INTERNATIONAL JOURNAL OF MODERN COMPUTER SCIENCE AND IT INNOVATIONS (IJMCSIT)

eISSN: 3087-4289

Volume. 02, Issue. 10, pp. 64-73, October 2025"

A COMPARATIVE ANALYSIS OF SERVICE MESH PROXY ARCHITECTURES: FROM SIDECARS TO AMBIENT AND PROXYLESS MODELS IN CLOUD-NATIVE ENVIRONMENTS

Felicia S. Lee
Department of Computer Science, National University of Singapore, Singapore

Kofi A. Mensah
Department of Computer Science, National University of Singapore, Singapore

Article received: 21/08/2025, Article Revised: 18/09/2025, Article Accepted: 21/10/2025, Article Published: 23/10/2025

ABSTRACT

Purpose: The proliferation of cloud-native, microservices-based applications has established the service mesh as a critical infrastructure component for managing security, observability, and traffic. However, the foundational "sidecar" proxy model, while functionally rich, introduces significant performance overhead and operational complexity. This paper provides a critical, comparative analysis of the evolving service mesh data plane proxy architectures.

Methodology: This research employs a systematic review and qualitative comparative analysis of four distinct proxy models: (1) the traditional per-pod sidecar, (2) the application-embedded proxyless model, (3) the kernel-native eBPF-based model, and (4) the emerging disaggregated hybrid model, exemplified by Ambient Mesh. The analysis evaluates these models against key metrics: resource consumption, latency, security isolation, and operational transparency.

Findings: The analysis reveals a fundamental shift away from the "one-size-fits-all" sidecar. Proxyless models offer superior performance at the cost of application coupling. eBPF-based models provide kernel-native speed but face challenges in complex L7 policy enforcement. The disaggregated Ambient Mesh model, splitting L4 and L7 responsibilities, emerges as a compelling synthesis, aiming to reduce overhead significantly while retaining on-demand L7 capabilities.

Implications: A critical trade-off exists between the granular security isolation of the sidecar and the node-level security boundary of new models. This "blast radius" shift has profound implications for DevSecOps practices and the implementation of Zero Trust architectures. The findings suggest the future of the service mesh data plane is disaggregated, hybrid, and increasingly eBPF-native.

Keywords: Service Mesh, Cloud-Native, Microservices, Sidecar Proxy, Ambient Mesh, eBPF, Proxyless, Zero Trust Architecture.

INTRODUCTION

1.1 The Ascendance of Cloud-Native Applications and Microservices

The last decade has witnessed a paradigm shift in application architecture, moving from monolithic structures to distributed systems composed of microservices. This cloud-native approach, characterized by containerization, dynamic orchestration (primarily via Kubernetes), and a philosophy of immutable infrastructure, offers organizations unprecedented agility, scalability, and resilience. However, this architectural decomposition is not without its costs. By breaking a monolith into a distributed network of dozens or even hundreds of discrete services, developers are confronted with the inherent complexities of distributed

computing.

These challenges, famously articulated as the "fallacies of distributed computing," include unreliable networks, variable latency, and the complexities of securing inter-service communication. In a monolithic application, a function call is a simple, reliable in-memory operation. In a microservices architecture, the equivalent "function call" is now a network request, subject to partial failures, timeouts, and malicious interception. Managing this service-to-service communication logic—such as implementing retries, enforcing timeouts, encrypting traffic, and gathering metrics—directly within each application's code is untenable. It leads to code duplication, inconsistent implementation, and a tight coupling between business logic and network plumbing.

1.2 The Emergence of the Service Mesh

The service mesh emerged as a direct solution to this challenge. It is a dedicated, transparent infrastructure layer designed to manage, secure, and observe all service-to-service communication within a microservices environment. The core principle of a service mesh is to abstract the logic of network communication away from the application code. This allows development teams to focus on business logic, while platform or security teams manage the policies governing communication.

A service mesh is architecturally bifurcated into two distinct components: the Control Plane and the Data Plane. The control plane serves as the "brain" of the mesh. It provides a unified API for operators to define policies (e.g., "service A can only communicate with service B," "retry failed requests to service C three times") and aggregates telemetry from the data plane. It configures the data plane but does not sit in the request path itself.

The data plane, in contrast, is the "workhorse" of the mesh. It is composed of a network of intelligent proxies that are deployed alongside each service instance. These proxies intercept every network packet entering or leaving a service, enforcing the policies defined by the control plane. This is where security is enforced (e.g., mutual TLS), reliability is implemented (e.g., circuit breaking), and observability metrics are generated.

