Exploring, Analyzing and Tuning Service Mesh Performance: A Literature Review

Pranav Singh
School of Computer Science and Engineering
Vellore Institute of Technology, Tamil Nadu, India
pranav.singh2020@vitstudent.ac.in

Dr. Ayyasamy.S
School of Computer Science and Engineering
Vellore Institute of Technology, Tamil Nadu, India
ayyasamy.s@gmail.com

***Abstract*—With the recent advancement of software development methodologies and frameworks, service meshes has been to rapidly gain the lime light. In this paper, we provide an introduction to service meshes and discuss their key use cases in modern cloud-native architectures. We also explore the challenges associated with performance analysis of service meshes and present a survey of recent research in this area. To address these challenges, we propose the use of soft computing techniques for performance analysis of service meshes. Our study includes a comparative analysis of different service mesh platforms and their performance under varying workloads. Overall, this paper provides insights into the benefits and limitations of service meshes and highlights the importance of performance analysis in ensuring the reliability and scalability of micro-services based applications.**

***Index Terms*—Benchmark Testing, Cloud-native, Configuration Management, Edge, Envoy, DevOps, Istio, Kubernetes, LinkerD, Micro-Services, Network Management, Policy-Driven Management, Performance Engineering, Reinforcement Learning , Service Mesh , Soft Computing**

I. Introduction

Service meshes is like a software defined network [1] where the operators gets complete control over the different set of activities that is occurring within their application. Its a dedicated infrastructure layer that oversees service-to-service communication. It's responsible for the reliable delivery of requests through the complex topology of services that comprise a modern, cloud native application [2]. In practice, the service mesh is typically implemented as an array of lightweight network proxies that are deployed alongside application code, without the application needing to be aware.

Each part of an app, called a “service” relies on other services to give users what they want. If a user of an online e-commerce store wants to by groceries, they need to know if the items they want, say “tomatoes” is in stock. So, the service that communicates with the e-commerce site’s inventory database needs to communicate with the product webpage, which itself needs to communicate with the users’s online shopping cart. To add business value, the owner of the site might eventually build a new service that gives users in-app product recommendation. Now this new service should also be cognizant of user’s shopping cart, and must be able to communicate with inventory database that the product page needed—it’s a lot of reusable, moving parts.

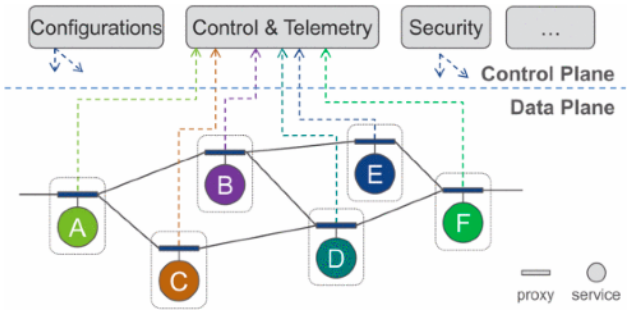

Figure 1. Sample Architecture for Service Mesh [3]

The above definition and use case example can further be mingled with the architecture depicted in Figure 1. In most cases, a service mesh in general is known to have two types of planes: Control Plane and Data Plane. Data plane is composed of intelligent proxies that are deployed as sidecars, basically an extra container deployed under the same Pod as the application container. These proxies regulate the flow of network communication between different microservices. Non-exhaustive list of activities managed by this data plane includes service discovery, health checking, routing, load balancing, authentication/authorization, and observability. While the control plane, sometimes referred to as brain of a service mesh, manages and configure corresponding components to enforce policies and collect telemetry.

A survey [4] of Cloud Native Computing Foundation(CNCF) community found that 68% of the organization are already using or planning to use service meshes in the next 12 months. In production, use of service meshes has been growing 40-50% annually [5]. Service meshes are popular because they solve critical problems related to communication between loosely coupled services (also referred as microservices [6]), which are widely adopted

by enterprise companies due to their ease of management/ development for production workloads.

II. Fundamental Feature

In general, a service mesh is designed to provide a wide set of fundamental features, as described and elaborated in the study of [3]. Few of the widely adopted features are:

- Service Discovery: In system of microservices, there is always a changing number and location of services. Thus, a tooling to discover the service and their location becomes a critical problem to be solved for production grade applications. Typically, service instances are discovered by tracking a registry which keeps track of creation and deletion of services.
- Load balancing: This feature is more inclined towards the capability of traffic routing across network. Compared to simple routing mechanism (eg round robin, and random routing), areas like latency and the state (eg., health status, and current variable load) of the backend instance are also covered. Moreover, developers could add their own logic by applying a set of custom filters in the network.
- Fault Tolerance/Resiliency: Seeing through the eyes of a networking engineer, a service mesh sits at a layer of abstraction above TCP/IP . Considering the past trends, we can assume that this part of the network layer is too fragile and more unreliable, so service meshes takes care of this area as well by redirecting the consumer request to a service instance with healthy state.
- Traffic monitoring: Every form/medium of communication is to be captured and reported for encompassing volumes per target, latency metrics, success and error rates, etc. Prometheus [7] and Grafana [8] are two of the well know CNCF [9] projects used mainly for this purpose and works well integrated with most popular service meshes.
- Circuit Breaking: In case of accessing an overly loaded service though a request with more latency than expected, a service mesh would not only disallow that request to even pass through the gateway and rather prevent the entire service from going into a broker state.
- Authentication and access control: This features allows the services to be aware of the host from from which an incoming request originated or allows only a set of requests to actually pass through the gateway that matches with the policies defined in a filter applied to the network. Further more disallowing the requests from unauthenticated services.

