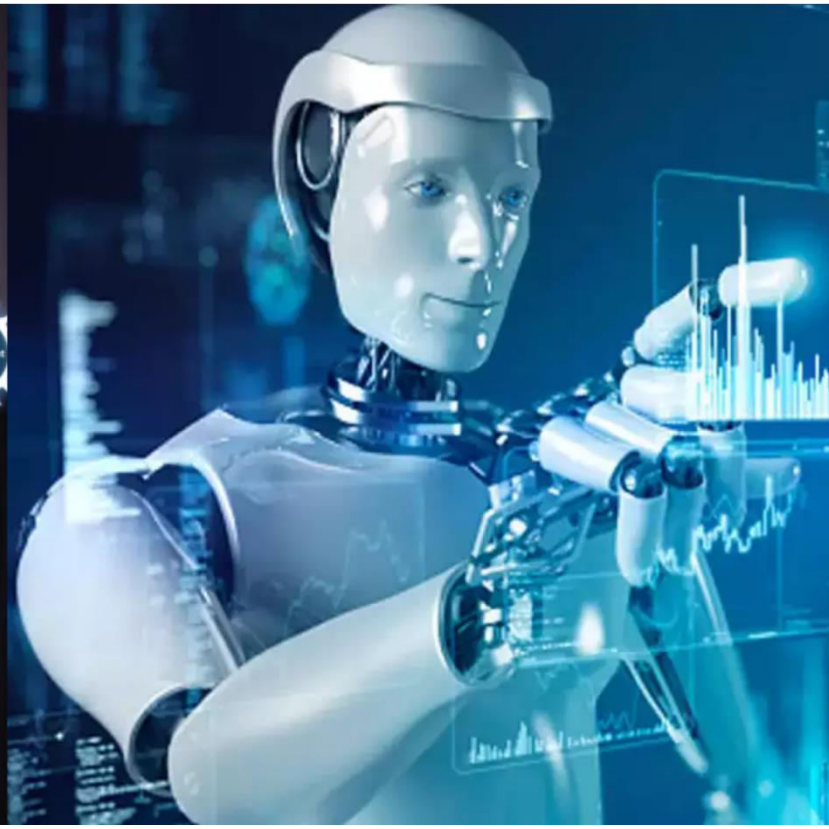
IJIRSET
International Journal of Innovative Research in Science | Engineering | Technology

ISSN(O): 2319-8753
ISSN(P): 2347-6710

International Journal of Innovative Research in Science Engineering and Technology (IJIRSET)

(A Monthly, Peer Reviewed, Refereed, Scholarly Indexed, Open Access Journal)



Impact Factor: 8.699

Volume 15, Issue 2, February 2026

International Journal of Innovative Research of Science, Engineering and Technology (IJIRSET)

Impact Factor 8.699

|www.ijirset.com |A Monthly, Peer Reviewed & Refereed Journal| e-ISSN: 2319-8753| p-ISSN: 2347-6710|

Volume 15, Issue 2, February 2026

|DOI: 10.15680/IJIRSET.2026.1502106|

eBPF-Powered Kubernetes Observability: Replacing Sidecar Proxies at Scale

Rohit Reddy

DevOps / Cloud Engineer, USA

ABSTRACT:

KEY INSIGHT Replacing sidecar proxies with kernel-level eBPF data planes substantially reduces latency, memory consumption, and operational complexity for service-mesh and observability workloads at scale - but the transition demands kernel-version discipline, deep platform-team expertise, and an honest accounting of where the eBPF approach is not yet a complete substitute for userspace proxies.

The sidecar-proxy pattern, popularized by service-mesh implementations over the past decade, solved an important problem: it gave operators a uniform place to enforce policy, terminate mutual TLS, collect L7 telemetry, and shape traffic without modifying application code. The pattern's costs, however, have become difficult to ignore at fleet scale. Every pod carries a proxy. Every request crosses additional userspace boundaries. Every node carries hundreds of megabytes of redundant infrastructure. Every configuration change ripples across every sidecar. The result is a tax on latency, resource consumption, and operational complexity that grows linearly with the cluster's workload count.

eBPF - the in-kernel virtual machine that runs verified, sandboxed programs attached to Linux kernel hook points - offers a structurally different approach. Network policy, traffic shaping, observability instrumentation, and security enforcement can all be implemented as kernel-resident programs that run once per node rather than once per pod, operate on packets and syscalls without crossing kernel boundaries, and present a uniform inspection point for the entire workload on a node. This article presents an end-to-end treatment of eBPF for Kubernetes observability and service-mesh data planes: the underlying mechanism, the reference architecture, the empirical performance characteristics, the operational risks, and the practical roadmap for replacing sidecar-based infrastructure with eBPF-based equivalents at production scale.

KEYWORDS: eBPF · Kubernetes · service mesh · sidecar · observability · Cilium · kernel tracing · cloud native

I. INTRODUCTION

The first decade of mainstream Kubernetes coincided with the rise of the sidecar proxy as the canonical solution to service-mesh and uniform-observability problems. A small handful of proxy implementations - Envoy foremost among them - became the de facto means by which platform teams added mutual TLS, traffic policy, retry logic, circuit breaking, and L7 telemetry to applications they did not control. The pattern's appeal was substantial: it was transparent to the application, it decoupled operational concerns from product code, and it produced a clear interface for policy and telemetry. The pattern was, for its moment, exactly the right answer.

The moment has passed. At fleet scale - thousands of nodes, tens of thousands of pods, millions of requests per second - the sidecar pattern produces costs that outweigh its benefits along several axes simultaneously. The per-pod memory consumption of a modern proxy can exceed the memory consumption of the application it is supposed to serve. The extra userspace hop adds milliseconds of latency to every request, and the noisy interaction between the application and the proxy adds tail latency that is difficult to characterize. The fleet-wide propagation of a configuration change involves orchestrating a rolling restart of every workload. And the integration burden - ensuring that every team's deployment manifest, scaling policy, security context, and resource budget account for the sidecar - consumes platform-team effort that scales with the size of the engineering organization.

