success Rate	Tap
TO	deploy/emoji	POST	/emojivoto.v1.EmojiService/FindByShortcode	4817	578 µs	34 ms	8 ms	100.00% ●	
FROM	10.42.6.59	GET	/api/leaderboard	1110	5 ms	97 ms	20 ms	100.00% ●	
TO	deploy/voting	POST	/emojivoto.v1.VotingService/Results	1073	723 µs	28 ms	2 ms	100.00% ●	

Current Top query
linkerd top deployment/web --namespace emojivoto

Inbound

Resource	Meshed	Success Rate	RPS	P50 Latency	P95 Latency	P99 Latency	Grafana
deploy/prometheus	1/1	100.00% ●	0.4	4 ms	18 ms	20 ms	
deploy/tap	1/1	100.00% ●	4	4 ms	16 ms	19 ms	
rs/vegeta	4/4	100.00% ●	799.75	16 ms	36 ms	55 ms	

Outbound

Resource	Meshed	Success Rate	RPS	P50 Latency	P95 Latency	P99 Latency	Grafana
deploy/emoji	4/4	100.00% ●	4.049k	1 ms	3 ms	8 ms	

Figure 3.5: Linkerd dashboard displaying live metrics for the emojivoto web deployment.

The dashboard gives you an overview over all services in your cluster, including none meshed ones. Clicking on a meshed service allows you to see live calls to it, both globally for the service and per route; an example of this is seen in Figure 3.5. The dashboard can also generate a graph over the traffic flow in the mesh, as seen in Figure 3.6.

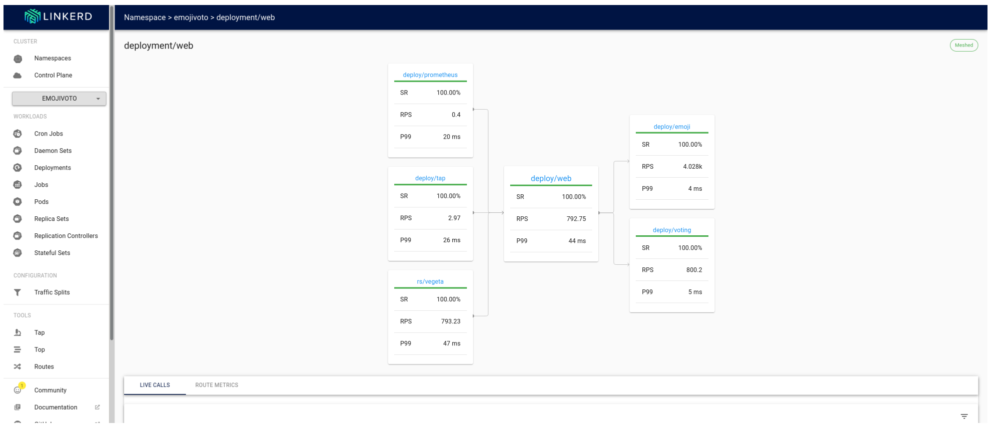
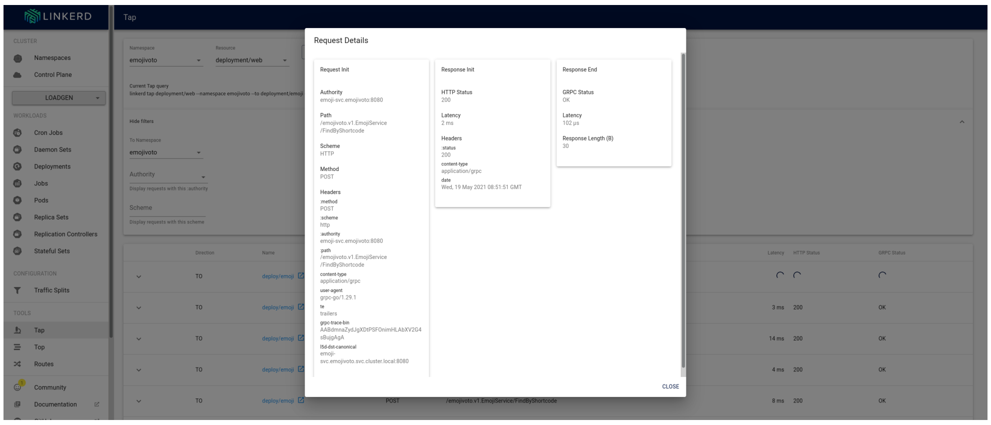


Figure 3.6: Linkerd dashboard showing a diagram over the network flow to and from the web deployment.