III. Service Meshes in Edge computing

The main idea behind edge computing is that, processing of client data is performed as close to the source as possible. This is usually preferred to be performed in a distributed fashion and at the periphery of the network. To achieve this methodology, some production based software are implemented in a fashion wherein, some portions of storage and compute resources are moved out of the central data center and closer to the source of the data itself. Hence, the extra overhead of transmitting the raw data is avoided. There are several applications of this approach like the health meter available in the modern watches does the processing of data right onto users' wrist, or autonomous self driving cars does the heavy lifting of predicting traffic movements right onto the cars system rather than transmitting the data to a remote server, then waiting for processing the data and the response to come back until it could decide its next set of movements on the road which is too time consuming and dangerous.

Considering the deployments scenarios, microservice deployment at the Edge could benefit from the variety of environments which Kubernetes [10] supports. Kubernetes is known go-to software and toolset for orchestration and management at the Edge. Moreover, a wide array of service meshes are known to be well integrated with Kubernetes, such as Istio [11], Envoy [12], LinkerD [13], Consul [14], Traefik Mesh [15], Cilium Mesh [16] etc.

Some of the different scenarios for considerations are:

- Pod to pod communication
- Pod to service communication
- Ingress controller to pod and vice-versa
- Load balancer to pod and vice-versa

A. Bare-metal Deployment

Various production grade systems adopts the concept of bare-metal as service for their applications. This gives an easy win wherein, the developers have complete control over the hardware resources and corresponding configurations to optimize performance of their workloads. [17] deployed Envoy sidecar proxy on bare-metal node to analyze the performance. Their deployment uses Docker containers [18].

In accordance to the Envoy's documentation [19], Envoy proxy is used as a load balancer or front proxy before allowing the traffic to micro service replicas that use Envoy sidecar in service-to-service proxy mode.

Envoy proxy connects to a software load generator Fortio [20] using a 10G back-to-back connection or connection through a top-of each switch. In accordance to Fortio's documentation [20], the Fortio client sends some http requests with varying Queries per Second (QPS), connections and test duration. The front-envoy routes all the incoming requests, acting as a reverse proxy sitting on the edge of the envoy mesh network. All traffic routed by the front Envoy to the service containers is routed to the service Envoys that in turn route the request to the Flask app, a simple app that responds with a "hello" message.

Goals of this experiment were to:

- Analyze envoy proxy performance with increasing number of queries per second.
- Increase the number of client connections to obtain maximum QPS resolved successfully
- 1 & 2 tested and compared on a 48 core Xeon vs a 32 core Xeon with no core pinning.
- Core scaling experiments done in 20 core Xeon with increasing QPS, connections and clones.

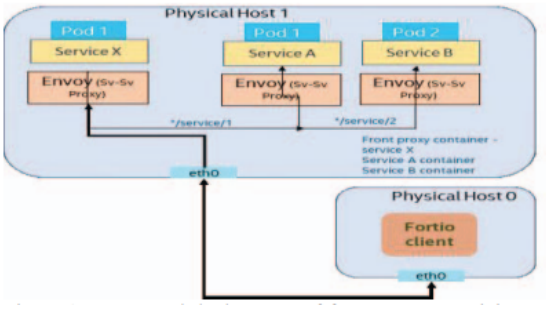

Figure. 2. BareMetal deployment for server mesh and edge computing [17]

As depicted from Figure 3 that there lies a correlation between the QPS and number of client connections. That is, with increasing number of client connections, increases the QPS. The above experiment used 100 clones of Flask app with front end proxy being configured to round-robin balancing mode. Through put was observe to be saturating at about 10000 input QPS at various connection rates. While P99 tail latencies remained steady across different QPS rate.

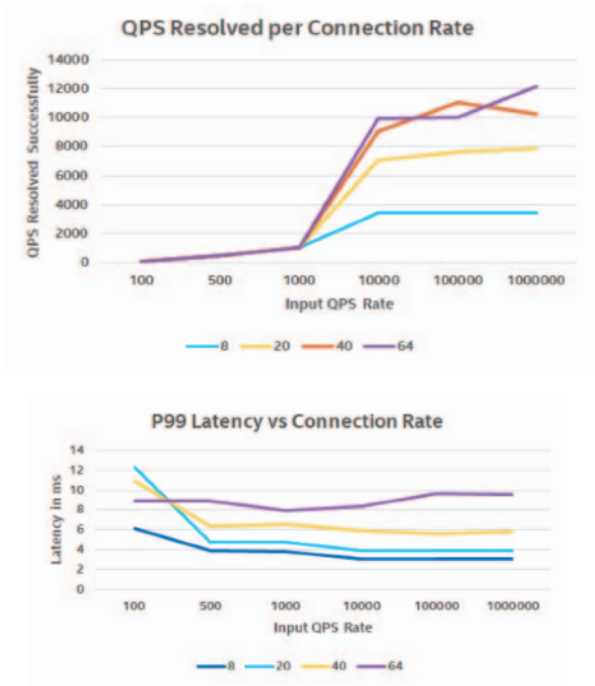

Figure 3. Connection scaling tests on Xeon for varying Connection Count [17]

According results captured by figure 4 OPS had interconnection with the number of cores allocated. That is, on a 20 core Xeon show that the QPS resolved increases with the increase in number of cores allocated to the sidecar+flask app. The tail latencies decrease with the the increase in number of cores for all 1. 10, and 100 clones but increases with the number of connections.