International Journal of Innovative Research of Science, Engineering and Technology (IJIRSET)

Impact Factor 8.699

|www.ijirset.com |A Monthly, Peer Reviewed & Refereed Journal| e-ISSN: 2319-8753| p-ISSN: 2347-6710|

Volume 15, Issue 2, February 2026

|DOI: 10.15680/IJIRSET.2026.1502106|

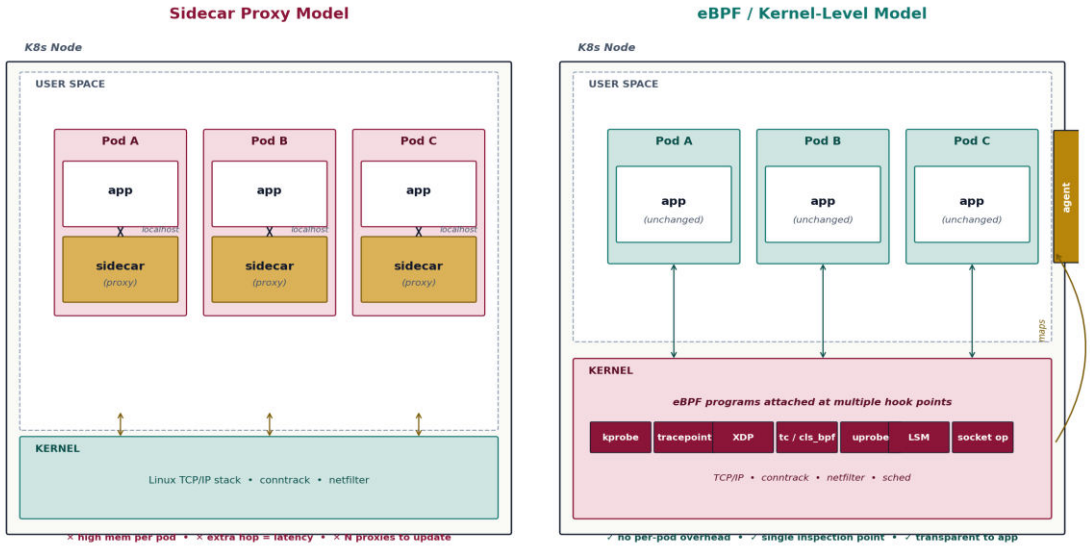


Figure 1. Side-by-side comparison of the sidecar proxy model and the eBPF kernel-level model. The eBPF model removes per-pod proxies in favor of a single per-node agent that programs kernel hooks.

1.1 The Shift to Kernel-Level Data Planes

eBPF presents a different model for the same set of problems. Rather than running a userspace proxy alongside every workload, eBPF allows the operator to attach verified, sandboxed programs to kernel hook points and have those programs implement networking, observability, and security functions in-kernel. The unit of deployment is the node rather than the pod. The path of a network packet does not cross additional userspace boundaries. The cost scales with the number of nodes, not the number of workloads. The control plane updates kernel maps rather than orchestrating proxy restarts.

The technology has matured rapidly over the past several years. The set of kernel hook points available to eBPF programs has expanded across networking, security, tracing, and performance subsystems. The compiler toolchain has consolidated around LLVM with consistent ABI guarantees. The verifier has grown substantially more capable, raising the complexity ceiling for production programs by orders of magnitude. Compile-Once Run-Everywhere relocation has substantially reduced the fragility of programs across kernel versions. The result is a technology that, while still demanding of operator expertise, is now production-credible for the data-plane workloads that have historically been the province of userspace proxies.

1.2 Scope and Position

This article focuses on the application of eBPF to two related Kubernetes platform concerns: comprehensive observability - metrics, traces, logs, profiles, and network flows - and service-mesh data planes that have historically been implemented through sidecar proxies. It does not attempt to survey every possible use of eBPF in operations, although many of the patterns described generalize naturally to host-based security, performance profiling, and network policy enforcement. It treats eBPF and userspace proxies as complementary tools rather than as adversaries: the architectures discussed routinely combine eBPF data planes for the bulk of the traffic with selective userspace proxies for the narrow workload classes where L7 features remain beyond the practical reach of in-kernel implementation.

1.3 Contributions

This article makes five contributions to the practitioner literature:

- A reference architecture for eBPF-based Kubernetes observability, decomposed into kernel, agent, and aggregation layers with stable interfaces.
- A detailed treatment of the eBPF hook taxonomy and the specific hooks relevant to service-mesh data-plane implementation.

International Journal of Innovative Research of Science, Engineering and Technology (IJIRSET)

Impact Factor 8.699

|www.ijirset.com |A Monthly, Peer Reviewed & Refereed Journal| e-ISSN: 2319-8753| p-ISSN: 2347-6710|

Volume 15, Issue 2, February 2026

|DOI: 10.15680/IJIRSET.2026.1502106|

- A quantitative analysis of the latency, throughput, and resource characteristics of eBPF data planes compared with traditional sidecar and ambient-mesh alternatives.
- A pragmatic catalog of the operational risks of production eBPF deployment and the controls that mitigate each.
- A sixteen-week rollout plan for replacing sidecar-based infrastructure with eBPF equivalents in a large enterprise cluster fleet.

II. BACKGROUND: EBPF MECHANISM AND LIFECYCLE

2.1 What eBPF Is

eBPF - the extended Berkeley Packet Filter - is a Linux kernel facility that permits the safe execution of operator-supplied programs at well-defined hook points throughout the kernel. The mechanism originated as a generalization of the packet-filtering capability that gave the technology its name, but it has expanded far beyond its original scope. Modern eBPF programs run not only on packet boundaries but on syscall entries and exits, on scheduler events, on file operations, on kernel function entries and returns, on user-space probes, and on many other classes of kernel event.

Three properties make eBPF distinctively useful for production infrastructure. First, the programs are verified before they are admitted to the kernel: the verifier proves that they will terminate, will not access memory outside their permitted scope, and will not produce undefined behavior. Second, the programs are JIT-compiled to native instructions after verification, producing performance comparable to native kernel code. Third, the programs can communicate with userspace through eBPF maps - typed shared data structures - allowing rich control-plane interaction without compromising the kernel's safety properties.

2.2 Program Lifecycle

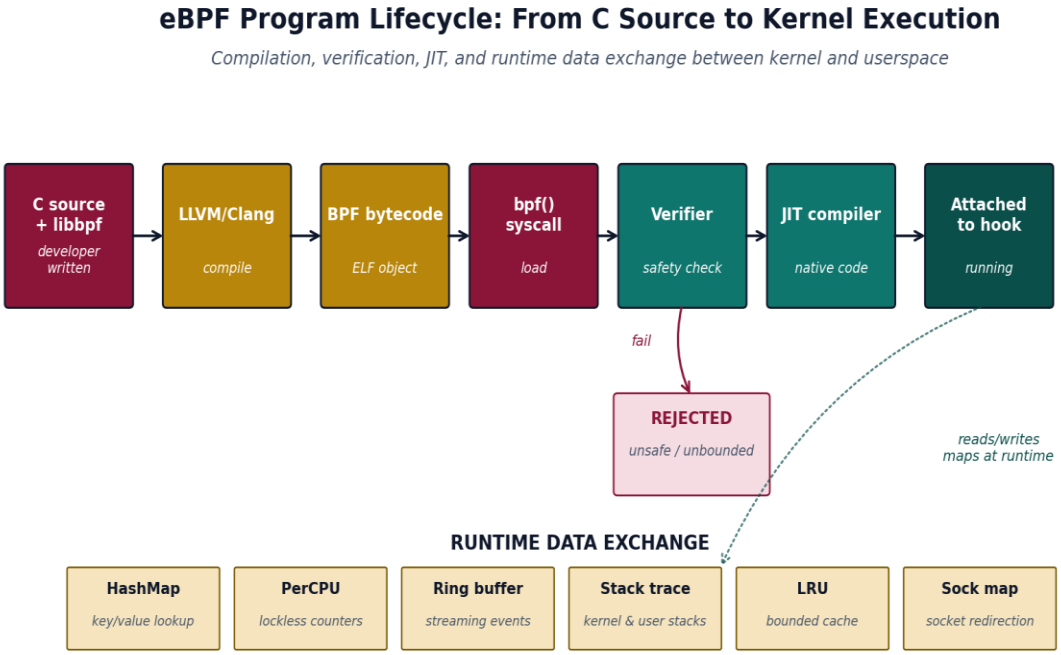


Figure 2. eBPF program lifecycle from C source through compilation, verification, JIT, and attachment, with runtime data exchange via typed maps.

