An Evaluation of Service Mesh Frameworks for Edge Systems

Yehia Elkhatib
Jose Povedano Poyato
School of Computing Science, University of Glasgow
Glasgow, UK

ABSTRACT

Service Mesh Technologies (SMTs) are increasingly popular in simplifying the networking between microservices. They allow one to declaratively and programmatically define service-to-service policies and interactions, and take all sorts of network management logic (*e.g.,* traffic splitting, request tracing, security, reliability) out of the application. This simplifies the development of microservice architectures, which are widely used in cloud and edge applications. However, the suitability for different SMTs for use in edge applications is unclear. Thus, this work compares the two most popular SMTs (Istio and Linkerd) in terms of performance and overhead for resource-constrained devices. Through extensive experimentation and comparing with a baseline of standard networking in a Kubernetes cluster, we identify that Linkerd offers a more edge-friendly SMT option in contrast to Istio. Overall, Istio's communications are ≈10% slower than Linkerd at an increased 1.2–1.4x more memory and ≈1.2x more CPU utilization.

CCS CONCEPTS

• **Computer systems organization** → **Cloud computing**; **n-tier architectures**; • **General and reference** → *Empirical studies.*

KEYWORDS

Service Mesh, Edge Computing, IoT Networking, System Performance and Measurement

ACM Reference Format:
Yehia Elkhatib and Jose Povedano Poyato. 2023. An Evaluation of Service Mesh Frameworks for Edge Systems. In *6th International Workshop on Edge Systems, Analytics and Networking (EdgeSys '23), May 8, 2023, Rome, Italy.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3578354.3592867

1 INTRODUCTION

The microservice architectural pattern has gained significant adoption for applications in the cloud and the edge [8, 17]. This is due to the number of benefits it offers, such as a high degree of technological heterogeneity, independent service development and maintenance timelines, better fault isolation, and individual scaling of services either horizontally or vertically.

However, deploying applications using such n-tiered architecture requires additional services for communication and management. Therefore, service mesh technologies (SMTs) are increasingly popular as they provide means to control service-to-service communication over the network. They function independently from the microservices (*i.e.,* outside the core business logic containers), abstracting network features from the service development and also providing useful management of the control plane.

There are a number of SMTs that are available as open source solutions. In this work, we aim to identify the suitability of SMT solutions for edge systems and in resource-constrained environments such as many Internet of Things (IoT) systems. For this purpose, we developed a number of experiments to evaluate Istio and Linkerd, as the most widely used SMTs.[1]

We set up controlled experiments to assess the performance and overhead of Istio and Linkerd for different scales of a microservice application deployed using Kubernetes. We compare the results of using each SMT against a baseline of native communication between Kubernetes pods. We conduct our experiments using a state-of-the-art edge server. To our knowledge, this is the first paper to provide a quantitative and empirical comparison of SMTs at scale for the purposes of edge applications.

We begin by explaining the microservice architecture and why it motivates the use of SMTs (Section 2), and reviewing related work (Section 3). We introduce the SMT solutions that we will evaluate (Section 4). Then we describe our setup for evaluation (Section 5), and present and discuss our results (Section 6).

2 BACKGROUND

2.1 The microservice architecture

Recent years have seen great rise in the popularity of microservice architecture applications. It was reported in 2021 that top industry performers run microservice architectures in 95.5% of their applications [12]. This popularity also extends to edge and IoT systems due to their inherent n-tiered architectures, and the relative ease in migrating services along the cloud-to-edge continuum [8, 17, 20].

A typical microservice application consists of many loosely-coupled services, each of which can be independently deployed, tested, and scaled [19]. The microservice architecture provides numerous benefits when compared to their monolith predecessors such as loosely coupled development and independent deployment. In turn, this improves scalability and availability, and affords the practice of continuous integration and continuous delivery (CI/CD).

However, microservice architectures come with their own host of challenges as the number of microservices can balloon to hundreds of microservices each in the order of tens of thousands of lines of

[1]https://www.cncf.io/blog/2021/07/15/networking-with-a-service-mesh-use-cases-best-practices-and-comparison-of-top-mesh-options/

code [22]. Furthermore, service calls in such a dynamic architecture are asynchronous and disordered by nature. Left unmanaged, this can create bottlenecks that affect the performance of the whole service chain (referred to as the 'back pressure' effect); *cf.* [9, 14].