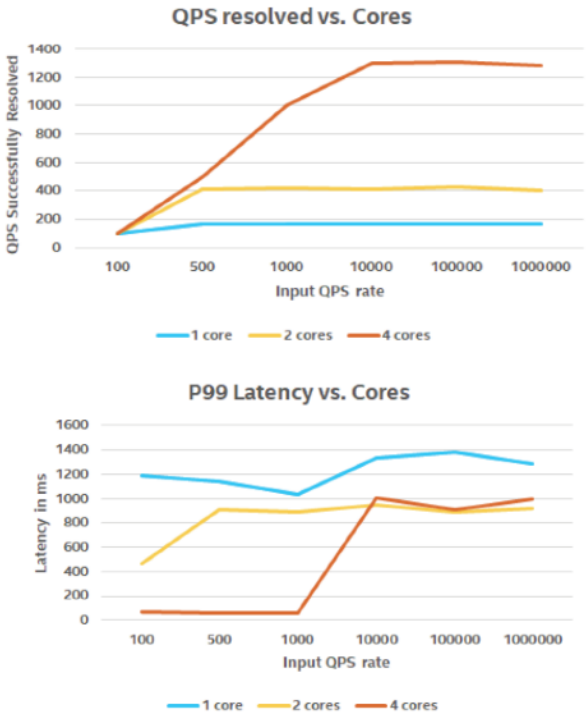

Figure 4. Core scaling tests with Envoy and Flask app [17]

B. Virtualized Deployment

When a microservice orchestrator (like Kubernetes) and service mesh is deployed as a stack in VM then such deployments is referred to as Virtualized deployment. According to the customer satisfaction, VMs are preferred over bare-metal due to hardware level isolation between application which comes in-house with VMs.

The second form of experiment by [17], involved two VMs across two nodes, with one acting as Kubernetes master while other as worker node, as depicted in Figure 5. The purpose of this form of their study was to determine the impact of OpenvSwitch (OVS) and Data Plane Development Kit (DPDK) on data plane performance between north-south traffic across microservices. Calico [21] is used as the CNI of choice enabling networking between Kubernetes pods using simple network policies that enable external traffic across Kubernetes NodePort of the worker node.

This study was more of a comparative type of approach for analyzing performance with and without Istio. For this case, a simple Nginx [22] web server container was used as the application server that returns client queries generated by HTTP load generator. Both hardware load generators, with Ixia IxLoad, and software load generator Fortio were used to study the max performance of the setup.

Fortio client is run as a standalone process as these two processes:

1. Fortio client in Kubernetes master VM, to simulate East-West traffic across VMs from outside the cluster.
2. Fortio client in Master-host (in Figure 5) to simulate North-South traffic from outside cluster.

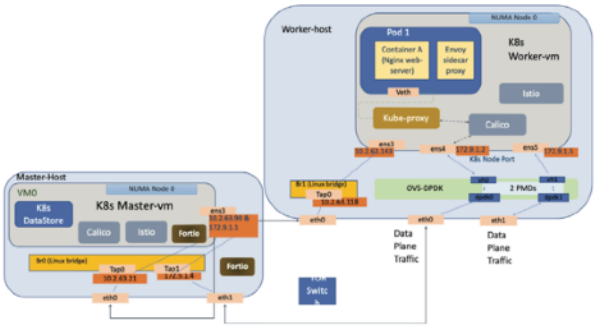

Figure 5. Virtualized deployment for server mesh and edge computing [17]

For the VM based deployment, it was discovered that throughput deviation of traffic between VMs of 50%-70% when using Istio and Envoy for 64 connections. Moreover, from the graph in figure 6 demonstrates that the performance degradation with Istio shows up with input QPS more than 1000 and tail latencies are lower without Istio by about 50%, indicating Istio doubles the tail latencies.

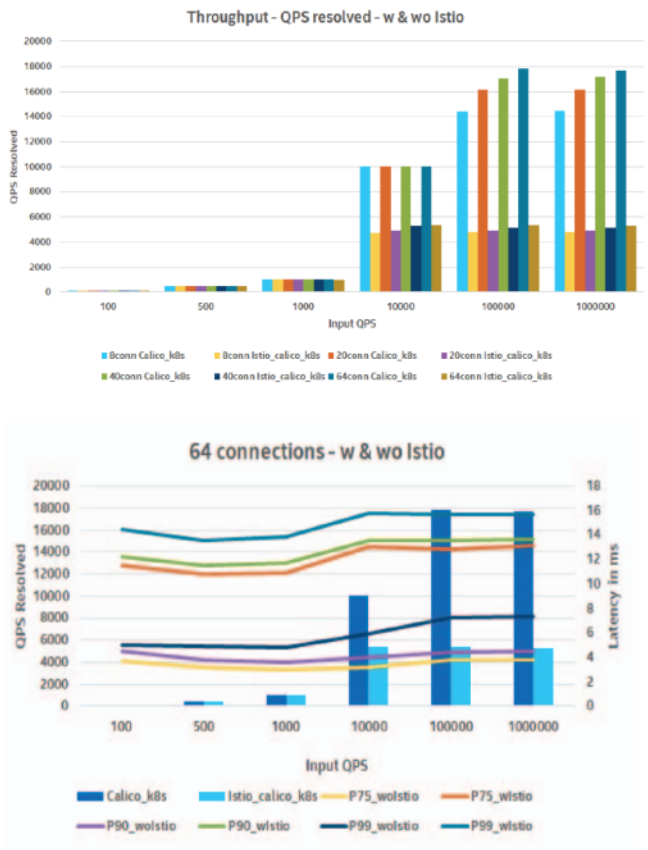

Figure 6 . Core scaling tests with Envoy and Flask app [17]

