EDUSCHOOL JOURNAL OF COMPUTER SCIENCE RESEARCH STUDIES (EJCSRS)

(Open Access, Double-Blind Peer Reviewed Journal)

ISSN Online:

Service Mesh Architecture in Depth: An ISTIO versus Linkerd Analysis for Multi-Cluster Kubernetes Environments

Bini P B

Assistant Professor, Department of Computer Science, CCSIT Dr John Matthai Center, Thrissur, India

Article information

Received: 27th February 2026
Received in revised form: 31st March 2026
Accepted: 4th May 2026
Available online: 12th June 2026

Volume: 1
Issue: 2
DOI: https://doi.org/10.5281/zenodo.20661101

Abstract

As organisations decomposed monoliths into hundreds of microservices, a hidden problem surfaced: the logic for securing, routing, retrying, and observing the traffic between services had been copied into every service, in every language, with subtle inconsistencies. The service mesh is the architectural answer. It lifts that cross-cutting concern out of application code and into a dedicated infrastructure layer, typically a proxy deployed alongside each service instance, governed by a central control plane. This paper examines the design and compares the two most prominent open-source meshes, Istio and Linkerd, with particular attention to the demands of multi-cluster Kubernetes deployments. We separate the control plane from the data plane and explain why that separation matters, describe how transparent traffic interception and mutual TLS provide zero-trust security without changing application code, and contrast the two projects' fundamentally different data-plane choices: Istio's general-purpose Envoy proxy against Linkerd's purpose-built lightweight proxy. We then weigh the trade-off that defines the comparison, feature breadth and configurability against operational simplicity and lower per-proxy overhead, and discuss multi-cluster connectivity, identity federation, and the emerging sidecar-free architectures that may reshape the field. The conclusion is not that one mesh wins but that the right choice follows directly from whether an organisation values control surface or operational economy.

Keywords:- Service Mesh, Istio, Linkerd, Kubernetes, Sidecar Proxy, Envoy, Mutual TLS, Multi-Cluster, Control Plane, Microservices, Observability.

I. INTRODUCTION

Microservice architectures solved a problem of organisational scale and created a problem of network scale. Splitting an application into many independently deployed services means that what used to be a function call is now a remote call across an unreliable network, and the concerns that a single process handled implicitly, retries, timeouts, encryption, authentication, and telemetry, must now be handled explicitly between every pair of services [1], [13]. The early response was to embed this logic in shared client libraries, but that approach tied every service to a language and a library version and made a fleet-wide change a fleet-wide redeployment.

The service mesh emerged to move this concern out of the application entirely [1], [2]. The dominant pattern places a proxy next to each service instance, the sidecar, which intercepts all of that instance's traffic and applies the cross-cutting policies on its behalf, while a central control plane configures the fleet of proxies and distributes the identities and certificates they need [2], [3]. Because this happens below the application, a service written in any language gains uniform security, resilience, and observability without a line of code changing. On Kubernetes, where workloads are already organised into pods and managed by a declarative control loop, the

model fits naturally [3], and surveys of cloud-native adoption show service meshes moving steadily into mainstream production use [15].

This paper compares the two leading open-source meshes, Istio and Linkerd, and frames the comparison around multi-cluster Kubernetes, where a single mesh spans several clusters for availability, locality, or isolation. The two projects make opposite bets at the most important design point, the data plane, and understanding that divergence is the key to choosing between them [4], [5]. We describe the shared architecture, contrast the two implementations, present the overhead trade-off that the choice turns on, and survey where the technology is heading.

II. ANATOMY OF A SERVICE MESH

A. Control Plane and Data Plane

Every mesh divides into two layers, and keeping them distinct is the central architectural idea. The data plane is the set of proxies that actually carry application traffic; it sees every request and is therefore on the critical path for both latency and reliability [6]. The control plane never touches a request. Its job is management: discovering services, generating and rotating the certificates that secure communication, compiling high-level policy into proxy configuration, and pushing that configuration out [3], [4]. This separation means an operator changes behaviour by declaring intent to the control plane, which translates it into thousands of consistent proxy configurations, rather than by touching individual services. Figure 1 shows the two planes and the sidecar topology that connects them.