Tap allows you to tap into the proxy to see live traffic flowing through it, not only from the dashboard but also through the *linkerdctl* command line tool [51], the dashboard version is depicted in Figure 3.7.


Request Details

Request Init

Authority
emoji-svc.emojivoto:8080

Path
/emojivoto.v1.EmojiService
/FindByShortcode

Scheme
HTTP

Method
POST

Headers
:method
POST
:scheme
http
:authority
emoji-svc.emojivoto:8080
:path
/emojivoto.v1.EmojiService
/FindByShortcode
content-type
application/grpc
user-agent
grpc-go/1.29.1
te
trailers
grpc-trace-bin
AABdmnaZydJgXDtPSFOnimHLAbXV2G4
sBujgAgA
l5d-dst-canonical
emoji-
svc.emojivoto.svc.cluster.local:8080

Response Init

HTTP Status
200

Latency
2 ms

Headers
:status
200
content-type
application/grpc
date
Wed, 19 May 2021 08:51:51 GMT

Response End

GRPC Status
OK

Latency
102 µs

Response Length (B)
30

CLOSE

Figure 3.7: Linkerd dashboard showcasing the tap feature, showing information for a live call.

Distributed tracing can be enabled by installing the *linkerd-jaeger* extension, which consists of a Jaeger backend, a trace collector and a Jaeger injector, the latter being responsible for configuring the sidecar proxy to emit traces [52]. Linkerd can be configured to use your own Jaeger backend [53].

Linkerd also has a thorough tutorial[2] on how to enable, configure and access call tracing.

Access logs Linkerd allows configuring various log levels of the sidecar proxy [54]; The log level can either be specified globally applying to all proxies, or for a particular Linkerd rust module [55]. Five log levels are available, trace being a more traditional access log level [55].

3.3.3 Traefik Mesh

Traefik mesh comes with Grafana, Prometheus and Jaeger, the monitoring is implemented in Traefik Proxy. As mentioned in Section 2.1.3 Traefik Mesh can be configured to use Grafana and Prometheus from *rancher-monitoring*. The Grafana dashboard is depicted in Figure 3.8