IV. Service Fault Resiliency with Reinforcement Learning

Development and deployment of modern web applications are increasingly becoming distributed, thanks to the microservice based architectures that facilitate such granular control over apps. While this provides tremendous flexibility in overall software development lifecycle, but it comes with its own pitfalls when observed in high traffic-based environment. That is, the notion of fault resiliency. The power of services meshes can be harnessed to tackle such cases, as these provides fault resiliency through attribute configurations that govern the behavior of request-response services (and the interactions among them) in presence of failures. [23], proposes a novel model-based reinforcement learning workflow towards service mesh fault resiliency, which they refer as SFR2L (Service Fault Resiliency with Reinforcement Learning). Their approach enables the prediction of most significant fault resilience behaviors at a web application level, scratching from single service to aggregated multi-service management with efficient agent collaboration.

A. Experimental setup

Traffic & Loading Setting	Explanation
Max Pending Requests	Max queries on requesting connection
Max Connections	Max existing connection
Max Requests Per Connection	Max allowed requests for each connection
Ejection Time	The service ejected duration
Max Ejection	Max ejected service
Interval	Time between ejection and recovery
Consecutive Errors	Max consecutive failed requests
Total Threads	Max available threads
Total Calls	Total number of pending requests

Figure 7. Represents fault resiliency attributes

i. Data collection from an Istio application

This step involved collection of datasets covering target parametric spaces of Istio httpbin service [24], each varying configuration settings of traffic rules and fault injection and load testing settings as demonstrated in Figure 7.

ii. Simulation model training and selection

This step involved multiple-layer perceptrons (MLP) to simulate the aggregated behaviors, enabling agents to interact with the environment and learn the best loading space given the traffic rule attributes. Figure 8 depicts the input-output relation for modeling application-level fault resiliency.

They selected networking communication models outlined in [25], which were Logistic Regression, Linear Ridge Regression, and Support Vector Regression, as baseline simulation models to fight against 5 MLP and observed which emulates the best application response.

They collected 5 groups of structured datasets and splitted them into 8:2 as training and testing set, respectively. For MLP, the input later had 9 neurons, 3 hidden layers have 512 neurons and the output layer has 2 neurons. The learning rate varied from 1^{-6} to 1^{-5}. Figure 10 demonstrates the range of traffic rule and thread and call settings used in their experiment.

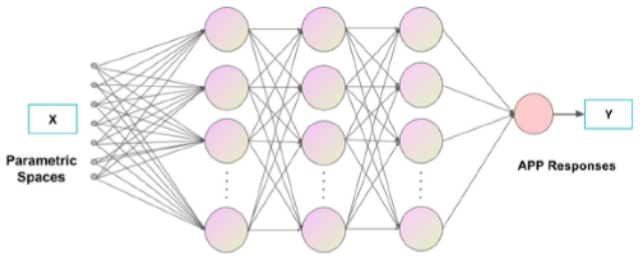

Figure 8 . Simulation Model of web application-level fault resiliency. X is a 9-dimensional vector with 7 deterministic traffic rules and 2 loading settings to be decided by reinforcement learning agent(s), Y is the web application response vector with 2 features: QPS and failure rate. [23]

*iii.*Model-based Reinforcement learning

This step involved training the model using reinforcement learning (RL). Here, environment was their simulation model (well trained MLP), states corresponded to the traffic rule settings, actions determines the loading settings (the number of threads and loading calls). Here, agents learn from responses of their simulation model of the Istio httpbin service to the perform actions as shown in figure 9.

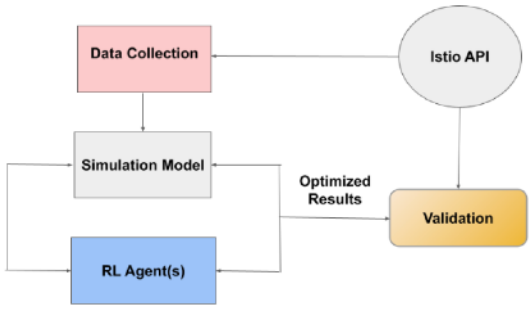

Figure 9. SFR2L Pipeline. Data collection and validation are based on actual Istio API. RL Agent(s) fully interact with simulation model and validate its decisions in actual Istio API. [23]

iv. Validation on policy learning results

At last, they trained the reinforcement learning algorithms [23] to perform policy learning and obtained optimized loading decisions given different traffic rule and loading setting combinations.

They focused on "worst-case" rewards (penalties) as it provides insight into configuration settings that are critical to use in load testing prior to application deployment. The learning ability is characterized by how much RL algorithm outperform baselines under the same context, whose metrics is the maximum rolling mean ratio of cumulative reward obtained in RL to baseline in last 25 epoch.