B. Transparent Interception and Zero-Trust Security

The sidecar earns its keep because interception is transparent. Traffic to and from the application is redirected to the local proxy by the platform's networking layer, so the application believes it is talking directly to its peer while in fact every byte passes through two proxies [2]. This is what lets a mesh deliver mutual TLS across the entire fleet automatically: each proxy is issued a cryptographic identity by the control plane, and the proxies authenticate each other and encrypt the connection between them with no involvement from application code [3], [7]. The result is a practical realisation of zero-trust networking, in which services prove who they are on every connection rather than trusting the network they sit on. The same interception point yields uniform, language-independent metrics, distributed-tracing headers, and access logs as a side effect [8].

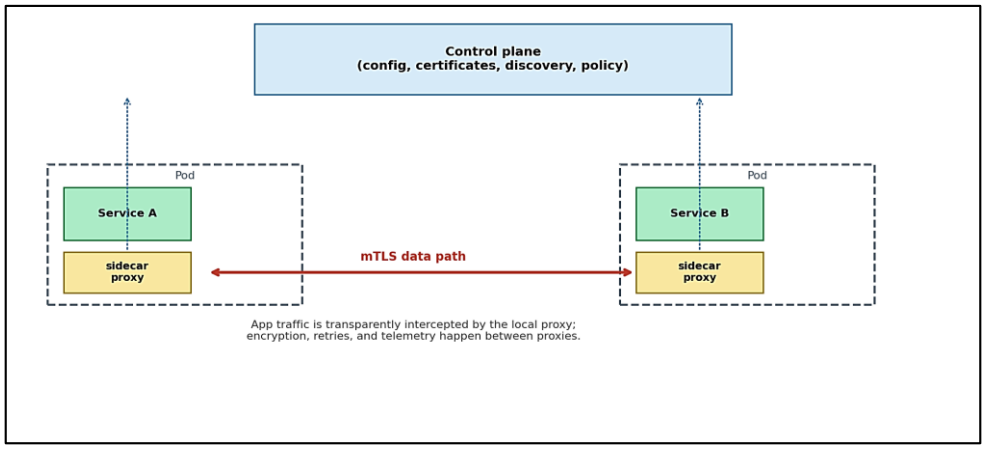

Fig 1: The two-plane architecture. The control plane configures and secures the proxies; the sidecar data plane transparently intercepts application traffic and enforces mTLS, retries, and telemetry between proxies.

III. THE DATA-PLANE DIVERGENCE: ENVOY VERSUS A PURPOSE-BUILT PROXY

A. Istio and Envoy

Istio's data plane is Envoy, a general-purpose, high-performance proxy originally built at Lyft and now a widely adopted project in its own right [6]. Envoy is extraordinarily capable: it supports rich traffic management, fine-grained routing, fault injection, and a dynamic configuration interface that lets the control plane reconfigure it at runtime without restarts [6], [10]. Istio builds on that capability to expose one of the broadest feature sets of any mesh, including sophisticated traffic shifting, policy, and telemetry options [4]. The cost of that generality is weight. Envoy is a large, configurable C++ proxy, and running one beside every workload consumes meaningful

memory and adds latency, while the breadth of configuration surface is itself a source of operational complexity [11].

B. Linkerd and the Micro-Proxy

Linkerd makes the opposite choice. Rather than adopt a general-purpose proxy, it ships a small data-plane proxy written specifically for the mesh use case in Rust, which gives it memory safety and a deliberately narrow feature set [5]. The proxy is designed to do the common mesh tasks, mTLS, latency-aware load balancing, retries, and metrics, with minimal resource consumption and few configuration knobs [5]. The philosophy is operational simplicity: fewer features that work the same way everywhere, lower per-proxy overhead, and a control plane that is correspondingly easier to run. The cost is the mirror image of Istio's: capabilities that Istio exposes through Envoy may be absent or require an external tool [9], [11].