Figure 2 traces the lifecycle of a typical eBPF program. The developer authors the program in restricted C, building against the libbpf library that provides ergonomic wrappers around the underlying bpf() syscall. LLVM with the BPF backend compiles the source into a BPF object file containing bytecode and metadata. The userspace loader invokes the bpf() syscall to submit the bytecode to the kernel. The verifier inspects the program for safety and either rejects it or

International Journal of Innovative Research of Science, Engineering and Technology (IJIRSET)

Impact Factor 8.699

|www.ijirset.com |A Monthly, Peer Reviewed & Refereed Journal| e-ISSN: 2319-8753| p-ISSN: 2347-6710|

Volume 15, Issue 2, February 2026

|DOI: 10.15680/IJIRSET.2026.1502106|

admits it for execution. Admitted programs are JIT-compiled to native instructions and attached to one or more hook points, after which they execute on every event that fires the hook.

At runtime the program reads and writes data through eBPF maps, which are typed shared structures - hash tables, arrays, per-CPU counters, ring buffers, stack-trace collectors, and so on - that span the kernel/userspace boundary. The agent in userspace consumes the contents of these maps to produce telemetry, drive control decisions, and inform observability backends. The maps themselves are the principal interface between the kernel-resident eBPF programs and the userspace platform that orchestrates them.

2.3 The Verifier and the Complexity Ceiling

The verifier is the single feature most responsible for eBPF's production credibility, and its limitations are the single feature most responsible for the discipline required to use eBPF well. The verifier explores the program's control-flow graph symbolically and proves that every path terminates, that every memory access is within bounds, and that no unprivileged operation is performed. Programs that the verifier cannot prove safe are rejected outright; there is no production path for unverified eBPF code.

Across recent kernel versions the verifier has grown markedly more capable. The permitted instruction count has risen from 4,096 to one million; bounded loops and BPF-to-BPF function calls are now supported; the verifier's path-exploration engine has grown more efficient. But the underlying constraint remains: every production eBPF program must fit inside what the verifier on the deployed kernel version is able to analyze. This places a real ceiling on the complexity of programs that can be deployed.

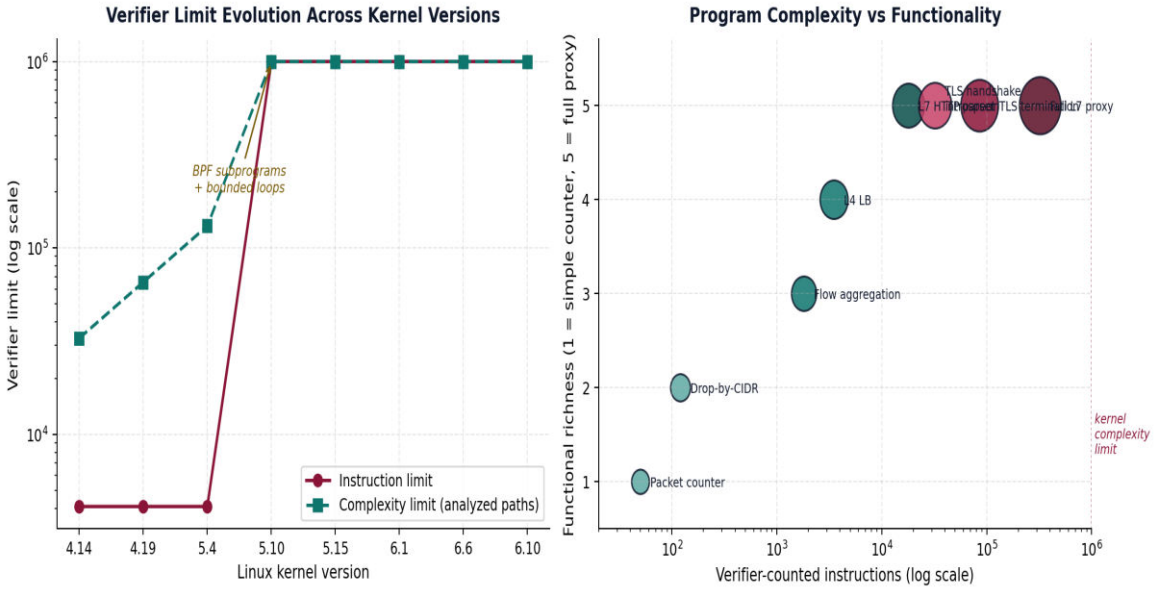


Figure 3. Verifier limits across kernel versions (left) and the relationship between program complexity and functionality (right). Full L7 proxying remains out of practical reach.

WARNING Programs that pass verification on the build host may fail verification on a target host running an older kernel. A continuous compatibility matrix across the supported kernel versions is non-negotiable for production deployment.

2.4 The Hook Point Taxonomy

eBPF programs attach to kernel hook points, and the catalog of available hooks is the catalog of available functions. Hooks fall into four broad categories: networking, tracing, security, and performance. Each category includes both specific, narrow hooks for individual subsystem events and broader, generalized hooks that allow programs to attach to arbitrary kernel functions. Figure 4 presents the taxonomy in detail.

International Journal of Innovative Research of Science, Engineering and Technology (IJIRSET)

Impact Factor 8.699

|www.ijirset.com |A Monthly, Peer Reviewed & Refereed Journal| e-ISSN: 2319-8753| p-ISSN: 2347-6710|

Volume 15, Issue 2, February 2026

|DOI: 10.15680/IJIRSET.2026.1502106|

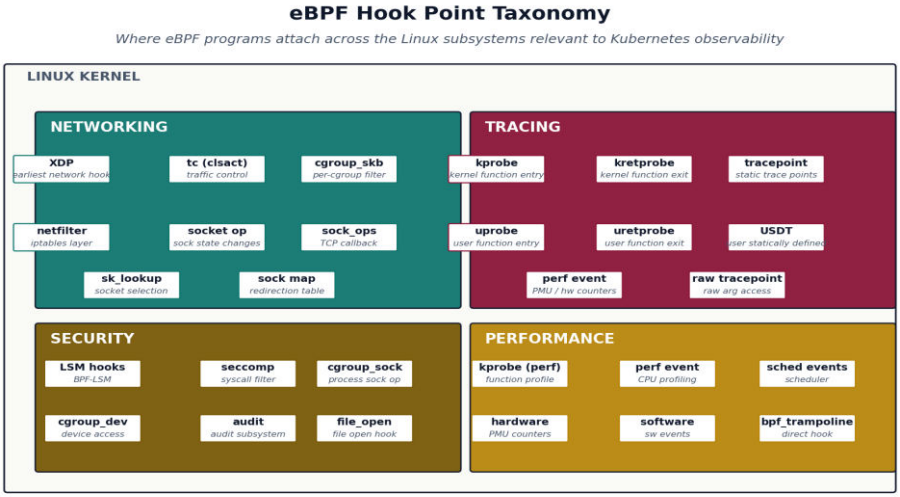


Figure 4. Taxonomy of eBPF hook points across Linux networking, tracing, security, and performance subsystems. The diversity of hooks is what makes eBPF a general-purpose platform rather than a single-feature tool.