1.3 The Critical Role of the Data Plane Proxy

The implementation, deployment, and architecture of the data plane proxy are the central focus of this paper. The proxy is the concrete manifestation of the service mesh's policies. Its efficiency, security, and transparency dictate the overall performance and operational overhead of the entire mesh. How this proxy is deployed—its proxy model—is arguably the most critical architectural decision in a service mesh implementation.

The initial and still-dominant proxy model is the sidecar. In this model, a proxy is injected as a separate container (the "sidecar") into every application pod. This model provided the transparency and isolation that made service mesh viable at scale. However, this model is not the only option, and its drawbacks have become increasingly apparent.

1.4 Literature Gap and Problem Statement

The first generation of service mesh literature focused primarily on the control plane (e.g., comparing Istio, Linkerd, and Consul) and on making the conceptual argument for the necessity of a service mesh. In these discussions, the sidecar proxy model was largely assumed to be the default, and only, implementation. Consequently, a significant literature gap exists in the rigorous, comparative analysis of data plane proxy architectures themselves.

This gap has become critically important. The operational and performance overhead of the sidecar model—often called the "sidecar tax"—is now a well-documented pain point, leading to high resource consumption and increased latency [8]. In response, the cloud-native ecosystem has produced several innovative, competing data plane architectures. These include:

1. Proxyless Models, which embed mesh logic directly into application libraries [14].

2. eBPF-based Models, which move traffic management into the host's Linux kernel [9].

3. Disaggregated Hybrid Models, most recently exemplified by Istio's Ambient Mesh [7].

These new models make fundamentally different trade-offs between performance, security, and transparency. The introduction of Ambient Mesh in 2022 [7, 15], in particular, represents a direct challenge to the sidecar's dominance, yet a comprehensive academic comparison of these models, their technical underpinnings, and their security implications remains nascent. This paper seeks to fill that gap.

1.5 Research Objectives and Article Structure

This article provides a qualitative, comparative analysis of the primary service mesh proxy models. The research objectives are threefold:

1. To critically review and define the evolution of service mesh proxy models, from the foundational sidecar architecture to contemporary proxyless, eBPF-native, and the disaggregated ambient model.

2. To comparatively analyze these four models against a consistent set of key metrics: resource overhead (CPU/memory), latency, security posture (isolation vs.

blast radius), and operational complexity/transparency.

3. To evaluate the implications of these emerging models for the future of cloud-native systems, particularly in the context of DevSecOps pipelines [4] and the implementation of Zero Trust Architectures [5].

The remainder of this paper is structured as follows. Section 2.0 details the methodological framework, defining the scope of the analysis and providing a detailed technical taxonomy of the four proxy models. Section 3.0 presents the core comparative analysis, evaluating each model's benefits and drawbacks. Section 4.0 discusses the implications of these findings, proposing a "proxy model trilemma" and exploring the future of the service mesh data plane. Finally, Section 5.0 provides a conclusion and suggests directions for future research.

2.0 Methodological Framework

2.1 Scope and Boundaries of the Analysis

This study employs a qualitative, comparative methodology based on a systematic review of technical literature. The scope of this analysis is precisely bounded to the data plane architecture of service meshes. It is not an evaluation of specific, all-in-one service mesh products (e.g., Istio vs. Linkerd). Instead, it uses specific implementations (like Istio's sidecar, gRPC's proxyless library, Cilium's eBPF, and Istio's Ambient Mesh) as illustrative examples of the architectural models they represent.

The analysis is situated within the context of Kubernetes as the de facto container orchestration platform for cloud-native applications. The metrics used for comparison are qualitative but grounded in the technical documentation and industry analysis of these systems:

- Resource Overhead: The CPU and memory cost associated with the proxy model (e.g., per-pod, per-node, or per-application).
- Latency: The network delay introduced by the proxy's interception and processing of traffic.
- Security Posture: The model's default isolation boundary and its "blast radius" in a security compromise.
- Operational Complexity: The "Day 2" operational burden of managing, upgrading, and debugging the proxy infrastructure, including its transparency to application teams.

2.2 A Qualitative, Comparative Taxonomy of Proxy Models

The analytical framework for this paper is a taxonomy of four distinct proxy models. The data for this analysis is synthesized from foundational technical specifications, official project documentation [6, 7, 10, 11, 13, 14], peer-reviewed industry analysis [8, 9, 15], and relevant government technical publications, particularly the special publications from the National Institute of Standards and Technology (NIST) on microservices security and Zero Trust Architecture [2, 3, 4, 5].

This framework defines each model not just by its implementation, but by where the service mesh logic is executed: in a dedicated container, in the kernel, in the application, or in a disaggregated, tiered system.