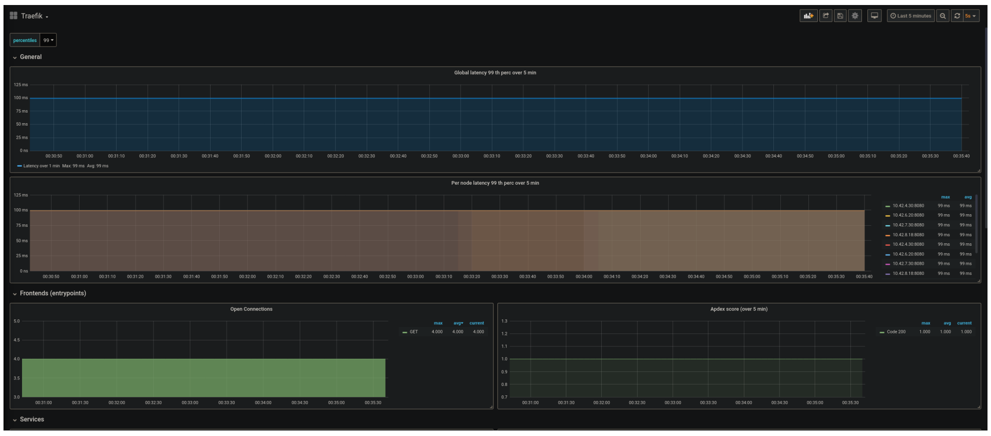

Traefik
Last 5 minutes
5s
percentiles
99
General
Global latency 99 th perc over 5 min
Latency over 1 min Max: 99 ms Avg: 99 ms
Per node latency 99 th perc over 5 min
max
avg
10.42.4.30:8080 99 ms 99 ms
10.42.6.20:8080 99 ms 99 ms
10.42.7.30:8080 99 ms 99 ms
10.42.8.18:8080 99 ms 99 ms
10.42.4.30:8080 99 ms 99 ms
10.42.6.20:8080 99 ms 99 ms
10.42.7.30:8080 99 ms 99 ms
10.42.8.18:8080 99 ms 99 ms
Frontends (entrypoints)
Open Connections
max avg current
GET 4.000 4.000 4.000
Apdex score (over 5 min)
max avg current
Code 200 1.000 1.000 1.000
Services

Figure 3.8: Traffic flow from Web pod when using Traefik Mesh.

Jaeger can be used to access network traces of meshed services, Traefik Mesh also has OpenTracing support, and alongside Jaeger also supports: Zipkin, Datadog, Instana, Haystack and Elastic as tracing backends [56].

[2]`https://linkerd.io/2.10/tasks/distributed-tracing/`

Access logs

Traefik Mesh supports access logs through Traefik proxy, the log level as well as the format of the logs can be formatted [43]. The logs aren't collected in a central location on a cluster scale, but since only one proxy exists per node, the logs belonging to a node are centralized in one pod. Traefik proxy supports a lot of filters for whether to log a request or not, however as its outside the scope of this thesis, they wont be covered here, for more info please see the Traefik Proxy documentation. [3]

[3]`https://doc.traefik.io/traefik/observability/access-logs/`

Chapter 4

Conclusion

To conclude I wanna summarize the strengths and weaknesses of each service mesh, and share how my experience was using them.

4.1 Istio's Strengths and Weaknesses

Istio was by far the most configurable technology that was looked at, particularly when it came to access control. Its security features are very robust and it has official rancher integration. It is also the most popular service mesh according to the CNCF as mentioned in the introduction.

However it also by far had the worst performance, both in terms of introduced latency and resource usage, particularly RAM, using an entire gibibyte for 800 RPS. Its configurability is also a double edged sword as I found it personally very easy to get overwhelmed within its documentation; It all hits you like a brick wall at first.

4.2 Traefik Mesh's Strengths and Weaknesses

Traefik Mesh ended up being the lightest on resource usage, which makes sense considering it only deploys one proxy per host instead of a sidecar alongside each pod. However by default it comes with a very low CPU limit, which needed to be increased for this thesis in order to complete any test above 300 RPS.

Traefik Mesh's most unique trait is that its non-invasive, this makes it more predictable which is nice, but also makes it lack auto inject capabilities and as demonstrated in Figure 3.1; makes it trivially easy to bypass the access control. Furthermore it itself does not have any mechanism to solve this issue, therefore it must be done via a 3rd party solution.

Another issue with the ACL mode is that since UDPRoutes aren't supported yet, thus UDP traffic doesn't work at all if ACL mode is enabled.

Another clear weakness of Traefik Mesh is its documentation. While it was the easiest to get started with, I quickly found its documentation lacking when it came to more advanced features; I learned more from reading issues on its Github repository then I did from its documentation.

4.3 Linkerd's Strengths and Weaknesses

Linkerd's command line tool was a great help getting started. It was in the middle of the pack when it came to performance, not to far behind Traefik Mesh, it also seemed to load balance quite well on its own. Its latencies are however closer to that of Istio then to Traefik due to it having more hops along the way for meshed traffic to go through.

An issue I encounted with Linkerd is that with kubernetes batch/jobs, the Linkerd sidecar proxy can fail to recognize the main container has finnished execution, and keeps the job from finnishing.

Another issue I encountered was that the main container would sometimes start sending requests before the sidecar proxy had fully initialized, causing the requests to be dropped. The final downside is that there is no mTLS enforcement yet.

Bibliography

[1] R. hat. (). 'Microservices: What is a service mesh?' [Online]. Available: https://www.redhat.com/en/topics/microservices/what-is-a-service-mesh. (accessed: 18.05.2021).

[2] Istio. (May 2017). 'Introducing istio,' [Online]. Available: https://istio.io/latest/news/releases/0.x/announcing-0.1/. (accessed: 17.05.2021).