They implemented their simulation model to interact with RL agent(s) and record activities at each epoch that gives highest reward ratio. Data points from each best epoch was used as loading and traffic parameters to trip the actual service response and obtain the validated reward ratio. All their experiments analyzed single agent and multiple/collaborative agent model-based reinforcement learning using their algorithm. And each experiment is performed for 3 times and then mean is calculated for 3 results.

Traffic Rule & Loading Setting	**S1**	**S2**	**S3**	**S4**	**S5**
Max Pending Requests	1-7	3-7	12-18	12-18	15-30
Max Connections	1-7	3-7	1-5	10-20	5-15
Max Req Per Connection	1-7	3-7	10-16	12-18	15-30
Ejection Time	3m	3m	3m	3m	3m
Max Ejection	100%	100%	4-8%	12-18%	22-30%
Interval Time	1s	1s	1s	1s	1s
Consecutive Error	1	1	4-8	12-18	22-30
Total Threads	1-5	3-7	10-16	12-18	16-20
Total Calls	400-450	100-700	50-500	250-600	1000-2000
Dataset Size	9302	12005	20592	12310	6970
Simulation Model	**S1**	**S2**	**S3**	**S4**	**S5**
Support Vector Regression	1.3	0.78	1.05	1.33	1.24
Logistic Regression	0.93	0.81	0.99	1.00	1.00
Linear Ridge Regression	0.85	0.84	0.98	1.01	0.96
5 layer-MLP	**0.13**	**0.17**	**0.52**	**0.63**	**0.38**

Figure 10. Ranges of Traffic Rule, Thread, and Call Settings; Model evaluation metrics: MSE (Mean Square Error).

Configurations \ Datasets		S1		S2		S3		S4		S5	
		Sim.	Val.	Sim.	Val.	Sim.	Val.	Sim.	Val.	Sim.	Val.
Single for Single	Call	1.03	1.01	2.71	1.77	1.75	1.31	2.05	1.81	1.31	1.29
	Thread	2.21	1.63	1.04	0.99	1.18	0.98	1.16	1.00	1.02	0.93
Multi for Single	Thread&Call	2.26	2.15	3.45	2.80	1.84	1.44	2.07	2.65	1.31	1.45
	Thread-Call	2.24	2.36	3.39	2.57	1.96	1.32	2.14	3.07	1.32	1.20
	Call-Thread	2.22	2.11	3.44	3.00	1.79	1.62	2.11	2.52	1.33	1.33
Multi for Multi	Call*5	1.01	1.00	2.96	2.30	1.77	1.43	2.05	2.87	1.33	1.30
	Thread*5	2.23	1.83	1.15	1.28	1.13	1.06	1.18	1.03	1.01	1.16
	Thread&Call*5	2.26	2.35	4.12	2.92	1.84	1.29	2.11	2.83	1.32	2.05
	Thread-Call*5	2.22	1.51	3.53	2.52	1.94	2.21	2.09	3.18	1.34	1.44
	Call-Thread*5	2.28	2.19	3.50	2.33	1.99	2.40	2.11	2.16	1.33	1.44

Figure 11. Policy Evaluations: Sim. is the maximum rolling reward ratio in simulation, Val. is the maximum rolling reward ratio in validation, *5 means 5 services are aggregated and communicative

B. Results

Their model-based RL algorithm outperformed all other baselines in most scenarios as depicted from figure 11. Thread/call had different effects on the policy learning: thread was more significant for S1 model and call was more significant for S2-S5 model. Moreover, multi-agents learning abilities were closed to single agent when either factor was trivial to fault resiliency, such as thread in S2, S5 and call in S1 model.

For the case of single service cases, most of multi-agents worked better than single agent decisions, which proved that complex parameter interdependence optimization could fulfill the potential of policy learning. Example, Thread&Call agents gained 27% higher rewards than Call only (3.45 to 2.71) agent in simulation and 69% higher rewards (2.80 to 1.77) in validation stage for S2 model.

Moreover, multi-agents usually had higher validation accuracy than single agent. Like for the case of S1 model, Thread only agent had 2.21 reward ratio in simulation and

1.63 in validation (36% higher), but Thread&Call agent had closed 2.26 validated ratio (5% higher).

All in all, their model based reinforcement learning algorithm could be used to predict which values of traffic rule settings, threads and calls yield rewards with respect to fault resiliency of the Istio httpbin service. The configuration settings that yield the "worst-case" rewards give insight into which combinations of configurations should be tested rigorously during load testing to ensure robust fault recovery, as these areas have high chances of compromising application level fault resiliency.

V. Service Mesh Performance and Meshery

A. Service Mesh Performance

To grease the wheels and provide a vendor neutral comparison of all major service meshes, Service Mesh Performance (SMP) [26], a CNCF project is widely being used for comparison of service meshes. Its a standard for capturing and characterizing the details of infrastructure capacity, service mesh configuration, and workload metadata.

It facilitates:

i) The ability to reason over the efficiency by which cloud native infrastructure is run, specifically in context of a service mesh and its network functions,
ii)Benchmarking of service mesh performance
iii)Common vernacular and measurement for exchange of performance information from system-to-system and mesh-to-mesh
iv)Apples-to-apples performance comparisons of service mesh deployments and tooling to trend workload performance.
v)A universal performance index to gauge a service mesh's efficiency against deployments in other organizations' environments.