2.3 Architectural Model Definitions

To ground the comparative analysis in Section 3.0, this section provides a detailed technical definition and description of the traffic flow for each of the four models.

2.3.1 Model 1: The Sidecar Proxy Model (Per-Pod)

The sidecar model is the original architecture for service meshes like Istio and Linkerd.

- Architecture: In this model, a fully-featured L4/L7 proxy (typically Envoy) is deployed as a separate container (the "sidecar") inside the same Kubernetes pod as the application container. The two containers share the same network namespace.
- Traffic Flow: When the pod is created, init containers configure the pod's iptables rules. These rules are set to redirect all outbound (egress) traffic from the application container to a specific port on the sidecar proxy. Similarly, all inbound (ingress) traffic is routed to the sidecar proxy first.

1. Egress: Application container sends a request (e.g., http://service-b). The packet hits the pod's iptables, which redirects it to localhost:15001 (the sidecar). The sidecar proxy processes the request (applies mTLS, gathers metrics, checks L7 policy), then sends it over the network to service-b's IP.

2. Ingress: Traffic from service-a arrives at the pod's IP. iptables redirects it to localhost:15006 (the sidecar). The sidecar terminates mTLS, validates L7 policy, and, if allowed, forwards the request to localhost:8080 (the application container).

- Security Model: The sidecar provides a very strong security boundary. Since each pod has its own dedicated proxy, identity and policy can be enforced at the individual pod level. The "blast radius" of a compromised proxy is limited to the single pod it resides in [2].

2.3.2 Model 2: The Proxyless Model (Application-Embedded)

The proxyless model eliminates the sidecar container entirely by integrating the service mesh logic directly into the application.

- Architecture: This model relies on "mesh-aware" application frameworks and libraries. Instead of a generic proxy, a specialized library (e.g., gRPC proxyless [14] or libraries for Spring [16]) is used by the application. This library communicates directly with the service mesh control plane to fetch policies and configuration.

- Traffic Flow: The traffic flow is simplified, as there is no iptables redirection or localhost network hop.

1. Egress: The application, using the proxyless library, makes a request. The library itself handles mTLS encryption, service discovery, load balancing, and policy enforcement in-memory before sending the packet directly over the network.

2. Ingress: An incoming mTLS connection is terminated directly by the application's embedded library, which also validates any relevant policies.

- Security Model: The security boundary shifts from the pod to the application itself. Security relies on the correct implementation and patching of the embedded library. This model breaks transparency, as the application is now explicitly "aware" of the mesh and coupled to its implementation.

2.3.3 Model 3: The Node-Level Proxy Model (e.g., eBPF-based)

This model also eliminates the sidecar, but instead of moving the logic up into the application, it moves it down into the host's Linux kernel. This approach is heavily championed by projects like Cilium [12] and discussed as a high-performance alternative [8, 9].

- Architecture: This model leverages eBPF (Extended Berkeley Packet Filter), a technology that allows for running sandboxed programs within the Linux kernel. A single, per-node agent installs eBPF programs at various kernel hooks (e.g., socket operations, traffic control).

- Traffic Flow: eBPF programs can intercept traffic at the earliest possible point, often bypassing much of the kernel's traditional networking stack (like iptables).

1. Egress: When the application container issues a send() system call, the eBPF program attached to that socket can immediately process the packet. It can apply L4 policies, perform load balancing, and encrypt data in the kernel before the packet is even passed to the network interface.

2. Ingress: An incoming packet is processed by an eBPF program on the network device, which can perform decryption, validation, and routing directly to the correct pod's socket, all within the kernel.

- Security Model: The security boundary becomes the node [12]. Policies are enforced at the kernel level for all pods on that host. This is extremely efficient but also carries a different risk profile: a vulnerability in the eBPF agent or the kernel could potentially compromise all traffic on the node. L7 policy enforcement is also significantly more complex to achieve in the kernel than in a user-space proxy like Envoy [9].

2.3.4 Model 4: The Disaggregated Hybrid Model (Ambient)

This is the newest model, introduced by the Istio project as "Ambient Mesh" [7, 15]. It is a hybrid, "sidecar-less" architecture that disaggregates the proxy's responsibilities into two separate components.

- Architecture: Ambient Mesh splits the L4 and L7 functionalities.

1. ztunnel (L4): A per-node agent (one per Kubernetes node) that acts as a secure L4 proxy. It handles mTLS, L4 authentication, and L4 telemetry for all pods on its node [10, 13].

2. waypoint (L7): A standard Envoy proxy, but instead of running per-pod, it is deployed per-service account (or namespace) on-demand. A waypoint proxy is only deployed if a service requires L7 policies (e.g., HTTP-based authorization, traffic splitting).