[3] C. N. C. Foundation, 'Cloud native survey 2020,' p. 16, Nov. 2020. [Online]. Available: https://www.cncf.io/wp-content/uploads/2020/11/CNCF_Survey_Report_2020.pdf.

[4] Istio. (). 'What is istio?: Why use istio,' [Online]. Available: https://istio.io/latest/docs/concepts/what-is-istio/#why-use-istio. (accessed: 17.05.2021).

[5] Istio. (). 'Istio architecture: Envoy,' [Online]. Available: https://istio.io/latest/docs/ops/deployment/architecture/#envoy. (accessed: 17.05.2021).

[6] G. Dury. (). 'A kubernetes service mesh comparison: Linkerd review,' [Online]. Available: https://www.toptal.com/kubernetes/service-mesh-comparison#linkerd-review. (accessed: 17.05.2021).

[7] B. C. Gain. (Dec. 2018). 'Buoyant ceo on how linkerd sprang from twitter's heady early days,' [Online]. Available: https://thenewstack.io/buoyant-ceo-on-how-linkerd-sprang-from-twitters-heady-early-days/. (accessed: 17.05.2021).

[8] O. Gould. (Apr. 2017). 'Announcing linkerd 1.0,' [Online]. Available: https://linkerd.io/2017/04/25/announcing-linkerd-1-0/. (accessed: 17.05.2021).

[9] W. Morgan. (Dec. 2020). 'Why linkerd doesn't use envoy,' [Online]. Available: https://linkerd.io/2020/12/03/why-linkerd-doesnt-use-envoy/. (accessed: 17.05.2021).

[10] D. Duportal. (Sep. 2019). 'Announcing maesh, a lightweight and simpler service mesh made by the traefik team,' [Online]. Available: https://traefik.io/blog/announcing-maesh-a-lightweight-and-simpler-service-mesh-made-by-the-traefik-team-cb866edc6f29/. (accessed: 17.05.2021).

[11] M. Zapf. (Oct. 2020). 'Traefik mesh version 1.4, new name new features,' [Online]. Available: `https://traefik.io/blog/announcing-traefik-mesh-1-4-new-name-new-features`. (accessed: 15.04.2021).

[12] E. Vauge. (Sep. 2020). 'Traefik labs: Incubating the future of cloud native networking,' [Online]. Available: `https://traefik.io/blog/traefik-labs-incubating-the-future-of-cloud-native-networking/`. (accessed: 17.05.2021).

[13] Traefik. (). 'Traefik mesh: Non invasive service mesh,' [Online]. Available: `https://doc.traefik.io/traefik-mesh/#non-invasive-service-mesh`. (accessed: 17.05.2021).

[14] Traefik. (). 'Quickstart - traefik mesh: Using traefik mesh,' [Online]. Available: `https://doc.traefik.io/traefik-mesh/quickstart/#using-traefik-mesh`. (accessed: 17.05.2021).

[15] LinkerD. (). 'Overview: How it works,' [Online]. Available: `https://linkerd.io/2.10/overview/#how-it-works`. (accessed: 18.05.2021).

[16] Linkerd. (). 'Automatic proxy injection: Details,' [Online]. Available: `https://linkerd.io/2.10/features/proxy-injection/#details`. (accessed: 18.05.2021).

[17] Linkerd. (). 'Adding your service to linkerd: Meshing a service with annotations,' [Online]. Available: `https://linkerd.io/2.10/tasks/adding-your-service/#meshing-a-service-with-annotations`. (accessed: 18.05.2021).

[18] T. Fromm. (May 2019). 'Performance benchmark analysis of istio and linkerd,' [Online]. Available: `https://kinvolk.io/blog/2019/05/performance-benchmark-analysis-of-istio-and-linkerd/`. (accessed: 30.04.2021).

[19] T. Senart. (). 'Comment #642596847 on vegeta2's github issue #520.'

[20] Rancher. (). 'Rancher docs: Istio in rancher v2.3-v2.4: Architecture,' [Online]. Available: `https://rancher.com/docs/rancher/v2.x/en/istio/v2.3.x-v2.4.x#architecture`. (accessed: 19.05.2021).