Networking hooks dominate the data-plane discussion. XDP runs at the earliest possible point in packet processing, before the kernel has allocated socket buffers; it is the canonical hook for high-performance packet filtering and load-balancing. tc and cls_bpf attach to traffic-control classifiers and run on every ingress and egress packet, providing the natural attachment point for Kubernetes network policy and routing. Socket-operation hooks run on TCP state changes and provide the substrate for connection tracking and socket-level redirection.

Tracing hooks are central to the observability story. kprobes attach to kernel function entry points and provide access to function arguments and call sites. Tracepoints are static, well-defined trace points authored by kernel developers and offer a more stable ABI than kprobes. uprobes attach to userspace functions and make application-level tracing possible without modifying application code. raw tracepoints provide low-overhead access to event arguments and are the preferred interface for high-frequency tracing.

III. REFERENCE ARCHITECTURE FOR EBPF OBSERVABILITY

An eBPF-based observability platform is decomposed into three layers: the kernel layer where programs attach to hooks and gather raw events; the agent layer that runs once per node, reads from eBPF maps, enriches events with Kubernetes metadata, and exports telemetry; and the aggregation layer that collects from agents across the cluster and feeds downstream backends. Each layer has a distinct responsibility and a stable interface to its neighbors.

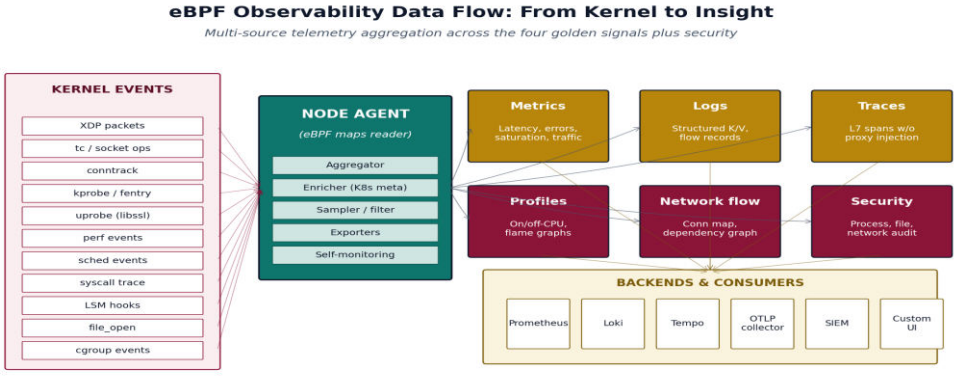


Figure 5. Observability data flow from kernel events through the per-node agent into the six telemetry pillars and downstream backends.

International Journal of Innovative Research of Science, Engineering and Technology (IJIRSET)

Impact Factor 8.699

|www.ijirset.com |A Monthly, Peer Reviewed & Refereed Journal| e-ISSN: 2319-8753| p-ISSN: 2347-6710|

Volume 15, Issue 2, February 2026

|DOI: 10.15680/IJIRSET.2026.1502106|

3.1 The Kernel Layer

The kernel layer is where the work happens. eBPF programs attached to networking hooks track every flow as it transits the node. Programs attached to tracing hooks fire on system calls, function entries, and tracepoints relevant to application behavior. Programs attached to security hooks audit file access, process execution, and network connections. The collective effect is that the node becomes self-instrumenting: every signal that the platform team wishes to observe is captured at its origin, in-kernel, without requiring application changes or additional userspace processes.

The discipline of the kernel layer is in keeping each program small and focused. A single eBPF program is bounded by the verifier's complexity limits; a system that requires elaborate logic must distribute that logic across multiple cooperating programs. The platform team must therefore design the kernel layer as a set of composable building blocks rather than as a small number of monolithic programs.

3.2 The Agent Layer

The agent runs once per node, typically as a DaemonSet, and is the single piece of userspace infrastructure that the eBPF-based platform requires. It is responsible for loading and attaching the eBPF programs, reading from the maps that those programs populate, enriching the raw events with Kubernetes metadata (the pod, service, namespace, and label context that gives a flow its operational meaning), filtering and sampling to bound the volume of exported telemetry, and exporting the result to whichever backends the platform supports.

FIELD NOTE The agent is the single highest-leverage component in the platform. A bug in the agent affects every workload on the node. Run the agent on the same release discipline as the kernel: pre-production verification, canary, gradual rollout, and explicit rollback capability.

3.3 The Aggregation Layer

The aggregation layer collects telemetry from agents across the fleet and routes it to the appropriate backends. In well-designed deployments the aggregation layer is the OpenTelemetry collector or an equivalent, and the agents export in OTLP format. This preserves substrate independence: the backends that consume the telemetry can change without requiring changes to the agent or to the kernel programs. The aggregation layer also performs cluster-wide tasks that the per-node agent cannot: deduplication of cross-node flow records, computation of service-level aggregates, and resolution of cross-node spans into coherent traces.

3.4 The Six Telemetry Pillars

An eBPF observability platform naturally produces six categories of telemetry, extending the familiar four golden signals with network-flow and security events:

- **Metrics.** Latency histograms, error counts, throughput, and saturation indicators, computed at fine temporal granularity from packet-level kernel events.
- **Logs.** Structured records of significant events: connection establishment and teardown, policy decisions, security audit events. Authored as key-value records, not free-form text.
- **Traces.** L7 spans captured at packet boundaries with selective userspace assistance for protocols requiring deep parsing. Traces are tied to existing application-level traces through context propagation.
- **Profiles.** On-CPU and off-CPU profiling, captured continuously through perf events and rendered as flame graphs. Profiles are correlated with application metrics for root-cause investigation.
- **Network flows.** Per-connection records with five-tuple plus Kubernetes context. Aggregated into a dependency graph that shows the runtime topology of the cluster.
- **Security events.** Process execution, file access, network connection, and policy decision records. Suitable for ingestion into a SIEM and for runtime anomaly detection.