- Traffic Flow: The path depends on the policies in effect.

1. L4-Only Path: Pod A sends traffic to Pod B. The traffic is transparently intercepted by the node's ztunnel agent [10]. ztunnel establishes an mTLS connection with Pod B's ztunnel and forwards the (encrypted) traffic. This is a simple, two-hop (ztunnel-to-ztunnel) path.

2. L7 Path: Pod A sends traffic to Pod B, which has an L7 policy. The traffic is intercepted by Pod A's ztunnel. The ztunnel, seeing the L7 policy, forwards the traffic not to Pod B's ztunnel, but to Pod B's designated waypoint proxy (which runs in its own pod) [11]. The waypoint proxy terminates mTLS, applies the L7 policy, and then forwards the traffic to Pod B's ztunnel, which delivers it to the application.

- Security Model: This model is tiered [6]. The ztunnel provides a node-level L4 security boundary. The waypoint proxy provides a service-account-level L7 security boundary. This means a compromised ztunnel could potentially affect all L4 traffic on a node, while a compromised waypoint could affect all services sharing its identity [6].

3.0 Results: A Comparative Analysis of Proxy Models

This section presents the comparative analysis, structured by each model, evaluating them against the metrics defined in the methodology.

3.1 The "Classic" Sidecar Proxy: Foundations and Flaws

The sidecar model is the foundation upon which the modern service mesh was built. Its prevalence is a direct result of its functional richness and the strong security guarantees it provides, which align well with microservices security principles [2, 3].

3.1.1 Functional Richness and Security Isolation

The primary advantage of the sidecar is its "gold standard" implementation of security and L7 features. By deploying a full Envoy proxy alongside every application, operators gain:

- True Transparency: The application container is completely unaware of the mesh. It simply sends network requests to localhost or a service DNS name, and iptables handles the rest. This allows organizations to "onboard" existing applications to the mesh without a single line of code change.

- Rich L7 Policy: Because every request is processed by a full L7 proxy, operators can implement highly granular policies. This includes HTTP-based authorization (e.g., "allow GET requests to /api/v1 but deny POST"), request-based retries, fault injection, and complex traffic shifting (e.g., "send 5% of traffic to service-v2").

- Strong Isolation: The security boundary is the pod. As advocated in NIST SP 800-204A [2], this model provides a "zero trust" boundary for each microservice instance. A credential compromise in one pod's proxy only affects that single pod. This "blast radius" of one is the smallest possible in a Kubernetes environment.

3.1.2 The "Sidecar Tax": Performance and Resource Overhead

These benefits come at a steep and well-documented cost, often referred to as the "sidecar tax" [8]. This tax manifests in several ways:

- Resource Overhead: The most significant issue is resource consumption. If a cluster runs 1,000 application pods, it is also running 1,000 full-fledged Envoy proxy containers. Each of these sidecars requires its own CPU and memory allocation (requests and limits). In many cases, the resource consumption of the sidecar fleet can meet or even exceed the consumption of the applications themselves. This dramatically increases cluster infrastructure costs.

- Latency Impact: The traffic path for a sidecar-based request involves at least two localhost network hops: (1) app -> sidecar and (2) sidecar -> app. For a full request-response cycle between two services, this results in four local network traversals, adding measurable latency to every single call. For latency-sensitive applications, this "proxy tax" is unacceptable.

- Operational Complexity: Managing a fleet of sidecars is a significant "Day 2" operational challenge. Upgrading the service mesh requires a rolling restart of every application pod in the mesh to inject the new sidecar version, which can be highly disruptive. Furthermore, misconfigured resource limits can lead to the sidecar being OOM-killed (Out Of Memory) or CPU-throttled, which effectively takes the application pod offline even if the application itself is healthy. Finally, the complex iptables rules used for redirection can be brittle and difficult to debug.

3.2 The "Performance" Proxyless Model: Speed vs. Transparency

The proxyless model was developed as a direct response to the performance problems of the sidecar. It seeks to reclaim the "sidecar tax" by eliminating the proxy container entirely.

3.2.1 gRPC Proxyless: A Case Study [14]

The gRPC framework provides a canonical example [14]. The gRPC libraries in various languages (e.g., Java, Go, C++) were enhanced to be "mesh-aware." These libraries can natively parse configuration from service mesh control planes (like Istio), enabling them to perform client-side load balancing, service discovery, and security enforcement directly.

The performance gains are substantial. By removing the two localhost hops and the overhead of a separate container, latency is significantly reduced, approaching that of a simple, non-mesh gRPC call. Resource consumption per-pod also drops dramatically, as the mesh logic is just a library loaded into the application's existing memory space.