[21] Linkerd. (). 'Bring your own prometheus,' [Online]. Available: `https://linkerd.io/2.10/tasks/external-prometheus/`. (accessed: 19.05.2021).

[22] T. M. D. Team. (). 'Traefik mesh chart's source code,' [Online]. Available: `https://github.com/traefik/mesh-helm-chart/blob/2542b19a6bfb17ab833e45dcb595db04a3e8` `mesh/charts/metrics/templates/prometheus.yaml#L40`. (accessed: 19.05.2021).

[23] Linkerd. (). 'Exporting metrics,' [Online]. Available: `https://Linkerd.io/2.10/tasks/exporting-metrics/`. (accessed: 19.05.2021).

[24] Rancher. (). 'Rancher docs: Architecture recommendations,' [Online]. Available: `https://rancher.com/docs/rancher/v2.x/en/overview/architecture-recommendations/#contrasting-rke-cluster-architecture-for-rancher-server-and-for-downstream-kubernetes-clusters`. (accessed: 8.05.2021).

[25] B. Schwartz. (Nov. 2016). 'Why percentiles don't work the way you think,' [Online]. Available: https://orangematter.solarwinds.com/2016/11/18/why-percentiles-dont-work-the-way-you-think/. (accessed: 18.05.2021).

[26] Traefik. (). 'Traefik mesh docs: Scheme limitations,' [Online]. Available: https://doc.traefik.io/traefik-mesh/configuration/#scheme. (accessed: 15.04.2021).

[27] M. Zapf. (Jan. 2021). 'Traefik proxy version 2.4 announcement,' [Online]. Available: https://traefik.io/blog/announcing-traefik-2-4. (accessed: 15.04.2021).

[28] Istio. (). 'Istio docs: Peerauthentication,' [Online]. Available: https://istio.io/latest/docs/reference/config/security/peer_authentication. (accessed: 19.04.2021).

[29] Istio. (). 'Istio docs: Globally enabling istio mutual tls in strict mode,' [Online]. Available: https://istio.io/latest/docs/tasks/security/authentication/authn-policy/#globally-enabling-istio-mutual-tls-in-strict-mode. (accessed: 19.04.2021).

[30] Istio. (). 'Istio docs: Plug in ca certificates,' [Online]. Available: https://istio.io/latest/docs/tasks/security/cert-management/plugin-ca-cert/#plug-in-certificates-and-key-into-the-cluster. (accessed: 19.04.2021).

[31] Linkerd. (). 'Linkerd2 documentation: Automatic mutual tls, how does it work,' [Online]. Available: https://linkerd.io/2.10/features/automatic-mtls/#how-does-it-work. (accessed: 20.04.2021).

[32] T. Rampelberg. (Dec. 2019). 'Comment #563402306 on linkerd2's github issue #2783,' [Online]. Available: https://github.com/linkerd/linkerd2/issues/2783#issuecomment-563402306. (accessed: 20.04.2021).

[33] Linkerd. (). 'Linkerd2 documentation: Inject,' [Online]. Available: https://linkerd.io/2.10/reference/cli/inject. (accessed: 20.04.2021).

[34] Linkerd. (). 'Linkerd2 documentation: Proxy configuration,' [Online]. Available: https://linkerd.io/2.10/reference/proxy-configuration/. (accessed: 20.04.2021).

[35] Linkerd. (). 'Linkerd2 documentation: Automatic mutual tls, caveats,' [Online]. Available: https://linkerd.io/2.10/features/automatic-mtls/#caveats-and-future-work. (accessed: 20.04.2021).

[36] Linkerd. (). 'Linkerd2 documentation: Automatic mutual tls, maintaince,' [Online]. Available: https://linkerd.io/2.10/features/automatic-mtls/#maintenance. (accessed: 20.04.2021).

[37] Linkerd. (). 'Linkerd2 documentation: Automatically rotating control plane tls credentials,' [Online]. Available: `https://linkerd.io/2.10/tasks/automatically-rotating-control-plane-tls-credentials`. (accessed: 20.04.2021).