IV. SERVICE MESH DATA PLANES COMPARED

Service-mesh architectures have evolved through three identifiable generations: the traditional sidecar pattern, the ambient pattern that splits L4 from L7 responsibilities, and the kernel-level pattern that pushes the data plane into eBPF. Each generation makes different trade-offs across resource efficiency, feature coverage, and operational complexity.

International Journal of Innovative Research of Science, Engineering and Technology (IJIRSET)

Impact Factor 8.699

|www.ijirset.com |A Monthly, Peer Reviewed & Refereed Journal| e-ISSN: 2319-8753| p-ISSN: 2347-6710|

Volume 15, Issue 2, February 2026

|DOI: 10.15680/IJIRSET.2026.1502106|

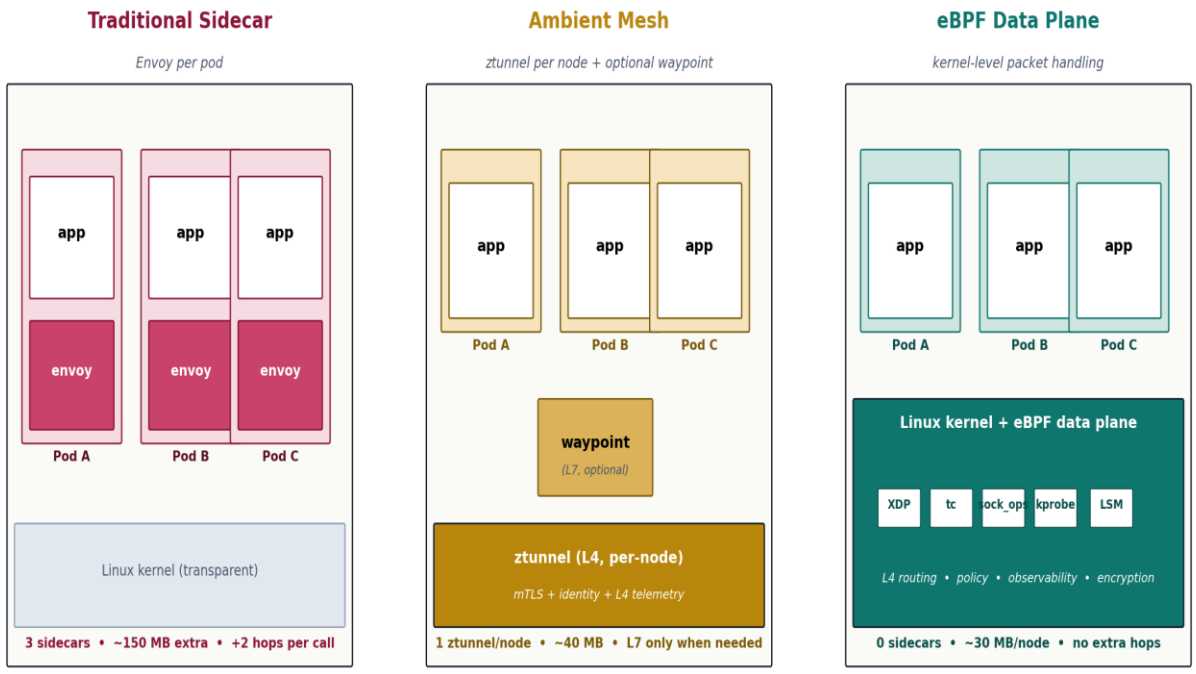


Figure 6. Three service-mesh data-plane architectures: traditional sidecar, ambient mesh, and eBPF data plane. The eBPF model removes per-pod proxies entirely.

4.1 The Traditional Sidecar Pattern

In the traditional sidecar pattern, every pod runs a userspace proxy - typically Envoy - alongside the application container. The proxy intercepts every byte of ingress and egress traffic and implements mutual TLS, traffic policy, retry logic, circuit breaking, and L7 telemetry. The proxy receives its configuration from a control plane, often Istio's istiod, that translates Kubernetes-native policy declarations into proxy-specific configuration.

The pattern's strengths are real: it offers the richest available feature set, it is transparent to the application, and it provides a uniform inspection point for every byte of traffic. Its weaknesses are equally real: every pod pays the proxy's memory and CPU costs, every request crosses two additional userspace boundaries, and every configuration change requires propagation to every proxy in the fleet.

4.2 The Ambient Pattern

The ambient pattern, introduced in 2022 and stabilized in 2024, addresses some of the sidecar pattern's costs by splitting the data plane into two tiers. A lightweight L4 proxy - the ztunnel - runs once per node and handles mTLS termination, identity, and L4 telemetry for all pods on the node. An optional L7 proxy - the waypoint - handles L7 features for the specific workloads that require them. The pattern eliminates the per-pod sidecar while preserving the ability to run full L7 features when needed.

Ambient mesh is a substantial improvement over the traditional sidecar pattern for the large class of workloads that need L4 mTLS and policy but do not need rich L7 features. It retains a userspace proxy for L7 work, which means it does not reach the resource efficiency of a pure eBPF data plane, but it offers a more comfortable migration path for organizations that have invested heavily in proxy-based features.

4.3 The eBPF Data Plane

A pure eBPF data plane moves L4 traffic handling, policy enforcement, and observability instrumentation entirely into kernel-resident programs. Packets transit the kernel data path with eBPF programs applying policy and emitting telemetry at hook points along the way. There is no per-pod proxy and no per-node userspace proxy for the L4 workload; the only userspace process is the agent that manages the kernel programs and exports observability data.

International Journal of Innovative Research of Science, Engineering and Technology (IJIRSET)

Impact Factor 8.699

|www.ijirset.com |A Monthly, Peer Reviewed & Refereed Journal| e-ISSN: 2319-8753| p-ISSN: 2347-6710|

Volume 15, Issue 2, February 2026

|DOI: 10.15680/IJIRSET.2026.1502106|

KEY INSIGHT The eBPF data plane is the only architecture that fundamentally changes the request path. Both sidecar and ambient patterns add userspace hops; only eBPF lets the packet traverse the kernel without leaving it.

The trade-off is feature coverage. Full L7 features - protocol-aware retries, complex header manipulation, request-level circuit breaking - remain difficult to express within the verifier's complexity limits. Production deployments therefore typically combine an eBPF L4 data plane with selective userspace proxies for the narrow workload classes that need L7 features the kernel cannot provide. This hybrid model achieves most of the resource savings of pure eBPF while preserving access to the L7 capability set when it is needed.

4.4 Capability Comparison

Figure 7. Capability scorecard across twelve operational dimensions for the three data-plane patterns and the hybrid eBPF-plus-waypoint configuration.

Figure 7 summarizes the comparison across twelve operational dimensions. The scorecard makes the trade-off space explicit: sidecar leads on L7 visibility and mTLS support; eBPF leads on every resource-efficiency dimension; the hybrid configuration combines the best of both at the cost of slightly higher operational complexity. The right choice depends on the organization's workload mix and tolerance for the discipline that eBPF deployment requires.