3.2.2 The Cost of Coupling

The primary drawback of the proxyless model is the complete loss of transparency. This trade-off is often unacceptable.

- Language/Framework Lock-in: This model only works if the application is written in a framework that has a "mesh-aware" library. A gRPC proxyless mesh [14] is useless for services written using REST, Thrift, or plain TCP. This creates a "two-tier" mesh: gRPC services get full-featured, high-performance integration, while all other services are either left out or must fall back to a

different model (like a sidecar), defeating the purpose of a unified mesh.

- Application Coupling: The application is now explicitly coupled to the service mesh. Developers must manage library dependencies and versions, and the application code is now responsible for network logic. This is precisely the problem the service mesh was invented to solve.

- Inconsistent Patching and Security: If a critical vulnerability is found in the mesh library, every single application must be re-compiled, re-tested, and re-deployed with the patched library. This is a massive operational burden compared to the sidecar model, where only the sidecar image needs to be updated in the pod template (followed by a rolling restart). This coupling makes fleet-wide security patching slow and difficult.

3.3 The "Kernel-Native" eBPF Model: A New Frontier [9]

The eBPF model offers a third, compelling path: achieving the transparency of the sidecar with the performance of the proxyless model by moving logic into the Linux kernel [9].

3.3.1 The Power of eBPF in Networking [8]

eBPF allows developers to attach small, sandboxed programs to kernel-level events, such as system calls or the arrival of network packets [8]. In the context of a service mesh, an eBPF-based agent (like Cilium) running on each node can manage pod traffic at a level below the pod's own network namespace.

This bypasses iptables entirely. An eBPF program can inspect a packet, identify its source and destination pods, apply L4 network policies, and even perform service-level load balancing, all within the kernel.

3.3.2 Benefits: Performance and Transparency

The benefits of this approach are profound:

- Near-Native Performance: Because the logic executes in the kernel, it is extremely fast. It avoids the user-space/kernel-space context switching of iptables and the multiple localhost hops of the sidecar.

- Complete Transparency: Like the sidecar, this model is 100% transparent to the application. The application container is completely unmodified and unaware of the mesh's existence.

- Reduced Resource Footprint: The resource cost is paid per-node (for the eBPF agent), not per-pod. This is a fixed cost that does not scale with the number of pods on the node, making it vastly more efficient in high-density environments than the sidecar model.

3.3.3 Challenges and Limitations

Despite its power, the eBPF-only model faces significant challenges, particularly at Layer 7:

- L7 Policy Complexity: eBPF is fundamentally a L3/L4 technology. While it excels at IP- and port-based rules, performing L7 inspection (e.g., parsing an HTTP header or a gRPC protobuf) is exceptionally complex within the kernel. It requires the eBPF program to reassemble TCP streams and understand application-layer protocols, which is difficult and can have performance implications.

- The TLS Problem: The biggest L7 challenge is TLS. If traffic is encrypted (as it should be for mTLS), an eBPF program in the kernel can only see opaque, encrypted bytes. It cannot enforce HTTP-level policies. This forces eBPF-based meshes to either terminate TLS in a user-space proxy (re-introducing a sidecar or a per-node proxy) or give up on L7 policy entirely.

- Security Threat Model [12]: The security model is different and, in some ways, more hazardous. A vulnerability in the eBPF agent or a malicious eBPF program loaded into the kernel could compromise the entire node, providing visibility into all network traffic and potentially bypassing all security policies [12]. This node-level "blast radius" is much larger than the sidecar's pod-level radius.

3.4 The "Disaggregated" Ambient Mesh Model: A New Synthesis? [7]

The Ambient Mesh model [7, 15] is a direct, pragmatic response to the challenges of the other three models. It acknowledges that the "one-size-fits-all" sidecar is inefficient. It also acknowledges that eBPF is excellent at L4 but struggles with L7 [9], and that proxyless models [14] break transparency.

Its solution is to disaggregate the proxy, providing a "pay-as-you-go" cost model for mesh features.

3.4.1 Architectural Deep Dive: ztunnel and Waypoint Proxies [13, 11]

The Ambient model is a two-component system that attempts to get the best of both eBPF and the traditional proxy:

1. ztunnel (The eBPF-powered L4 Mesh): The ztunnel [10, 13] is a per-node agent. It is responsible for the foundational part of the mesh: establishing a secure mTLS overlay. It intercepts all pod traffic (often using eBPF or a lightweight transparent proxy) and provides L4 authentication, authorization, and telemetry. For the majority of services that only need simple, secure transport, the ztunnel is all they will ever interact with.

This provides the per-node resource efficiency of the eBPF model.