Consequently, there is a need for the adopters of microservices to operate additional 'glue-ware' to carry out communication and management tasks such as orchestration, monitoring, load balancing, message passing, resilience, etc. In fact, 2022 was the first year when such glue-ware outweighed the number of core microservices [7]. The measures taken by software engineers to provide such glue-ware includes a number of tools and practices [1, 21, 22], including widely used solutions such as Kubernetes and Spinnaker.

2.2 The need for SMTs

Orchestration tools, such as Kubernetes, deploy containerized applications into pods and provide basic east-west communication channels between them. However, such communication is by default not monitored and unencrypted. More generally, and importantly, Kubernetes does not allow any management of such pod-to-pod communication. Consider the example of an application deployer who wants to set policies about when and how frequently to call certain services. For this to be possible, they would need to integrate such policy enforcement as auxiliary instruments in the core containers (*i.e.,* the ones with business logic). This has several significant disadvantages: mainly, it violates the concept of separation of concerns as business logic is muddled with network management logic in an ad hoc fashion, and it is an invasive and fragile practice.

An alternative approach is the pattern of *sidecar proxies*. This is where each microservice is shipped as a container that contains the core business logic as well as another container called the *sidecar*. The sidecar acts as the ingress and egress point of the business container and, thus, can apply auxiliary logic such as networking services, monitoring, metering, and configuration.

SMTs use the pattern of sidecar proxies to provide communication between microservices that is low-latency, secure, and monitored. As such, it offers a feature-rich data plane while also providing steer over the control plane in terms of traffic management, and service and traffic observability. In effect, SMTs allow application deployers to declaratively and programmatically define policies to govern service to service interactions, and extract this logic out of the core business logic. This could be versioned and applied consistently at different levels, *i.e.,* at container or cluster level.

In a survey of the CNCF End User Community, the use of SMTs increased from 27% of respondents in 2020 [6] to 47% in 2022 [7].

3 RELATED WORK

We are motivated to assess the degrees of suitability for running major SMTs in edge and IoT environments where resources are constrained and the need for low latency is of high priority. A fair number of works studied SMTs either in qualitative or quantitative ways. Here, we give an overview of these works.

3.1 Qualitative studies

A number of papers offer qualitative studies of SMTs. El Malki and Zdun [4] conduct an analysis of 40 grey literature publications to distil the practices around using SMTs. Li et al. [15] give an early survey of SMTs and their challenges. Duque et al. [3] provide a qualitative evaluation of SMTs for edge applications, focusing specifically on traffic management use cases (*e.g.,* QoS assurance and flexible configurability). There is also countless tech blog posts about the qualitative differences in SMTs[2], but we do not list them for space constraints.

3.2 Quantitative studies

In 2019, a team at Kinvolk set out to compare the performance of Istio and Linkerd[3]. They created a benchmark that employed emojivoto[4], a 3-microservice application created by Linkerd for demonstration purposes. The benchmark sends multiple requests per second, and records the latency per request and resource usage of the cluster. Overall, they found Linkerd to be more efficient than Istio in terms of both resource usage and latency. It is important to note that this study used Istio version 1.1.6, as Istio has had a major restructure of its internal components with release 1.5[5] wherein control plane functionality was consolidated into the `istiod` daemon[16]. The above study was reproduced by Linkerd[6] using Linkerd 2.10 and Istio 1.10.0, and came up with very similar findings. However, they also used an application that consists of only 3 microservices so it is not viable to generalize their findings to more realistic microservice applications that consist of tens or hundreds of microservices.

In 2020, Dahlberg created a benchmark to compare the performance of Linkerd and Istio [2]. His benchmark used TeaStore, a web store application of seven microservices. Apache JMeter was used to send concurrent requests to the meshes. He observed Linkerd to be more memory-efficient, but Istio to be faster. He also noted that Linkerd struggled more when handling concurrent requests and speculates that this could be due to a resource-limiting feature, as the CPU was not fully utilized, which would explain the added latency when dealing with multiple users. However, the benchmark in this study only tests the application with a maximum of 40 simultaneous users. This may be representative of smaller internal technologies, but may not accurately reflect larger applications.

Ganguli et al. [10] presented an analysis of using a Kubernetes cluster with an Istio mesh on an edge application. The study is primarily concerned with the overheads of such a setup on the ability to serve an increasing number of user requests. Finally, Saleh Sedghpour et al. [18] present experimental results to identify the best way to use SMTs particularly for the circuit breaker functionality in a microservice-based application of a complex topology.