It consistently runs performance tests on most of the service meshes under different test environments on CNCF hosted labs [27] and exports the results into a public facing dashboard [28]. This helps end-users to make decision on the choice of their service mesh. For example, Figure 12 and Figure 13 demonstrates the latest soak test results from SMP dashboard. These test were performed on Istio service mesh and LinkerD service mesh with Fortio as the load generator and both system under test were in the same testing environment.

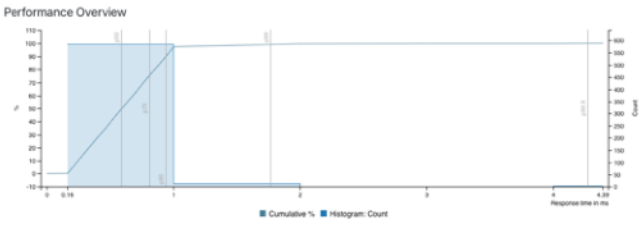

Figure 12. Istio Soak Test with Fortio load generator [29]

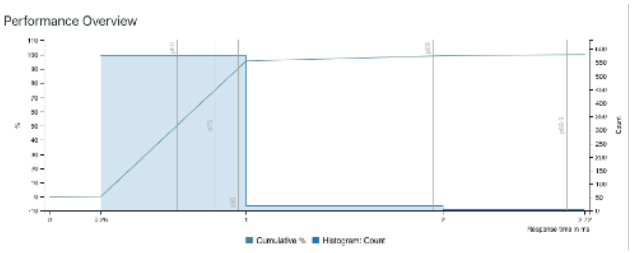

Figure 13. LinkerD Soak Test with Fortio load generator [30]

This project further comes with a novel implementation of performance index, called MeshMark [31]. This measures the value and overhead of cloud native environment. It converts performance measurements into insights about the value of individual, cloud native application networking functions, MeshMark distills a variety of overhead signals and key performance indicators into a simple index. It helps in assessing the value of your service mesh in context of its cost. This was announced in ServiceMeshCon EU 22 [32], and since then it has received positive feedback from community members and adopters, and has been actively used for benchmarking performance tests results through Meshery.

B. Meshery

Meshery [33] is another popular Layer5 project donated to CNCF, that is known as a cloud native manager, capable of providing governance, policy and performance, and configuration management of cloud native infrastructures. As a tool, it supports wide array of deployment scenarios to fit into almost all use cases of end-users. This further helps the adoption of Meshery in enterprises and greases the wheels for performance benchmarking, just one of the many capabilities of Meshery.

As explained in [34], its a canonical implementation of SMP, which facilitates end-users to perform load testing on their environment, that could be testing, staging or production based. Tooling that implements MeshMark includes Meshery, the cloud native management plane. Users of Meshery can configure their Kubernetes deployments, any and every service mesh as well as onboard and off-board their workloads onto any given mesh. Once they have done so, Meshery will begin to calculate MeshMarks continuously.

[35] explains several use cases and key features of Meshery and its service mesh management capabilities. Some of its key features includes:

i) 3 load generators, that is Fortio, wk2 [36], and Nighthawk [37], to fulfill different needs and use cases of end users.
ii)Protocols like TCP, gRPC, HTTP for load generation
iii)Performance profiling
iv)Scheduled performance management

V. Comparisons Between Popular Meshes

Some of the attractive options for service meshes are Istio, LinkerD, Amazon App Mesh [38], and Airbnb Synapse [39].

Istio and LinkerD are one of the most trusted and used service meshes for prod based infrastructure. This is because they provide handful of the fundamental functionalities of a mesh. While Istio has more active community that revolves

around the project and its development, LinkerD is more stable and accredited by many adopters as productions ready. While LinkerD is a graduated CNCF project while Istio was just recently accepted into CNCF, and is in incubating stage. Moreover, Istio provides a large number of flexible APIs for developers, and comes with a visual topology offering, i.e., Kiali, to facilitate visual representation of the deployment environment.

Unlike Istio and LinkerD, Airbnb Synapse does not have an abundant feature set. Rather, it comes with the feature of automated failover through a simplistic but efficient, Zookeeper-based service discovery mechanism. So, Synapse configures a local HAProxy process using the information read from Zookeeper where states of the service are stored and continuously updated. Though this, the local HAProxy can take care of properly routing the request from a service consumer. With an optimized HAProxy, Synapse can react to the change (eg, service failure) in Zookeeper, and reconfigure HAProxy immediately. Synapse has been proven to be with broad applicability, however, it wasn't designed to be used as a mesh but more of an internal project for Airbnb to solve internal development challenges. Because of this, there isn't much development around it.

As of 2018, Amazon, announced its own service mesh, named Amazon App Mesh. Its offering are largely native to the existing Amazon cloud ecosystem. So, if an organization is already using Amazing cloud for its production usage then its no brainer to use the App Mesh as it would solve many inherent challenges but at the same time would make the customers into vendor lock-in since its not so applicable into hybrid cloud environment with multiple vendors like Google, RedHat, VMware etc.

VI. Future Research Opportunities

Though adoption of service mesh seems a daunting task but it has already been started to use in production and adopted by wide range of companies. IT complements micro service architectures in terms of observability, traceability, and manageability.