2. waypoint (The On-Demand L7 Proxy): When a service requires L7 policy, the operator deploys a waypoint proxy [11] for that service's identity (e.g., its Kubernetes Service Account). This is a full-featured Envoy proxy, just like a sidecar. However, instead of running one-per-pod, it runs one-per-service-account, managing L7 traffic for all pods with that identity. The ztunnel is configured to intelligently redirect traffic destined for that service to its waypoint proxy first [11].

3.4.2 Addressing the Sidecar Tax

This disaggregated model directly addresses the "sidecar tax."

- Vast Resource Reduction: For a cluster where, for example, 80% of services only require mTLS (L4) and 20% require complex L7 rules, the resource cost is dramatically lower. The 80% of services use the efficient, shared, per-node ztunnel. The 20% use waypoint proxies, but even this cost is consolidated (e.g., 50 pods sharing one service account might only use 2-3 waypoint proxy replicas, instead of 50 sidecars).

- Simplified Pod Lifecycle: Application pods are just application pods. There is no injected container. This simplifies development, speeds up pod startup times, and makes mesh upgrades non-disruptive to applications. The ztunnel and waypoints can be upgraded independently as platform components.

3.4.3 The New Security Posture [6]

The most critical discussion around Ambient Mesh concerns its security posture, which is more nuanced than the sidecar model [6].

- The ztunnel Blast Radius: The ztunnel agent runs as a privileged DaemonSet on each node. If a ztunnel component is compromised, the attacker has a foothold on that node and could potentially intercept or manipulate all L4 traffic for all pods on that node. This is a larger blast radius than the sidecar. However, the ztunnel is designed to be a minimal, secure component, handling only mTLS and L4 forwarding, which reduces its attack surface compared to a full L7 proxy [13].

- The waypoint Blast Radius: If a waypoint proxy is compromised, the attacker could intercept and manipulate L7 traffic for all pods sharing that waypoint's identity. This is a larger blast radius than a single pod, but it is contained within a specific service identity boundary, which is still a core principle of Zero Trust [5].

- The Trade-off: Ambient Mesh trades the perfect pod-level isolation of the sidecar for dramatically better performance and operational simplicity [15]. The argument is that the sidecar's "perfect" isolation was already imperfect (a compromised application in the pod could attack the sidecar over localhost) and that a well-defined, identity-based boundary (the waypoint) combined with a secure node-level transport (the ztunnel) is a more pragmatic and efficient implementation of Zero Trust [5, 6].

4.0 Discussion and Implications

The analysis presented in Section 3.0 indicates that the service mesh data plane is in a period of rapid evolution, moving away from the monolithic sidecar. This shift is not merely technical; it has broad implications for how organizations balance performance, security, and operations.

4.1 Synthesizing the Findings: The Proxy Model Trilemma

The comparison of the four models reveals a fundamental "Proxy Model Trilemma." An ideal data plane would be:

1. High Performance (Low latency, low resource overhead)

2. Strong Security (Granular isolation, small blast radius)

3. Operationally Transparent (No application changes, easy to manage)

The analysis shows that no single model currently achieves all three. Instead, they represent different trade-offs:

- Sidecar Model: Prioritizes Strong Security (pod-level isolation) and Operational Transparency (no code changes), but sacrifices High Performance (the "sidecar tax").

- Proxyless Model: Prioritizes High Performance (no proxy), but sacrifices Operational Transparency (application coupling, language lock-in) and offers a weaker, application-level Security boundary.

- eBPF-Only Model: Prioritizes High Performance (kernel-native) and Operational Transparency (no code changes), but struggles with L7 Security/Features (the TLS problem, kernel complexity).

- Ambient Mesh Model: Attempts a new, dynamic balance. It provides High Performance for L4 workloads (via ztunnel) and Operational Transparency (no sidecar injection). It compromises on Strong Security at the L4 layer (moving from pod to node-level isolation) but re-introduces strong L7 isolation on-demand via the waypoint proxy.

This trilemma illustrates why the disaggregated model is so compelling. It allows operators to choose their point in the trilemma on a per-service basis, rather than forcing a single, cluster-wide compromise.

4.2 The Future of the Data Plane: Is the Sidecar Obsolete?

The evidence strongly suggests that the "one-size-fits-all" sidecar model is becoming obsolete. Its resource-intensive, per-pod deployment is an inefficient solution for the common case where many services only require L4 mTLS. The future of the service mesh data plane is unequivocally disaggregated and hybrid.