3.3 Summary

We surmise that there is a number of works that studied SMTs from a qualitative point of view, but do not compare them in an empirical and experimental manner. There are only a few efforts that do assess the overheads of different SMTs, but they are either limited in scale or use outdated SMT releases.

[2]For instance: https://www.containiq.com/post/comparing-service-meshes-istio-vs-linkerd-vs-consul
[3]https://kinvolk.io/blog/2019/05/performance-benchmark-analysis-of-istio-and-linkerd/
[4]https://github.com/BuoyantIO/emojivoto
[5]https://istio.io/v1.5/blog/2020/istiod/
[6]https://linkerd.io/2021/11/29/linkerd-vs-istio-benchmarks-2021/

4 SMT SOLUTIONS

In this section we provide an introduction to the SMTs that we study and compare. They are chosen due to their popularity in the DevOps community at large and specifically in cloud and edge systems. Note that a detailed qualitative differentiation of these solutions is not our aim here, and is covered by other works as already mentioned (*e.g.,* [3, 13, 15]).

4.1 Envoy

Envoy[7] is not strictly an SMT, but it needs introduction as it plays a vital role in major SMTs such as Istio and Kuma[8]. Written in C++, Envoy is an open source sidecar proxy that was donated by Lyft to the CNCF in September 2017.

Essentially, Envoy is a component that sits on the request path and acts as part of the application, forcing all application communications to go through it. It configures the container's IP table and gives the application a clean and unified interface to the network that is high-performance and managed universally. This relates to offering added features such as load balancing, rate limiting, circuit breaking, and more. Envoy's architecture is made up mainly of *listeners* that ingests data plane traffic, *filters* that process said traffic, and *routes* that applies layer 7 logic (*i.e.,* traffic policies).

4.2 Istio

Istio is a very popular SMT, partly because it is developed to be well integrated with Kubernetes and is an CNCF project since September 2022. It uses Envoy for the data plane and implements its own control plane. The data plane deploys a set of Envoy proxies as sidecars attached to pods, and they mediate and control all network communication between microservices. They also collect and report measurements on all mesh traffic. The control plane, in the form of the `istiod` service since release 1.5, manages and configures the proxies to route traffic and operates discovery and certification operations.

Sidecar configuration is done through Custom Resource Definitions (CRDs) that are then converted into Envoy configurations that are communicated to all proxies. `istiod` operates a service registry to detect newly deployed microservices and track currently deployed ones. It also operates as a deployment-specific certification authority for certificate management (*e.g.,* automated certificate circulation every x hours/days). Istio gives further added-values such as the ability to use GraphQL for querying service communications and telemetry. It also supports Extended Berkeley Packet Filter (eBPF) for additional network observability.

Setup of an Istio mesh can be done using predefined profiles that apply the mesh automatically. If more customized configuration is required, the user needs to edit Kubernetes manifests files (*e.g.,* in YAML) to annotate a namespace or pods that need meshing.

4.3 Linkerd

Like Istio, Linkerd is an open-source CNCF project (since July 2021) and is well integrated with Kubernetes. Architecturally speaking, Linkerd looks similar to Istio prior to version 1.5. It consists of sidecar proxies and a distributed set of control plane services. The differences, however, are abundant when you inspect the implementation details. Instead of Envoy, Linkerd uses its own proxy, called Linkerd2-proxy, which is written in Rust and is dubbed a 'micro-proxy' by its developers. It is purpose-built with clear emphasis on asynchronous networking and being lightweight, and thus ships with a smaller set of features when compared to Envoy yet still supports the functionality expected from an SMT such as observability, security, and reliability. It also has native integration with Prometheus monitoring framework.

Linkerd uses a declarative configuration model. It does not need editing of any user-facing YAML per se, only the inclusion in the Kubernetes deployment plan to signify where proxy sidecars are needed. As such, its configuration is rather straightforward.

5 EXPERIMENTAL SETUP

We now explain the setup we used in our evaluation experiments.

5.1 Methodology

Scale. We devised microservice application of different scales, being composed of 10, 20, 40, and 80 microservices.

SMTs and baseline. We evaluated the latest releases from Istio and Linkerd, specifically Istio version 1.15.3 and Linkerd version 2.12.3. As a baseline, we compare against a bare Kubernetes deployment.