Table 1. Istio and Linkerd Compared at the Data Plane

Dimension	Istio	Linkerd
Data-plane proxy	Envoy (general-purpose, C++)	Purpose-built micro-proxy (Rust)
Feature breadth	Very broad, highly configurable	Focused on common mesh tasks
Per-proxy overhead	Higher memory and latency	Lower memory and latency
Operational complexity	Higher; large config surface	Lower; opinionated defaults
Extensibility	Strong (WASM, filters, EnvoyFilter)	Limited by design

IV. OVERHEAD AND PERFORMANCE

Because every request traverses two proxies, the data plane adds latency and consumes resources, and this overhead is the quantity any mesh evaluation must measure. The cost has two components: the tail latency a proxy adds to each hop, which compounds across a call chain, and the memory and CPU each proxy reserves, which multiplies by the number of workloads [11]. The two meshes sit at different points on this axis as a direct consequence of their data-plane choices. Linkerd's purpose-built proxy is engineered for a small footprint and typically adds less tail latency and far less memory per instance, while Envoy's generality carries a heavier cost that buys greater capability [5], [11]. Figure 2 illustrates the representative shape of this trade-off using publicly reported figures; exact numbers depend heavily on workload, configuration, and version, so the value of such comparisons is the relative pattern rather than any single measurement.

The performance discussion also motivates active research into reducing the mesh tax. Studies that dissect where mesh overhead actually goes find that a large share is spent in the proxy's connection handling and in the redirection machinery rather than in the security work itself, which points toward leaner data planes and kernel-assisted interception as the direction of improvement [7], [11], [16].

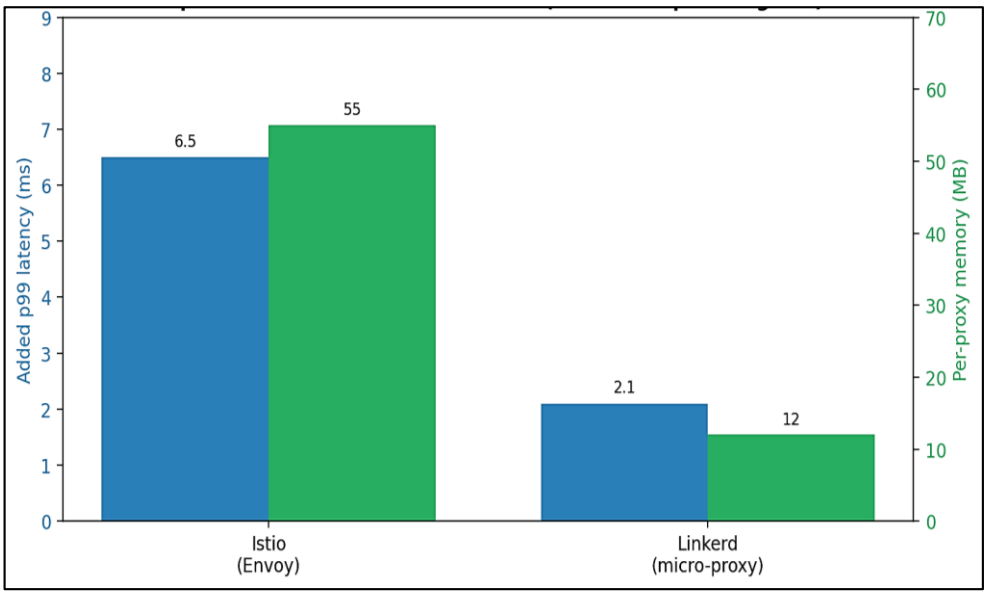

Fig 2: Representative data-plane overhead. Linkerd's purpose-built proxy is engineered for lower added latency and memory; Istio's Envoy trades a heavier footprint for greater configurability. Absolute values are workload-dependent.

V. MULTI-CLUSTER KUBERNETES

Spanning a mesh across several clusters raises problems a single cluster never poses. Services in one cluster must discover and reach services in another, the cryptographic identities issued in each cluster must be mutually trusted so that mTLS works across the boundary, and traffic must prefer local endpoints while failing over to remote ones when needed [4], [9]. Both meshes support multi-cluster topologies but with characteristically different emphases. Istio offers several models, including a shared control plane across clusters and a federation of independent control planes, with rich configuration for locality-aware routing and cross-cluster failover, reflecting its general philosophy of breadth and control [4], [10]. Linkerd approaches the same goal with a deliberately simpler model that links independent clusters and mirrors services across the boundary, keeping each cluster's mesh independently operable, consistent with its emphasis on simplicity [5].