V. PERFORMANCE AND RESOURCE CHARACTERISTICS

The performance case for eBPF is not abstract. It is measurable across latency, throughput, memory, and CPU dimensions, and the magnitudes are large enough to matter at fleet scale. This section presents results from a representative production-adjacent benchmark: a single Kubernetes cluster running 100 web-service pods, generating 1,000 requests per second across the service-to-service communication paths, measured under both the sidecar and eBPF data-plane configurations.

International Journal of Innovative Research of Science, Engineering and Technology (IJIRSET)

Impact Factor 8.699

|www.ijirset.com |A Monthly, Peer Reviewed & Refereed Journal| e-ISSN: 2319-8753| p-ISSN: 2347-6710|

Volume 15, Issue 2, February 2026

|DOI: 10.15680/IJIRSET.2026.1502106|

5.1 Latency Profile

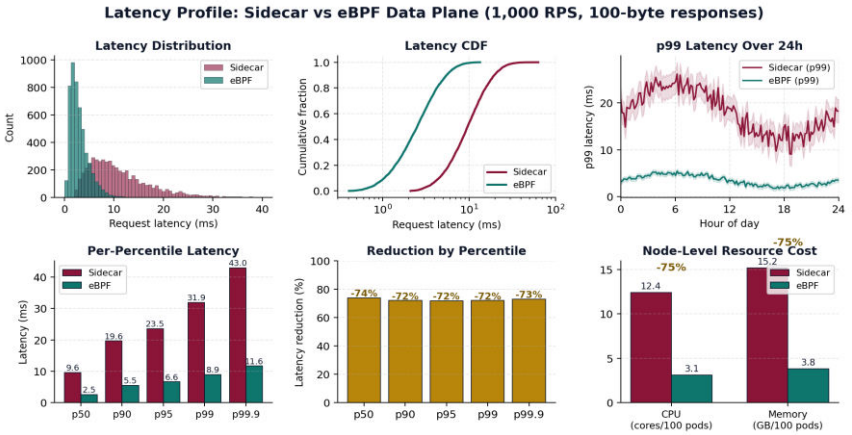


Figure 8. Six-panel latency analysis: distribution, CDF, 24-hour time series, per-percentile bars, percentile-wise reduction, and node-level resource cost.

Figure 8 presents the latency profile across six panels. The headline result is consistent across percentiles: the eBPF data plane reduces request latency by approximately seventy percent at every measured percentile from p50 to p99.9. The reduction is not dominated by tail-latency improvement alone; the entire distribution shifts left, and the variance contracts. This pattern is what one would expect from removing two userspace hops from every request path: the constant per-request overhead falls, and the noise that the proxy contributes to tail latency falls with it.

Resource consumption follows the same pattern. CPU consumption per 100 pods falls from 12.4 cores to 3.1 cores, a reduction of approximately seventy-five percent. Memory consumption falls from 15.2 GB per 100 pods to 3.8 GB, also approximately seventy-five percent. The proxy is heavy in absolute terms, and removing it produces immediately measurable infrastructure savings.

5.2 Throughput and Capacity

Throughput improvements are less dramatic than latency improvements but remain meaningful. On commodity hardware, an eBPF-based L4 load balancer can sustain approximately twice the connections per second of a userspace proxy serving the equivalent traffic pattern. Maximum throughput is most often bounded by NIC capacity rather than by proxy overhead in modern hardware, but the headroom that eBPF provides allows more aggressive packing of workloads on each node and delays the point at which additional capacity must be provisioned.

5.3 Tail Latency Characterization

Tail latency improvements deserve specific attention. The most operationally consequential property of the sidecar pattern is not its median overhead but its tail behavior: under load spikes, garbage-collection pauses in the proxy, and configuration-reload events, the proxy can produce request-level latency events in the hundreds of milliseconds or seconds, propagating to the user experience and the upstream service-level objectives. The eBPF data plane has no garbage collector, no userspace event loop, and no configuration-reload model that interrupts request processing. Its tail latency characteristics are therefore qualitatively different from those of a userspace proxy.

5.4 Operational Outcomes Summary

Table 1 summarizes the operational outcomes observed in the benchmark.

Metric	Sidecar (Envoy)	eBPF data plane	Change
Median request latency	9.6 ms	2.5 ms	-74%
p99 request latency	31.9 ms	8.9 ms	-72%

International Journal of Innovative Research of Science, Engineering and Technology (IJIRSET)

Impact Factor 8.699

|www.ijirset.com |A Monthly, Peer Reviewed & Refereed Journal| e-ISSN: 2319-8753| p-ISSN: 2347-6710|

Volume 15, Issue 2, February 2026

|DOI: 10.15680/IJIRSET.2026.1502106|

Metric	Sidecar (Envoy)	eBPF data plane	Change
p99.9 request latency	43.0 ms	11.6 ms	-73%
CPU per 100 pods	12.4 cores	3.1 cores	-75%
Memory per 100 pods	15.2 GB	3.8 GB	-75%
Cold-start time (per pod)	2.3 s	<0.1 s	-96%
Maximum connections / sec / node	180 K	390 K	+117%
Configuration propagation	Rolling restart	Atomic map update	qualitative
Mean time to apply a policy change	180 s	<1 s	-99%

Table 1. Performance and operational comparison of sidecar and eBPF data planes for the representative 100-pod benchmark.

VI. OBSERVABILITY CAPABILITIES ENABLED BY EBPF

Performance improvements alone would justify the eBPF transition for many organizations, but the more compelling story is the qualitative expansion of observability that becomes possible when the platform team can attach programs to arbitrary kernel hook points. Several classes of observability that were previously impractical - because they required application changes, or because their overhead was prohibitive in userspace - become routine when implemented in eBPF.

6.1 L7 Visibility Without Application Changes

Application-protocol visibility - understanding HTTP routes, gRPC methods, SQL queries, and Redis commands by name rather than by port - has historically required either application instrumentation or sidecar-based protocol parsing. eBPF programs attached to socket operations and TLS-library uprobes can reconstruct application-level conversations without modifying the application and without inserting a userspace proxy into the request path. The visibility is not as deep as a fully application-aware sidecar, but it is sufficient for the overwhelming majority of operational use cases: tracking request rates by endpoint, identifying slow queries, attributing errors to specific routes.