Metrics. We are interested chiefly in the effectiveness and efficiency of SMTs specifically for edge systems where resources may be constrained and/or costly. We measured the following metrics:

- **Latency**: The time taken to communicate between microservices as evidence for the effectiveness of each SMT. This is measured as the time between submitting a request and receiving a corresponding request.
- **Memory footprint**: This is the first indication of the overhead of an SMT. We look at both the overall memory used as well as the breakdown.
- **CPU utilization**: This is another measurement of SMT overhead, which we also measure in aggregate and detailed forms.

5.2 Hardware

Our experiments were carried out on an OnLogic Helix 600 edge server. This is a state-of-the-art device towards the high-end of edge capabilities [11], which is representative of devices used in un-staffed micro data centers to support suburban and remote settings [5]. It has an Intel Core i7-10700T Comet Lake 4.5 GHz 8-core processor with 32 GB of DDR4 memory, and it runs Ubuntu 20.04.

5.3 Workload

We developed a web application using the Flask Python framework[9]. For our purpose, the key requirement is to have an architecture that can be easily scaled horizontally, *i.e.,* by adding more or removing microservices. For an overview, see Fig. 1.

Our application represents workloads that are made up of Read-Modify-Write transactions. Functionally, the application works as a counter: it receives a request containing a start value and an objective value. The application increments the current value and

[7]https://www.envoyproxy.io/
[8]https://kuma.io/
[9]https://flask.palletsprojects.com/en/2.2.x/

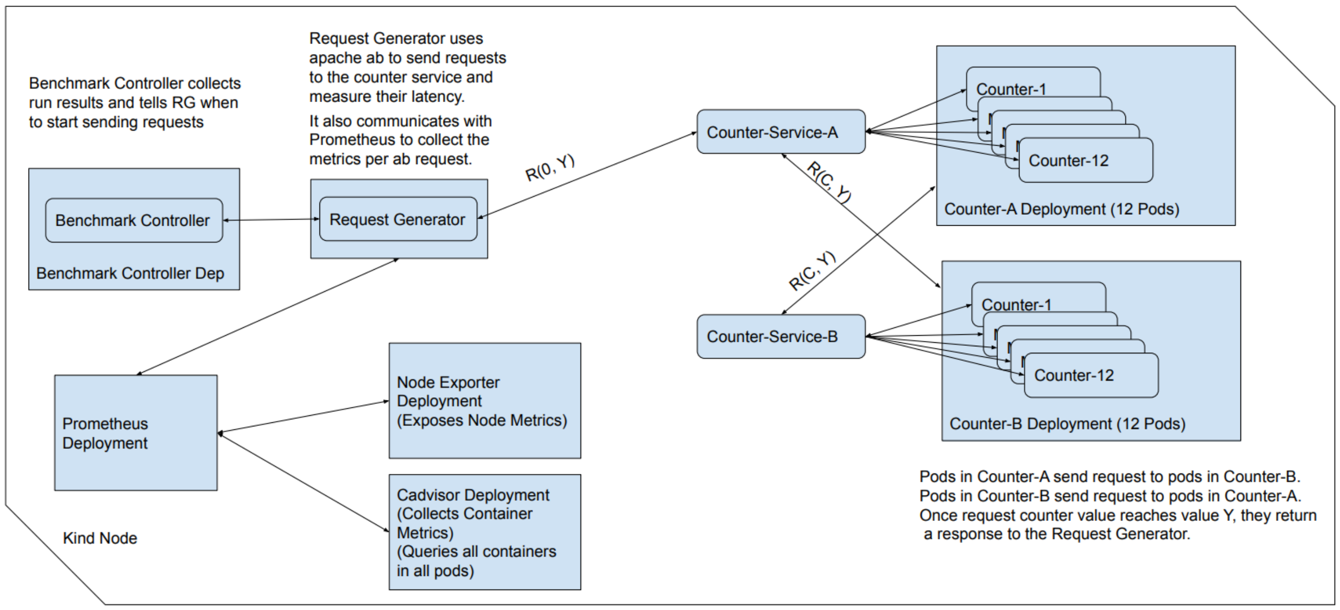

Benchmark Controller collects run results and tells RG when to start sending requests
Request Generator uses apache ab to send requests to the counter service and measure their latency.
It also communicates with Prometheus to collect the metrics per ab request.
Benchmark Controller
Benchmark Controller Dep
Request Generator
R(0, Y)
Counter-Service-A
R(C, Y)
R(C, Y)
Counter-Service-B
Counter-1
Counter-12
Counter-A Deployment (12 Pods)
Counter-1
Counter-12
Counter-B Deployment (12 Pods)
Prometheus Deployment
Node Exporter Deployment (Exposes Node Metrics)
Cadvisor Deployment (Collects Container Metrics) (Queries all containers in all pods)
Kind Node
Pods in Counter-A send request to pods in Counter-B.
Pods in Counter-B send request to pods in Counter-A.
Once request counter value reaches value Y, they return a response to the Request Generator.