Identity federation is the subtle hard part. For cross-cluster mTLS to function, the trust anchors of the participating clusters must be reconciled, whether by sharing a common root of trust or by establishing trust between distinct roots [7]. Standardised workload identity frameworks aim to make this federation portable across meshes and platforms, so that a workload's identity is meaningful regardless of which cluster issued it [7], [14]. Getting this right is what separates a genuine multi-cluster mesh from several single-cluster meshes that merely route to one another.

VI. THE SIDECAR-FREE TURN

The sidecar model, for all its elegance, has an inherent cost: one proxy per workload means overhead that scales with the number of workloads rather than with traffic, and an upgrade of the data plane means restarting every pod [11]. This has driven a recent architectural shift toward sidecar-free, or ambient, designs that move the common mesh functions out of per-pod proxies and into shared per-node components, reserving a heavier proxy only for workloads that need advanced layer-seven features [12]. Kernel-level programmability is central to this turn: by performing interception and some policy enforcement in the operating system kernel rather than in a user-space proxy, these designs aim to keep the security and observability benefits of a mesh while cutting the per-workload tax [7], [12]. Whether the ambient model supplants the sidecar or coexists with it, the direction of travel is clearly toward reducing the data-plane footprint that currently distinguishes the two meshes.

VII. OBSERVABILITY AND TRAFFIC MANAGEMENT IN PRACTICE

A. Telemetry Without Instrumentation

The clearest day-to-day benefit of a mesh, beyond security, is observability that costs the application nothing. Because every request passes through a proxy, the mesh can emit a consistent set of golden signals, request rate, error rate, and latency distribution, for every service in the fleet, in the same format regardless of the language each service is written in [8]. The same interception point injects and propagates the headers that distributed tracing depends on, so a request can be followed across service boundaries even when the developers never added tracing code, although the application must still forward those headers for traces to remain unbroken [8], [17]. This uniform, automatic telemetry is what makes a large microservice estate legible at all, and it is delivered as a side effect of the architecture rather than as a separate project.

B. Resilience and Progressive Delivery

The proxy is also the natural place to implement the resilience patterns that every distributed system needs and that were previously scattered through client libraries. Configured centrally and enforced uniformly, the data plane applies timeouts, bounded retries with retry budgets to prevent retry storms, and circuit breaking that sheds load from a failing dependency before it cascades [9], [18], [22]. The same weighted-routing capability that the control plane configures enables progressive delivery: a new version receives a small percentage of traffic as a canary, its golden signals are compared against the stable version, and the rollout proceeds or rolls back on the evidence, all without redeploying the client [4], [9]. Figure 3 shows the proxy applying these controls and splitting traffic between a stable and a canary version. Istio exposes a notably rich surface for this kind of traffic shaping through Envoy, while Linkerd offers a focused subset aligned with its simplicity-first philosophy [4], [5].