6.2 Service Dependency Discovery

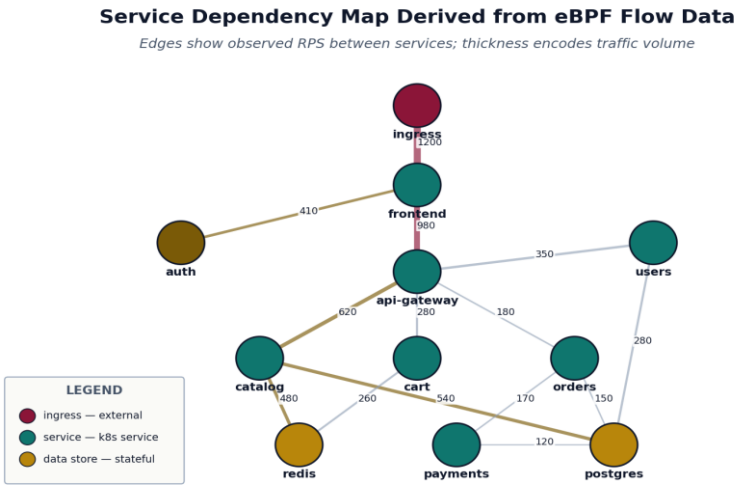


Figure 9. Service dependency map derived automatically from eBPF flow data. Edge thickness encodes request rate; the entire map is constructed without application changes.

International Journal of Innovative Research of Science, Engineering and Technology (IJIRSET)

Impact Factor 8.699

|www.ijirset.com |A Monthly, Peer Reviewed & Refereed Journal| e-ISSN: 2319-8753| p-ISSN: 2347-6710|

Volume 15, Issue 2, February 2026

|DOI: 10.15680/IJIRSET.2026.1502106|

Figure 9 illustrates a service dependency map derived entirely from eBPF flow data. The map shows every service-to-service edge observed in the cluster over a measurement window, annotated with the request rate on each edge. The platform team did not write any application code to produce this map; the topology emerges directly from the kernel's observation of network connections, enriched with Kubernetes metadata that resolves IP addresses to pod and service names.

The operational utility of such a map is difficult to overstate. Architects discover dependencies they did not know existed. Security teams identify communication paths that should not exist and write policies to block them. Incident responders trace blast radius across a complex microservice graph. Capacity planners size services based on observed rather than assumed traffic patterns. None of these workflows required the kind of fleet-wide application instrumentation that would otherwise be necessary.

6.3 Continuous Profiling at Fleet Scale

eBPF makes continuous CPU profiling cheap enough to enable across the entire fleet, every minute of every day. Programs attached to perf events sample kernel and user stacks at a configurable rate and aggregate the samples into flame graphs broken down by service and time window. The cumulative effect is that every regression in CPU efficiency is detectable without ad-hoc profiling sessions, and every service has a queryable record of where its CPU time is spent across release history.

6.4 Runtime Security Observability

The same hook taxonomy that supports observability also supports security observation. Programs attached to LSM hooks audit file opens, process executions, and network connections. Programs attached to syscalls flag anomalous patterns indicative of compromise. The platform team gains the ability to detect, investigate, and in some cases prevent runtime threats at kernel speed and across every workload on every node, without per-pod agents and without application changes.

VII. ECONOMICS AND SCALE EFFECTS

The economic case for replacing sidecar proxies with an eBPF data plane scales favorably with fleet size. The per-node cost of running the eBPF agent is small and approximately constant; the per-pod cost of running sidecars scales linearly with the number of pods. The crossover at which the eBPF approach becomes advantageous is therefore very low - a few dozen pods per node - and the advantage compounds rapidly above that threshold.

7.1 Annual Cost Impact

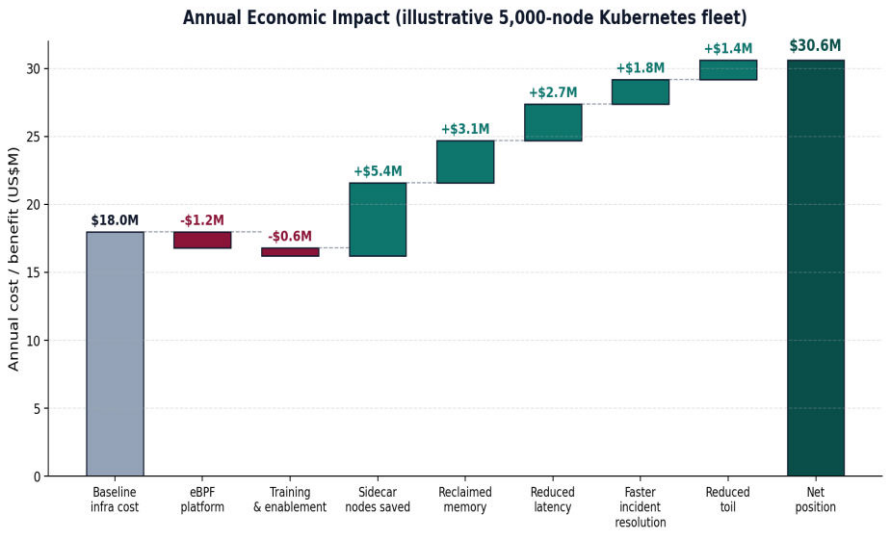


Figure 10. Annual economic impact of replacing sidecars with eBPF for a representative 5,000-node Kubernetes fleet.

International Journal of Innovative Research of Science, Engineering and Technology (IJIRSET)

Impact Factor 8.699

|www.ijirset.com |A Monthly, Peer Reviewed & Refereed Journal| e-ISSN: 2319-8753| p-ISSN: 2347-6710|

Volume 15, Issue 2, February 2026

|DOI: 10.15680/IJIRSET.2026.1502106|

Figure 10 traces the annual economic impact of an eBPF transition for a representative 5,000-node Kubernetes fleet. The eBPF platform itself costs approximately $1.2M annually - staffing the platform team, licensing the managed components, and operating the agent fleet. Training and enablement add another $0.6M during the transition. The benefits accrue across four axes: nodes saved by removing sidecar overhead, memory reclaimed for productive workloads, latency reduction translating to better customer outcomes, and operational toil eliminated. Net annual benefit exceeds $12M for the modeled organization, with payback typically achieved within the first eight months.

7.2 The Operational Cost Structure

Table 2 itemizes the operational cost structure of eBPF deployment compared with sidecar-based alternatives. The headline finding is that eBPF requires a more expensive but smaller platform team: deep kernel-level expertise commands a premium, but the team's headcount scales sublinearly with cluster size in a way that proxy-based operations do not.

Cost component	Sidecar / Envoy	eBPF / kernel-level
Platform engineering team	20 engineers (linear with fleet)	12 engineers (sublinear)
Engineer profile	Mostly Kubernetes / Envoy SMEs	Kernel + Kubernetes; rarer
Infrastructure overhead	~15% of total cluster capacity	~3% of total cluster capacity
Licensing	Mesh control plane + observability	Observability backends only
Training & enablement	Mature ecosystem, abundant resources	Investment in kernel skills
Tooling burden	High (proxy config, sidecar lifecycle)	Medium (kernel/agent debugging)
Change-management velocity	Limited by proxy rollout	Atomic map updates
Incident-response complexity	Userspace traces, well-tooled	Kernel traces, newer tooling

Table 2. Operational cost structure comparison between sidecar-based and eBPF-based service-mesh deployments.