Figure 1: An overview of the overall experiment architecture.

resubmits the request to another microservice. This process continues until the value reaches the objective value.

To represent realistic usage of an edge system, we simulate multiple users submitting requests simultaneously to the application. To do so, we used Apache ab[10]. We specified a total number of requests and how many requests were to be sent concurrently. The concurrent requests are used to simulate simultaneous users sending requests to the application, while the total number of requests divided by the number of concurrent ones, gives us an average number of requests per user (RPUs). Each user will send a request, await a response, and once received send another request until the total RPUs is reached for every simulated user.

5.4 Cluster

A Kubernetes cluster with one node was created using the Kubernetes in Docker (KinD) distribution version 0.17.0. KinD allows the easy building and testing of Kubernetes pods. Eight pods were used by the Kubernetes system to manage the cluster, while three are dedicated to collect resource usage. One pod is used to start the benchmark execution and store the results. Another pod is used to generate requests and record their response rate. An additional 24 pods were used to simulate the micro-services. Some of the remaining pods are used for the service mesh control plane.

5.5 Deployment and Operation

The counter application was containerized using Docker and two Kubernetes deployments were defined containing 12 pods each running the application. Pods can run multiple containers simultaneously, however only one counter container was created per pod. Every replica runs the counter application and is hosted in its own pod. Two Kubernetes services were created to expose the counter deployments to the cluster's network. This creates an URL per deployment, *i.e.,* all pods inside the deployment are pointed at by the same URL. When a request is sent to that URL, either Kubernetes's or the SMT's load balancer (depending on what is being tested) selects the pod to receive the request.

Two deployments, Counter-A and Counter-B, were specified, such that Counter-A would send requests to Counter-B and vice versa. This is to avoid having pods sending requests to themselves, which is unrealistic.

A benchmark controller and a request generator were deployed. The controller starts the experiment and stores the results. Once a service mesh is installed, the controller will send a collection of values defining the number of simultaneous users, RPU and number of microservices to the request generator. The request generator will then simulate those users, each sending the specified number of requests, which are sent through the counters until the specified number of microservices is attained.

To record the memory footprint and CPU utilization, Prometheus alongside Node Exporter and cAdvisor were used. Like the benchmark controller, these also remained unmeshed throughout the experiment to prevent the meshes from delaying the measurements of resource usage. To be clear, the only components of the experiment that were meshed were the request generator and the counter applications.

5.6 Code availability

To support reproducibility, we release our code and data on Github[11].

[10]https://httpd.apache.org/docs/2.4/programs/ab.html

[11]https://github.com/JoseLuisPovedanoPoyato/SMTComparisonEdge/

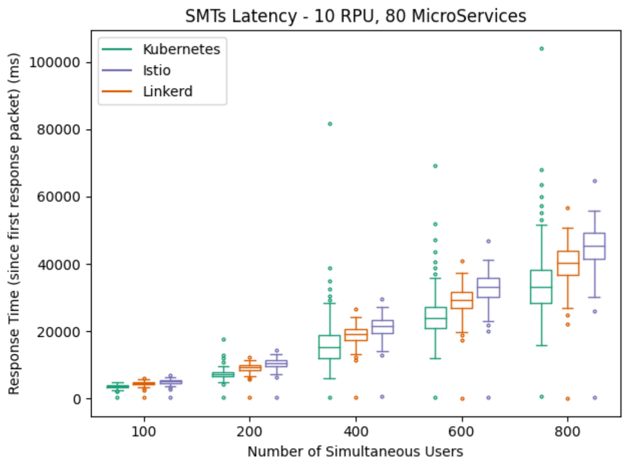


Figure 2: A box-plot showing the latency experienced using native Kubernetes, Istio, and Linkerd for increasing number of application users.