One of the attractive offering of a service mesh is dependency management, and performance tracing and analysis for micro service applications. Some of well known extension supports, eg Jaeger [40] and Zipkin [41] are based on OpenTracing framework which requires instrumentation to the applications using Open Tracing API specification. Though this approach helps in precisely identify where failures occurs and what causes poor performance, such instrumentation is not always realistic, eg, when service implementation are not exposed is not allowed to be modified due to copyright/license issues etc

Hence, an enhancement with non-intrusive mechanism becomes a necessity to expand the use case scenario on performance tracking and analysis. One promising direction would be to investigate the feasibility of integration inference-based approaches, such as Project5 [42], WAP5 [43], Microsofts' Sherlock [44], and Facebook's The Mystery Machine [45]. In order to adapt the application context of service mesh, we expect this may require extensive effort with both research and engineering.

More research is needed to explore the use of soft computing techniques such as fuzzy logic and genetic algorithms for optimizing the performance of service meshes. There is a need to investigate the impact of service mesh configuration parameters on performance and to develop more efficient algorithms for selecting optimal configurations. There is also a need to develop better visualization techniques for monitoring and diagnosing the performance of service meshes. There is an opportunity to explore the use of machine learning techniques for predicting the performance of service meshes based on historical data. Finally, there is a need to investigate the performance of service meshes in complex, real-world scenarios and to develop new techniques for handling scalability, security, and fault tolerance. Addressing these research challenges will enable us to fully realize the potential of service meshes for modern, cloud-native applications.

VII. Conclusion

In this paper, we have presented an overview of service meshes and their use cases in modern cloud-native applications. We have also discussed the challenges associated with performance analysis of service meshes, including the need to consider multiple metrics and the complexity of workload patterns. To address these challenges, we have presented a literature survey of different tooling available in the market to capture and enhance the performance of service meshes. That includes the use of soft computing techniques and configuration management management tools like Meshery for performance analysis of service meshes, and presented a comparative analysis of different service mesh platforms using these techniques.

Our study highlights the importance of performance analysis in ensuring the reliability and scalability of microservices-based applications. We have shown that while service meshes offer numerous benefits such as traffic management, security, and observability, their performance can be impacted by factors such as the number of nodes in the cluster, the configuration of the mesh, and the type of workload. By using soft computing techniques or tools like Meshery, we can gain a deeper understanding of the impact of these factors on service mesh performance and identify ways to optimize it.

In conclusion, our study contributes to the growing body of research on service meshes and their performance analysis. We hope that this paper helps practitioners and researchers in making informed decisions about the use and possible optimization of service meshes in their applications. Future research can explore the use of other soft computing techniques, such as fuzzy logic and genetic algorithms, or use inbuilt support offered by MeshMark from SMP for service mesh performance analysis and optimization.

VIII. References

1. W. Xia, Y. Wen, C. H. Foh, D. Niyato and H. Xie, "A Survey on Software-Defined Networking," in IEEE Communications Surveys & Tutorials, vol. 17, no. 1, pp. 27-51, Firstquarter 2015, doi: 10.1109/COMST.2014.2330903.
2. N. Kratzke and R. Peinl, "ClouNS - a Cloud-Native Application Reference Model for Enterprise Architects," in 2016 IEEE 20th International Enterprise Distributed Object Computing Workshop (EDOCW), Vienna, Austria, 2016 pp. 1-10.
3. W. Li, Y. Lemieux, J. Gao, Z. Zhao and Y. Han, "Service Mesh: Challenges, State of the Art, and Future Research