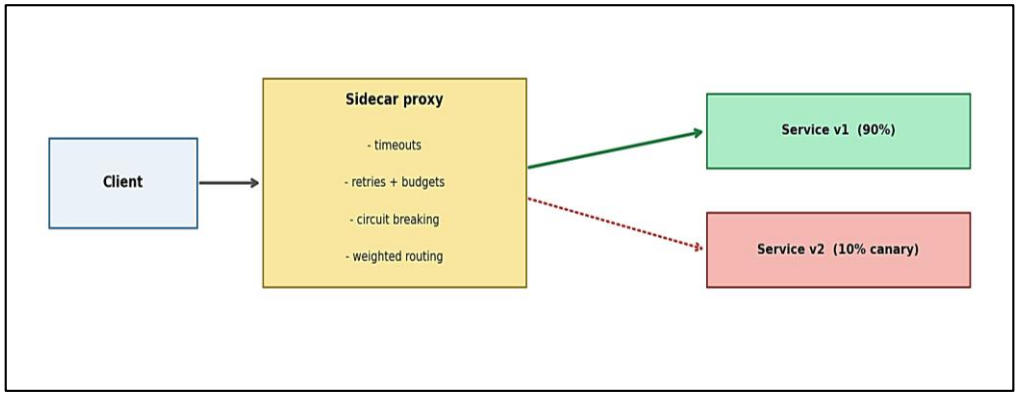

Fig 3: The data plane as a control point. The proxy applies timeouts, retries, and circuit breaking, and splits traffic by weight so a new version can be canaried and rolled back on measured signals, with no change to the client.

VIII. SECURITY BEYOND MUTUAL TLS

Automatic mutual TLS gives encryption and peer authentication, but a mesh's security model extends further into authorization and identity, and this is where the zero-trust promise is realised. Because each workload is issued a verifiable cryptographic identity by the control plane, the mesh can enforce policy in terms of who may call whom: a rule that only the checkout service may reach the payments service is expressed against identities and enforced at every proxy, independent of network location [4], [21]. This shifts access control from the perimeter, where a network address is a weak proxy for identity, to the workload, where identity is cryptographic and continuously verified, which is the essence of a zero-trust architecture [21]. The control plane also handles the unglamorous but critical task of issuing and rotating short-lived certificates automatically, so that credentials expire quickly and a compromised key is useful only briefly [3], [19]. Standardised identity frameworks make these identities portable, so that the same notion of a workload identity can be recognised across meshes, clusters, and even non-mesh systems, which is what allows the zero-trust model to extend beyond a single cluster boundary [14], [20].

IX. OPERATIONAL REALITY AND WHEN NOT TO USE A MESH

A service mesh is powerful infrastructure, and like all powerful infrastructure it has an operational cost that must be weighed honestly. Running the control plane is itself a distributed-systems responsibility, upgrades of the data plane traditionally require restarting every workload to replace its sidecar, and the abstraction the mesh introduces becomes another layer to understand when debugging a request that misbehaves [9], [11]. The per-proxy resource budget, multiplied across thousands of workloads, is a line item that has to be planned for, and the configuration surface, especially the broad one Istio exposes, is itself a source of operational risk if misapplied [11]. The mature judgement is that a mesh earns its cost only past a certain scale and complexity: a handful of services rarely justifies it, whereas a large polyglot estate with serious security and observability requirements often does [9], [15]. The sidecar-free architectures of Section VI are in large part a response to exactly these operational frictions, aiming to keep the benefits while shrinking the footprint and the upgrade burden [12]. Table 2 distils the criteria that, in practice, push a decision toward one mesh or the other once the decision to adopt a mesh at all has been made.

Table 2. Selection Criteria Once a Mesh Is Warranted

If the priority is...	Favors Istio	Favors Linkerd
Traffic-management richness	Yes (broad Envoy surface)	Adequate for common cases
Operational simplicity	Higher effort	Yes (opinionated defaults)
Per-proxy overhead budget	Heavier footprint	Yes (lightweight proxy)
Extensibility / custom filters	Yes (WASM, EnvoyFilter)	Limited by design
Multi-cluster routing control	Yes (multiple models)	Simpler linked model

X. CHOOSING, AND CONCLUSION

The comparison resists a single winner because the two meshes optimise for different things, and the right answer depends on the organisation rather than the technology. An organisation that needs fine-grained traffic control, deep extensibility, and a broad policy surface, and that has the operational maturity to run a complex control plane, will find Istio's generality worth its cost [4], [10]. An organisation that values operational simplicity, predictable low overhead, and a mesh that does the common things well with few knobs will be better served by Linkerd [5]. In multi-cluster deployments the same logic holds, with the added consideration of how much cross-cluster routing sophistication and identity-federation configurability the workload genuinely requires [9].