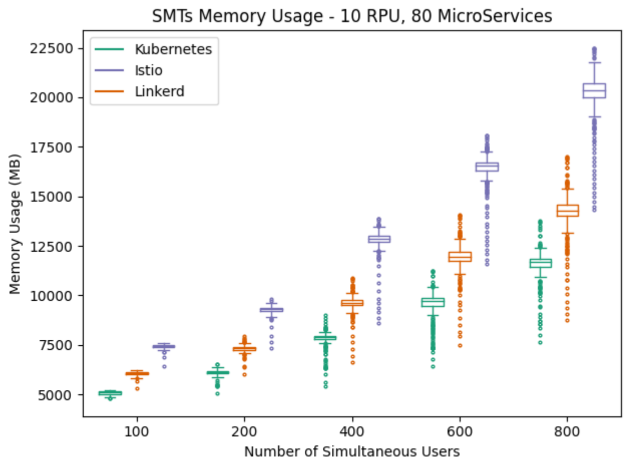


Figure 3: A box-plot showing the total memory used by Kubernetes, Istio, and Linkerd for increasing number of application users.

6 RESULTS

Note that although we ran experiments with different numbers of microservices, we only show and discuss the results for an 80 microservice application due to space constraints. The results for the 10-, 20-, and 40-microservice applications show similar findings. Similarly, we also ran experiments with 3 and 5 RPU but we only show the results for 10 RPU as it suffices. The full set of results is available through our Github repository.

6.1 Latency

Overall, we notice that Istio incurs the highest latency across different scales of number of users (Fig. 2). Linkerd comes second (about 6%–12% lower than Istio), while native Kubernetes has the least latency. Linkerd's gain over Istio increases with scale as the difference, as Linkerd's median values get increasingly lower than Istio's lower quartile latency.

Additionally, it is interesting to note that latency with either SMT is much more predictable than that of Kubernetes. The latter becomes increasingly unpredictable as the number of simultaneous users increases, when its latency distribution exhibits a very long tail as shown by the outlier markers at the top of the plot in Fig. 2.

6.2 Memory footprint

When we look at overall memory utilization (Fig. 3), we find that it follows the general trend that we observed with latency: Kubernetes has the least memory overhead (obviously), followed by Linkerd, then finally Istio. However, here the differences are significant especially between Linkerd and Istio: Istio uses between 20% and 43% more memory than Linkerd. This might be a deciding factor for favoring the former over the latter for devices with limited memory resources.

In order to stratify the memory utilization and identify the cause for Istio's higher consumption, we also plot the breakdown of memory utilization in Fig. 4. We only show the plots for 100 simulated application users with 20 microservices, and 800 users with 80 services for space constraints. The key observation here is that Istio's data plane requires 3x–5x more memory than Linkerd. This is due to Istio's use of the Envoy proxy as a universal sidecar.

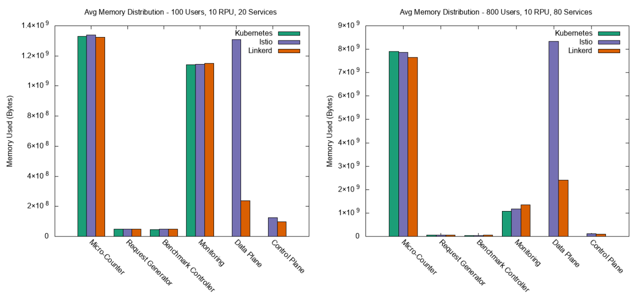
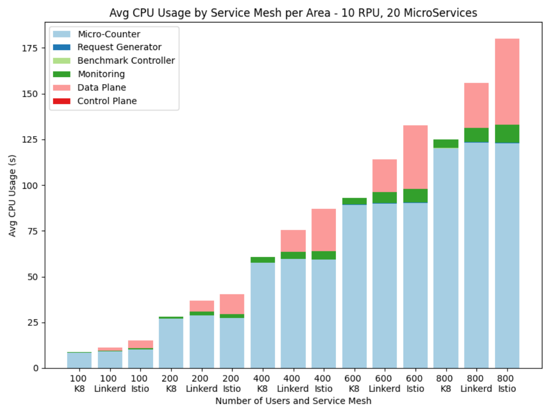


Figure 4: A breakdown of the memory utilization of the application subprocesses using each of Kubernetes, Istio, and Linkerd for the bottom of the experiment scale at 100 users and 20 services, and the top at 800 users and 80 services.