This future is enabled by eBPF. Technologies like eBPF [8, 9] are no longer just an alternative to the sidecar; they are becoming the foundation for the next generation of data planes. The ztunnel component of Ambient Mesh, for example, is a prime candidate for an eBPF-based implementation [10]. eBPF provides the high-performance, transparent traffic redirection needed to efficiently get traffic to the correct proxy, whether that's another ztunnel on a different node or a shared waypoint proxy.

The future architecture will likely look like Ambient Mesh: a highly efficient, eBPF-based L4 transport mesh for all services, with the option to "opt-in" to a shared, on-demand pool of L7 proxies (like waypoints) for services that require rich, application-aware policies.

4.3 Implications for DevSecOps and Zero Trust Architectures

This architectural shift has profound implications for both DevSecOps practices and the implementation of Zero Trust Architectures (ZTA).

4.3.1 Impact on DevSecOps

The sidecar model created a fuzzy line of responsibility. Is the sidecar's resource consumption the application team's problem or the platform team's? Who is responsible when the sidecar crashes?

- Simplifying Application Development: Sidecar-less models (eBPF and Ambient) create a much cleaner separation of concerns, aligning with modern DevSecOps goals [4]. Application teams deploy just their application. The service mesh becomes a true, transparent platform utility, managed entirely by the platform or SRE team. This simplifies CI/CD pipelines, as the pod definition is simpler and application startup is faster.

- Shifting Platform Responsibility: The responsibility for the mesh shifts squarely to the platform team. They are now managing per-node agents (ztunnels) and shared waypoint proxies. This is a more centralized and, arguably, more efficient management model. However, it also raises the stakes. A misconfiguration of a node-level ztunnel or a shared waypoint proxy can now affect many services, not just one pod. This aligns with the "platform engineering" model of providing common, shared services.

4.3.2 Re-evaluating Zero Trust Architectures [5]

The sidecar model was a near-perfect implementation of the "micro-segmentation" tenet of Zero Trust: assume the network is hostile, and enforce policy at the most granular boundary possible (the pod). The new models challenge this implementation.

- From Per-Pod to Per-Identity: Ambient Mesh forces a subtle but important shift in ZTA thinking: from per-pod segmentation to per-identity segmentation [6]. The waypoint proxy enforces L7 policy at the service-account level, not the pod level. This aligns with NIST's guidance on ABAC (Attribute-Based Access Control) [3] and ZTA [5], where the identity of the service (its attributes) is the core of the policy, not its ephemeral location in a specific pod.

- Is Node-Level mTLS "Zero Trust" Enough? The Ambient model proposes that for many services, a node-to-node mTLS connection (via ztunnel) is a "good enough" implementation of Zero Trust transport security. This is a pragmatic compromise. It secures the "untrusted" network between nodes, but it assumes the node itself is a trusted boundary. This is a significant departure from the classic sidecar, which trusted nothing outside the pod's network namespace. This new model places a much higher security burden on securing the node and the ztunnel agent itself.

4.4 Limitations of this Analysis

This analysis, while comprehensive, has several limitations that provide fertile ground for future research:

1. Qualitative Nature: This is a qualitative, comparative analysis based on technical documentation and industry reports. It lacks quantitative, empirical benchmarking data. The true performance (latency, CPU, memory) of these four models under various load profiles and cluster sizes can only be determined through rigorous, controlled experimentation.

2. Rapidly Evolving Field: The cloud-native landscape, particularly eBPF [9] and Ambient Mesh [7, 15], is evolving at an exceptional pace. Ambient Mesh is still considered experimental in Istio, and its design may change. The L7 capabilities of eBPF-based solutions are also improving monthly. This analysis represents a snapshot in time.

3. Implementation-Specific Nuances: This paper

intentionally generalized the models. In practice, specific vendor implementations (e.g., Cilium's eBPF, Linkerd's "micro-proxy") will have unique features and trade-offs that are not fully captured in this high-level architectural comparison.

4.5 Conclusion and Future Research Directions

This paper has charted the evolution of the service mesh data plane, from the foundational sidecar model to the emerging proxyless, eBPF-native, and disaggregated ambient architectures. The central finding is that the "one-size-fits-all" sidecar, while pioneering, is no longer the optimal solution. Its "sidecar tax" in performance and resource consumption has driven the development of new models that make different, and often more efficient, trade-offs.

The disaggregated hybrid model, exemplified by Ambient Mesh [7], represents the next logical synthesis. It combines the per-node efficiency of eBPF-based L4 transport with the on-demand, identity-based L7 policy enforcement of traditional proxies. This shift from a per-pod to a per-node and per-identity model has profound implications, simplifying operations for application teams while forcing a re-evaluation of Zero Trust security boundaries from the pod to the node and service identity.