Opportunities," 2019 IEEE International Conference on Service-Oriented System Engineering (SOSE), 2019, pp. 122-1225, doi: 10.1109/SOSE.2019.00026.
4. CNCF survey 2020. https://www.cncf.io/ wp-content/ uploads/2020/11/CNCF_Survey_ Report_2020.pdf, 2020.
5. Benchmarking Linkerd and Istio: 2021 re- dux. https://linkerd.io/2021/11/29/ linkerd-vs-istio- benchmarks-2021/, 2021.
6. Dragoni, N. et al. (2017). Microservices: Yesterday, Today, and Tomorrow. In: Mazzara, M., Meyer, B. (eds) Present and Ulterior Software Engineering. Springer, Cham. https:// doi.org/10.1007/978-3-319-67425-4_12
7. Prometheus, [online] Available: https://prometheus.io/.
8. Grafana, [online] Available: https://grafana.com/.
9. CNCF: Cloud Native Computing Foundation, [online] Available: https://www.cncf.io/.
10. Kubernetes: Production-Grade Container Orchestration, [online] Available: https://kubernetes.io/.
11. Istio: An Open Platform to Connect Manage and Secure Microservices., [online] Available: https://github.com/istio/ istio.
12. Envoy: an open source edge and service proxy, designed for cloud native applications,. [online] Available: https:// www.envoyproxy.io/.
13. Linkerd: Production-grade Feature-rich Service Mesh for Any Platform., [online] Available: https://github.com/ linkerd/linkerd.
14. Traefik mesh: and open source service mesh, [online] Available: https://traefik.io/traefik-mesh/.
15. Consul service mesh: a service mesh offering from Hashicorp, [online] Available: https:// developer.hashicorp.com/consul/docs/connect.
16. Cilium service mesh: an eBPF-based open source service mesh from Isovalent, [online] Available: https://cilium.io/.
17. M. Ganguli, S. Ranganath, S. Ravisundar, A. Layek, D. Ilangovan and E. Verplanke, "Challenges and Opportunities in Performance Benchmarking of Service Mesh for the Edge," 2021 IEEE International Conference on Edge Computing (EDGE), 2021, pp. 78-85, doi: 10.1109/ EDGE53862.2021.00020.
18. Docker containers, [online] Available: https:// www.docker.com/resources/what-container/.
19. Envoy documentation, [online] Available: https:// www.envoyproxy.io/docs/envoy/v1.25.5/.
20. Fortio, a load testing library, [online] Available: https:// github.com/fortio/fortio/#fortio.
21. Project Calico, an open source project for container networking, [online] Available: https://www.tigera.io/ project-calico/.
22. Nginx, a project offering from F5, [online] Available: https://docs.nginx.com/.
23. Meng, Fanfei, Lalita Jagadeesan, and Marina Thottan. "Model-based Reinforcement Learning for Service Mesh Fault Resiliency in a Web Application-level." arXiv preprint arXiv:2110.13621 (2021).
24. Istio httpbin, a sample application provided by Istio, [online] Available: https://github.com/istio/istio/blob/ master/samples/httpbin/httpbin.yaml.
25. R.Boutaba,M.A.Salahuddin,N.Limam,S.Ayoubi,N.Shahriar ,F.E.Solano,and O. M. C. Rendon. A comprehensive survey on machine learning for networking: evolution, applications and research opportunities. J. Internet Serv. Appl., 9(1):16:1– 16:99, 2018.
26. Service mesh performance (SMP), a CNCF project, [online] Available: https://smp-spec.io/.
27. CNCF Community Infrastructure Lab (CIL), [online] Available: https://github.com/cncf/cluster.
28. SMP dashboard, [online] Available: https://smp-spec.io/ dashboard.
29. LinkerD soak test on SMP Dashboard, [online] Available https://smp-spec.io/dashboard/performance/ individual#306389ec-9bc6-4b1f-92e3-fe765632c572#0.
30. Istio soak test on SMP Dashboard, [online] Available: https://smp-spec.io/dashboard/performance/ individual#4b8f9849-cd1f-40e0-a95c-55a84b34ff31#0
31. MeshMark, a performance index for cloud native environment, [online] Available: https://smp-spec.io/ meshmark.
32. Lee Calcote and Mrittika Ganguli, 2022 Service Mesh Con EU 22, [online] Available: https:// events.linuxfoundation.org/servicemeshcon-europe/.
33. Meshery, a CNCF project, [online] Available: https:// meshery.io/.
34. The Enterprise Path to Service Mesh Architectures (2nd Edition) by Lee Calcote, [online] Available: https:// layer5.io/learn/service-mesh-books/the-enterprise-path-to-service-mesh-architectures-2nd-edition.
35. Lee Calcote, Mrittika Ganguli, Sunku Raganath, Otto Van der Schaaf, “Analyzing Service Mesh Performance” in IEEE Quality-of-Service Architecture for Cloud Computing Networking Magazine, vol 117, issue 3, 2021
36. Wrk2, a HTTP based benchmarking tool, [online] Available: https://github.com/giltene/wrk2.
37. Nighthawk, a L7 (HTTP|HTTPS|HTTP2) performance characterization tool, [online] Available: https://github.com/ envoyproxy/nighthawk.
38. AWS App Mesh, [online] Available: https:// aws.amazon.com/app-mesh/.
39. Synapse: A Transparent Service Discovery Framework for Connecting an SOA, [online] Available: https://github.com/ airbnb/synapse.
40. Jaeger: Open Source End-to-End Distributed Tracing, [online] Available: http://jaegertracing.io.
41. OpenZipkin: A Distributed Tracing System, [online] Available: https://zipkin.io.
42. M. K. Aguilera, J. C. Mogul, J. L. Wiener, P. Reynolds and A. Muthi-tacharoen, "Performance Debugging for Distributed Systems of Black Boxes", Proceedings of the 19th ACM Symposium on Operating Systems Principles ser. SOSP '03, pp. 74-89, 2003.
43. P. Reynolds, J. L. Wiener, J. C. Mogul, M. K. Aguilera and A. Vahdat, "WAP5: Black-box Performance Debugging for Wide-area Systems", Proceedings of the 15th International Conference on World Wide Web ser. WWW '06, pp. 347-356, 2006.
44. P. Bahl, R. Chandra, A. Greenberg, S. Kandula, D. A. Maltz and M. Zhang, "Towards Highly Reliable Enterprise Network Services via Inference of Multi-level Dependencies" in ACM SIGCOMM Computer Communication Review, ACM, vol. 37, no. 4, pp. 13-24, 2007.
45. M. Chow, D. Meisner, J. Flinn, D. Peek and T. F. Wenisch, "The Mystery Machine: End-to-end Performance Analysis

of Large-scale Internet Service", Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation ser. OSDI '14, pp. 217-231, 2014.
46. C. Ren, H. Li, Y. Li, Y. Wang, H. Xiang and X. Chen, "On Efficient Service Function Chaining in Hybrid Software Defined Networks," in IEEE Transactions on Network and Service Management, vol. 19, no. 2, pp. 1614-1628, June 2022, doi: 10.1109/TNSM.2021.3123502.

Pranav Singh is a postgraduate student, studying computer science and engineering at Vellore Institute of Technology, Tamil Nadu, India. He is passionate about building technology and making an impact through software development. Currently, he is exploring the software world through open-source contributions. His interest lies in DevOps, micro-services, software architecture, software defined networks, service meshes and cloud-native technologies.

Dr. Ayyasamy.S is a senior professor from school of computer science and engineering at Vellore Institute of Technology, Tamil Nadu, India.