6.3 CPU utilization

The levels of CPU utilization (shown in Fig. 5) follow similar trends as those seen before, but at much less pronounced differences. Here, we display the values for 20 microservices and that of 80. Although the values are different (notice the scale of the y-axis), the different series compare to each other in almost identical ways. Linkerd uses around 22%–25% more CPU than Kubernetes, while Istio uses a further 16%–20% more than Linkerd. The main culprit is again the data plane, namely the collection of Envoy proxies and the central `istiod` service.



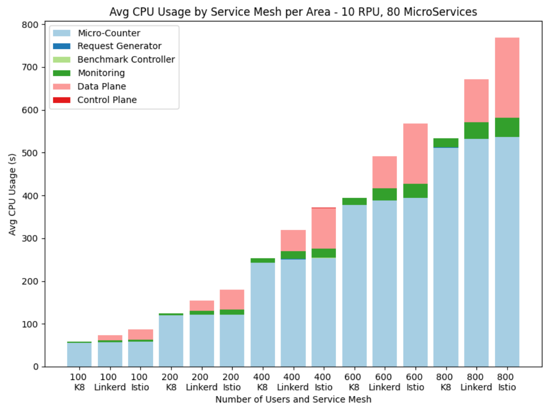


Figure 5: The CPU utilization of the SMTs compared to that of Kubernetes at the scales of 20 (top) and 80 microservices (bottom). The total value in each case is dissected into the different application sub-processes.

7 CONCLUSION

In this paper, we provided a quantitative assessment of the suitability of two service mesh technologies (SMTs) for use in edge environments: namely, Istio and Linkerd. In our experiment, we observed Linkerd to be less resource-hungry than Istio especially in memory utilization. We also remarked that this reduced computational overhead comes with response latencies that are lower than Istio's, and are more predictable in their variance than bare Kubernetes. These findings coupled with the relative ease of setup of Linkerd lends it quite well to use in edge and IoT environments.

ACKNOWLEDGMENTS

This work was partly supported by the UK EPSRC under grant number EP/R010889/2.

REFERENCES

[1] Justus Bogner, Jonas Fritzsch, Stefan Wagner, and Alfred Zimmermann. 2019. Microservices in Industry: Insights into Technologies, Characteristics, and Software Quality. In *International Conference on Software Architecture Companion (ICSA-C)*. 187–195. https://doi.org/10.1109/ICSA-C.2019.00041

[2] Erik Dahlberg. 2020. *Analytical and Experimental Comparison of Service Meshes in Kubernetes*. Master's thesis. Umeå University, Department of Computing Science.

[3] Aleksandra Obeso Duque, Cristian Klein, Jinhua Feng, Xuejun Cai, Björn Skubic, and Erik Elmroth. 2022. A Qualitative Evaluation of Service Mesh-based Traffic Management for Mobile Edge Cloud. In *IEEE International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*. 210–219. https://doi.org/10.1109/CCGrid54584.2022.00030

[4] Amine El Malki and Uwe Zdun. 2019. Guiding Architectural Decision Making on Service Mesh Based Microservice Architectures. In *Software Architecture*, Tomas Bures, Laurence Duchien, and Paola Inverardi (Eds.). Springer International Publishing, Cham, 3–19.

[5] Yehia Elkhatib, Barry F. Porter, Heverson B. Ribeiro, Mohamed Faten Zhani, Junaid Qadir, and Etienne Rivière. 2017. On Using Micro-Clouds to Deliver the Fog. *Internet Computing* 21, 2 (March 2017), 8–15. https://doi.org/10.1109/MIC.2017.35

[6] Cloud Native Computing Foundation. 2020. *CNCF Annual Survey*. Technical Report. https://www.cncf.io/reports/cloud-native-survey-2020/.

[7] Cloud Native Computing Foundation. 2022. *CNCF Annual Survey*. Technical Report. https://www.cncf.io/reports/cncf-annual-survey-2022/.