[38] Linkerd. (). 'Linkerd2 documentation: Automatically rotating control plane tls credentials, third party cert management solutions,' [Online]. Available: `https://linkerd.io/2.10/tasks/automatically-rotating-control-plane-tls-credentials/#third-party-cert-management-solutions`. (accessed: 20.04.2021).

[39] Istio. (). 'Istio docs: Authorizationpolicy,' [Online]. Available: `https://istio.io/latest/docs/reference/config/security/authorization-policy`. (accessed: 20.04.2021).

[40] Istio. (). 'Istio docs: Protocol selection,' [Online]. Available: `https://istio.io/latest/docs/ops/configuration/traffic-management/protocol-selection/`. (accessed: 26.04.2021).

[41] O. Gould. (Dec. 2020). 'Comment #749688453 on linkerd2's github issue #3342,' [Online]. Available: `https://github.com/linkerd/linkerd2/issues/3342#issuecomment-749688453`. (accessed: 21.04.2021).

[42] Traefik. (). 'Traefik mesh docs: Installation: Access control lists,' [Online]. Available: `https://doc.traefik.io/traefik-mesh/install/#access-control-list`. (accessed: 30.04.2021).

[43] Traefik. (). 'Traefik mesh docs: Configuration: Static configuration,' [Online]. Available: `https://doc.traefik.io/traefik-mesh/configuration/#static-configuration`. (accessed: 30.04.2021).

[44] Traefik. (). 'Traefik mesh docs: Configuration: Access control,' [Online]. Available: `https://doc.traefik.io/traefik-mesh/configuration/#access-control`. (accessed: 30.04.2021).

[45] Traefik. (). 'Traefik mesh docs: Compatibility - smi specification support,' [Online]. Available: `https://doc.traefik.io/traefik-mesh/compatibility/#smi-specification-support`. (accessed: 30.04.2021).

[46] S. M. Interface. (). 'Service mesh interface specification: Access v1alpha2,' [Online]. Available: `https://github.com/servicemeshinterface/smi-spec/blob/d17d048f88fc8e9bbd2069c6f5187f12eeb54fbb/apis/traffic-access/v1alpha2/traffic-access.md`. (accessed: 30.04.2021).

[47] Istio. (). 'Getting envoy's access logs: Default access log format,' [Online]. Available: `https://istio.io/latest/docs/tasks/observability/logs/access-log/#default-access-log-format`. (accessed: 19.05.2021).

[48] Istio. (). 'Observability: Distributed traces,' [Online]. Available: `https://istio.io/latest/docs/concepts/observability/#distributed-traces`. (accessed: 19.05.2021).

[49] Linkerd. (). 'Telemetry and monitoring,' [Online]. Available: `https://linkerd.io/2.10/features/telemetry/`. (accessed: 19.05.2021).

[50] Linkerd. (). 'Telemetry and monitoring: Lifespan of linkerd metrics,' [Online]. Available: `https://linkerd.io/2.10/features/telemetry/#lifespan-of-linkerd-metrics`. (accessed: 19.05.2021).

[51] Linkerd. (). 'Tap,' [Online]. Available: `https://linkerd.io/2.9/reference/cli/tap`. (accessed: 19.05.2021).

[52] Linkerd. (). 'Distributed tracing with linkerd: Install the linkerd-jaeger extension,' [Online]. Available: `https://linkerd.io/2.10/tasks/distributed-tracing/#install-the-linkerd-jaeger-extension`. (accessed: 19.05.2021).

[53] Linkerd. (). 'Distributed tracing with linkerd: Bring your own jaeger,' [Online]. Available: `https://linkerd.io/2.10/tasks/distributed-tracing/#bring-your-own-jaeger`. (accessed: 19.05.2021).

[54] Linkerd. (). 'Modifying the proxy log level,' [Online]. Available: `https://linkerd.io/2.10/tasks/modifying-proxy-log-level/`. (accessed: 19.05.2021).

[55] Linkerd. (). 'Proxy log level,' [Online]. Available: `https://linkerd.io/2.10/reference/proxy-log-level/`. (accessed: 19.05.2021).

[56] Traefik. (). 'Tracing: Overview,' [Online]. Available: `https://doc.traefik.io/traefik/observability/tracing/overview/`. (accessed: 19.05.2021).