The future of the service mesh is disaggregated. The "sidecar-less" future is not "proxyless"—it is "right-sized." It is a future where proxies are not monolithically injected everywhere, but are applied intelligently, efficiently, and transparently where and when they are needed.

Future research should focus on three critical areas:

1. Quantitative Benchmarking: Longitudinal, empirical performance studies are needed to benchmark sidecar, ambient, and eBPF-only models across multiple axes (latency, resource usage, scalability) under realistic, high-throughput workloads.

2. Security Threat Modeling: In-depth security analysis, including penetration testing, of the "blast radius" of compromised ztunnel and waypoint proxies is essential to validate the new security posture of ambient models [6, 12].

3. Operational Usability: Studies are needed on the "Day 2" operational complexity of managing these new disaggregated data planes. While they simplify the application pod, do they introduce new, complex failure modes for the platform team?

References

1. Wikipedia (2024) OSI Model. Available at https://en.wikipedia.org/wiki/OSI_model
2. Chandramouli R, Butcher Z (2020) Building Secure Microservices-based Applications Using Service-Mesh Architecture. (National Institute of Standards and Technology, Gaithersburg, MD), NIST Special Publication (SP) NIST SP 800-204A. https://doi.org/10.6028/NIST.SP.800-204A
3. Chandramouli R, Butcher Z, Aradhna C (2021) Attribute-based Access Control for Microservices-based Applications using a Service Mesh. (National Institute of Standards and Technology, Gaithersburg, MD), NIST Special Publication (SP) NIST SP 800-204B. https://doi.org/10.6028/NIST.SP.800-204B
4. Chandramouli R (2022) Implementation of DevSecOps for a Microservices-based Application with Service Mesh. (National Institute of Standards and Technology, Gaithersburg, MD), NIST Special Publication (SP) NIST SP 800-204C. https://doi.org/10.6028/NIST.SP.800-204C
5. Chandramouli R, Butcher Z (2023) A Zero Trust Architecture Model for Access Control in Cloud-Native Applications in Multi-Cloud Environments. (National Institute of Standards and Technology, Gaithersburg, MD), NIST Special Publication (SP) NIST SP 800-207A. https://doi.org/10.6028/NIST.SP.800-207A
6. Zero-Trust Architecture in Java Microservices. (2025). International Journal of Networks and Security, 5(01), 202-214. https://doi.org/10.55640/ijns-05-01-12
7. Jackson E, Kohavi Y, Pettit J, Posta C (2022) Ambient Mesh Security Deep Dive. (Istio) Available at https://istio.io/latest/blog/2022/ambient-security/
8. Howard J, Jackson EJ, Kohavi Y, Levine I, Pettit J, Sun L (2022) Introducing Ambient Mesh. (Istio) Available at https://istio.io/latest/blog/2022/introducing-ambient-mesh/#what-about-security
9. Turner M (2022) eBPF and Sidecars - Getting the Most Performance and Resiliency out of the Service Mesh. (Tetrate) Available at https://tetrate.io/blog/ebpf-and-sidecars-getting-the-most-performance-and-resiliency-out-of-the-service-mesh/
10. Graf T (2021) How eBPF will solve Service Mesh - Goodbye Sidecars. (Isovalent) Available at https://isovalent.com/blog/post/2021-12-08-ebpf-servicemesh/
11. Chandra Jha, A. (2025). VXLAN/BGP EVPN for Trading: Multicast Scaling Challenges for Trading Colocations. International Journal of Computational

and Experimental Science and Engineering, 11(3). https://doi.org/10.22399/ijcesen.3478

12. Song J (2022) Transparent Traffic Intercepting and Routing in the L4 Network of Istio Ambient Mesh. (Tetrate) Available at https://tetrate.io/blog/transparent-traffic-intercepting-and-routing-in-the-l4-network-of-istio-ambient-mesh/

13. Song J (2022) L7 Traffic Path in Ambient Mesh. (Tetrate) Available at https://tetrate.io/blog/l7-traffic-path-in-ambient-mesh/

14. Cilium (2024) Threat Model — Cilium 1.15.6 documentation. (Cilium) Available at https://docs.cilium.io/en/stable/security/threat-model/

15. Istio (2024) Ambient mode overview: ztunnel. Available at https://istio.io/latest/docs/ambient/overview/#ztunnel

16. Landow S (2021) gRPC Proxyless Service Mesh. (Istio) Available at [suspicious link removed]

17. Butcher Z (2024) Ambient Mesh: What you need to know about this experimental new deployment model for Istio Available at https://tetrate.io/blog/ambient-mesh-what-you-need-to-know-about-this-experimental-new-deployment-model-for-istio/

18. Spring (2024) Spring Framework Available at https://spring.io/projects/spring-framework