[8] Kaihua Fu, Wei Zhang, Quan Chen, Deze Zeng, and Minyi Guo. 2022. Adaptive Resource Efficient Microservice Deployment in Cloud-Edge Continuum. *Transactions on Parallel and Distributed Systems* 33, 8 (2022), 1825–1840. https://doi.org/10.1109/TPDS.2021.3128037

[9] Yu Gan, Yanqi Zhang, Dailun Cheng, Ankitha Shetty, Priyal Rathi, Nayan Katarki, Ariana Bruno, Justin Hu, Brian Ritchken, Brendon Jackson, Kelvin Hu, Meghna Pancholi, Yuan He, Brett Clancy, Chris Colen, Fukang Wen, Catherine Leung, Siyuan Wang, Leon Zaruvinsky, Mateo Espinosa, Rick Lin, Zhongling Liu, Jake Padilla, and Christina Delimitrou. 2019. An Open-Source Benchmark Suite for Microservices and Their Hardware-Software Implications for Cloud & Edge Systems. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. ACM. https://doi.org/10.1145/3297858.3304013

[10] Mrittika Ganguli, Sunku Ranganath, Subhiksha Ravisundar, Abhirupa Layek, Dakshina Ilangovan, and Edwin Verplanke. 2021. Challenges and Opportunities in Performance Benchmarking of Service Mesh for the Edge. In *IEEE International Conference on Edge Computing (EDGE)*. 78–85. https://doi.org/10.1109/EDGE53862.2021.00020

[11] Cheol-Ho Hong and Blesson Varghese. 2019. Resource Management in Fog/Edge Computing: A Survey on Architectures, Infrastructure, and Algorithms. *ACM Comput. Surv.* 52, 5, Article 97 (sep 2019), 37 pages. https://doi.org/10.1145/3326066

[12] Humantiec. 2021. *DevOps Setups – A Benchmarking Study*. Technical Report. https://humanitec.com/whitepapers/2021-devops-setups-benchmarking-report.

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

[14] Richard Li, Min Du, Zheng Wang, Hyunseok Chang, Sarit Mukherjee, and Eric Eide. 2022. LongTale: Toward Automatic Performance Anomaly Explanation in Microservices. In *The ACM/SPEC International Conference on Performance Engineering (ICPE)*. 5–16. https://doi.org/10.1145/3489525.3511675

[15] Wubin Li, Yves Lemieux, Jing Gao, Zhuofeng Zhao, and Yanbo Han. 2019. Service Mesh: Challenges, State of the Art, and Future Research Opportunities. In *IEEE International Conference on Service-Oriented System Engineering (SOSE)*. 122–1225. https://doi.org/10.1109/SOSE.2019.00026

[16] Nabor C. Mendonça, Craig Box, Costin Manolache, and Louis Ryan. 2021. The Monolith Strikes Back: Why Istio Migrated From Microservices to a Monolithic Architecture. *IEEE Software* 38, 5 (2021), 17–22. https://doi.org/10.1109/MS.2021.3080335

[17] Qian Qu, Ronghua Xu, Seyed Yahya Nikouei, and Yu Chen. 2020. An Experimental Study on Microservices based Edge Computing Platforms. In *IEEE INFOCOM 2020 - IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*. 836–841. https://doi.org/10.1109/INFOCOMWKSHPS50562.2020.9163068

[18] Mohammad Reza Saleh Sedghpour, Cristian Klein, and Johan Tordsson. 2022. An Empirical Study of Service Mesh Traffic Management Policies for Microservices. In *Proceedings of the 2022 ACM/SPEC on International Conference on Performance Engineering* (Beijing, China) *(ICPE '22)*. Association for Computing Machinery, New York, NY, USA, 17–27. https://doi.org/10.1145/3489525.3511686

[19] Johannes Thönes. 2015. Microservices. *IEEE Software* 32, 1 (2015), 116–116. https://doi.org/10.1109/MS.2015.11

[20] M. Villari, M. Fazio, S. Dustdar, O. Rana, and R. Ranjan. 2016. Osmotic Computing: A New Paradigm for Edge/Cloud Integration. *IEEE Cloud Computing* 3, 06 (nov 2016), 76–83. https://doi.org/10.1109/MCC.2016.124

[21] Muhammad Waseem, Peng Liang, and Mojtaba Shahin. 2020. A Systematic Mapping Study on Microservices Architecture in DevOps. *Journal of Systems and Software* 170 (2020), 110798. https://doi.org/10.1016/j.jss.2020.110798

[22] Qilin Xiang, Xin Peng, Chuan He, Hanzhang Wang, Tao Xie, Dewei Liu, Gang Zhang, and Yuanfang Cai. 2021. No Free Lunch: Microservice Practices Reconsidered in Industry. *CoRR* abs/2106.07321 (2021). arXiv:2106.07321 https://arxiv.org/abs/2106.07321