What both meshes share is more fundamental than what divides them. Each lifts the cross-cutting concerns of service-to-service communication out of application code and into a managed, language-independent layer secured by automatic mutual TLS [1], [3]. That architectural move, separating a data plane of proxies from a control plane of intent, is the durable contribution of the service mesh, and it is being refined rather than abandoned as the field moves toward leaner, kernel-assisted data planes [12]. The choice between Istio and Linkerd is, in the end, a choice between control surface and operational economy, and an engineering organisation should make it with that trade-off, not a feature checklist, in front of it.

REFERENCES

[1] W. Morgan, “What’s a service mesh? And why do I need one?,” Buoyant, Inc., Tech. Blog, 2017.
[2] B. Burns and D. Oppenheimer, “Design patterns for container-based distributed systems,” in Proc. 8th USENIX Workshop Hot Topics Cloud Comput. (HotCloud), 2016.
[3] B. Burns, B. Grant, D. Oppenheimer, E. Brewer, and J. Wilkes, “Borg, Omega, and Kubernetes,” Commun. ACM, vol. 59, no. 5, pp. 50–57, 2016.
[4] Istio Authors, “Istio: An open platform to connect, manage, and secure microservices,” Istio Project Documentation, 2023.
[5] Linkerd Authors, “Linkerd: An ultralight, security-first service mesh for Kubernetes,” Cloud Native Computing Foundation (CNCF), 2023.
[6] Envoy Project Authors, “Envoy proxy: Architecture overview,” Cloud Native Computing Foundation (CNCF), 2023.
[7] S. Ashok, P. B. Godfrey, and R. Mittal, “Leveraging service meshes as a new network layer,” in Proc. 20th ACM Workshop Hot Topics Netw. (HotNets), pp. 229–236, 2021.
[8] B. H. Sigelman et al., “Dapper, a large-scale distributed systems tracing infrastructure,” Google Tech. Rep. dapper-2010-1, 2010.
[9] W. Li, Y. Lemieux, J. Gao, Z. Zhao, and Y. Han, “Service mesh: Challenges, state of the art, and future research opportunities,” in Proc. IEEE Int. Conf. Service-Oriented Syst. Eng. (SOSE), pp. 122–127, 2019.
[10] L. Calcote and Z. Butcher, Istio: Up and Running. Sebastopol, CA, USA: O’Reilly Media, 2019.
[11] X. Zhu, G. She, B. Xue, Y. Zhang, Y. Zhang, X. Zou, X. Duan, P. He, A. Krishnamurthy, M. Lentz, D. Zhuo, and R. Mahajan, “Dissecting service mesh overheads,” arXiv preprint arXiv:2207.00592, 2022.
[12] Istio Authors, “Introducing ambient mesh: A sidecar-less data plane for Istio,” Istio Project Tech. Blog, 2022.
[13] J. Newman, Building Microservices: Designing Fine-Grained Systems, 2nd ed. Sebastopol, CA, USA: O’Reilly Media, 2021.
[14] SPIFFE Authors, “SPIFFE: Secure Production Identity Framework for Everyone,” Cloud Native Computing Foundation (CNCF), 2022.
[15] Cloud Native Computing Foundation, “CNCF annual survey,” CNCF, 2022.
[16] A. Sheikh, “Performance evaluation of service mesh data planes on Kubernetes,” in Proc. IEEE Int. Conf. Cloud Eng. (IC2E), pp. 1–8, 2021.
[17] World Wide Web Consortium, “Trace Context,” W3C Recommendation, 2020.
[18] M. T. Nygard, Release It! Design and Deploy Production-Ready Software, 2nd ed. Raleigh, NC, USA: Pragmatic Bookshelf, 2018.
[19] Istio Authors, “Istio security: Authentication and authorization policies,” Istio Project Documentation, 2023.
[20] SPIFFE Authors, “SPIRE: The SPIFFE Runtime Environment,” Cloud Native Computing Foundation (CNCF), 2022.
[21] S. Rose, O. Borchert, S. Mitchell, and S. Connelly, “Zero trust architecture,” NIST Special Publication 800-207, National Institute of Standards and Technology, Gaithersburg, MD, USA, 2020.
[22] M. Fowler, “CircuitBreaker,” martinfowler.com, 2014.