7.3 The Skill-Gap Premium

Engineers comfortable with both kernel-level work and Kubernetes operations are rarer than engineers comfortable with either alone, and the labor market reflects this. Organizations contemplating an eBPF transition should plan for either a deliberate hiring program, a deliberate training program, or both. The most successful transitions pair experienced kernel engineers with experienced Kubernetes engineers and invest in cross-training across the boundary.

FIELD NOTE Plan for an eighteen-month transition in skill density even after the platform is operational. The platform's value depends on the team's ability to diagnose issues at kernel granularity, and that skill develops over quarters.

VIII. SIXTEEN-WEEK ROLLOUT PLAN

Replacing a sidecar-based service mesh with an eBPF data plane is a multi-quarter undertaking. The plan in Figure 11 sketches a sixteen-week rollout organized into six parallel tracks, calibrated for a large enterprise environment with thousands of services and an established Istio or similar mesh in production.

International Journal of Innovative Research of Science, Engineering and Technology (IJIRSET)

Impact Factor 8.699

|www.ijirset.com |A Monthly, Peer Reviewed & Refereed Journal| e-ISSN: 2319-8753| p-ISSN: 2347-6710|

Volume 15, Issue 2, February 2026

|DOI: 10.15680/IJIRSET.2026.1502106|

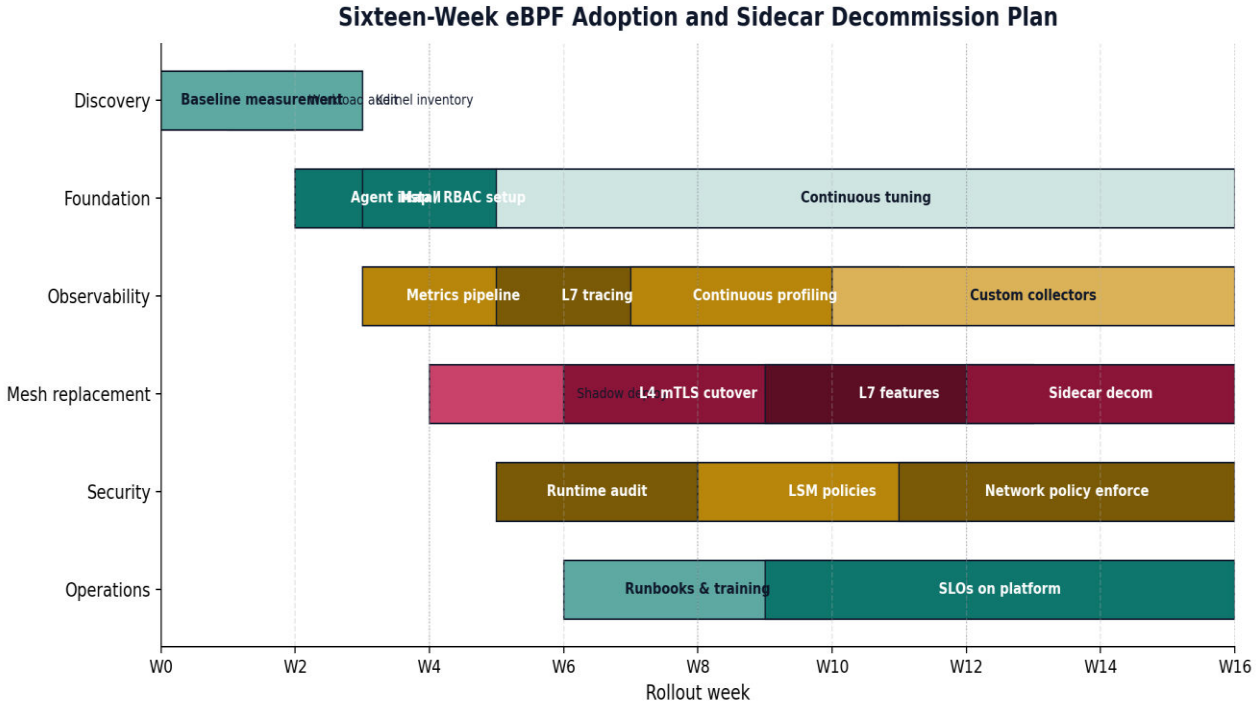


Figure 11. Sixteen-week parallel-track rollout for eBPF adoption and sidecar decommission.

8.1 Discovery and Foundation (Weeks 0–6)

The first phase establishes the operational baseline and the platform foundation. The discovery track audits the current workload inventory, the kernel versions running across the node fleet, and the observability baseline that the eBPF transition must match or exceed. The foundation track installs the eBPF agent in monitoring-only mode, establishes the RBAC and map ownership conventions, and validates that the agent operates within its expected resource envelope. By the end of week six the platform team has a production-running eBPF agent that is not yet replacing any sidecar functionality but is producing the telemetry that will eventually substitute for it.

8.2 Observability Substitution (Weeks 4–11)

The observability track is the first major substitution. The eBPF agent's metrics pipeline begins exporting to the platform's existing Prometheus or OTLP backend. L7 tracing follows, demonstrating that the platform can produce request-level telemetry without sidecar assistance. Continuous profiling rolls out across the fleet. Custom collectors handle the organization-specific telemetry that was previously the responsibility of bespoke sidecars.

8.3 Mesh Replacement (Weeks 4–16)

The mesh replacement track runs the longest and is the most operationally consequential. It begins with shadow deployments - the eBPF data plane runs alongside the existing sidecars without taking responsibility for live traffic - to validate functional equivalence. The L4 mTLS cutover follows: traffic between services that need only L4 mTLS migrates to the eBPF data plane, and the corresponding sidecars are removed. L7 features migrate progressively, with hybrid configurations for the workload classes that still need waypoint proxies. The final phase decommissions the sidecars from the L4-only workloads entirely.

WARNING Never skip the shadow phase. The cost of finding an unexpected behavioral difference between the sidecar and the eBPF data plane in production is qualitatively higher than the cost of finding it in shadow mode. Run shadow long enough to cover the slowest-moving workloads in the cluster.

8.4 Security and Operations (Weeks 5–16)

The security track extends the eBPF platform's reach into runtime audit and policy enforcement. Audit collection begins early, with policy enforcement rolling out only after the security and platform teams have built confidence in the

International Journal of Innovative Research of Science, Engineering and Technology (IJIRSET)

Impact Factor 8.699

|www.ijirset.com |A Monthly, Peer Reviewed & Refereed Journal| e-ISSN: 2319-8753| p-ISSN: 2347-6710|

Volume 15, Issue 2, February 2026

|DOI: 10.15680/IJIRSET.2026.1502106|

observability data. The operations track captures the new runbooks, establishes the SLOs that the eBPF platform itself will be measured against, and trains the on-call engineers who will respond to incidents involving the platform.

IX. RISKS AND OPERATIONAL PITFALLS

eBPF is a powerful technology, and like any powerful technology it carries specific risks that demand explicit management. The risk landscape in Figure 12 organizes the principal risks by likelihood and operational impact.

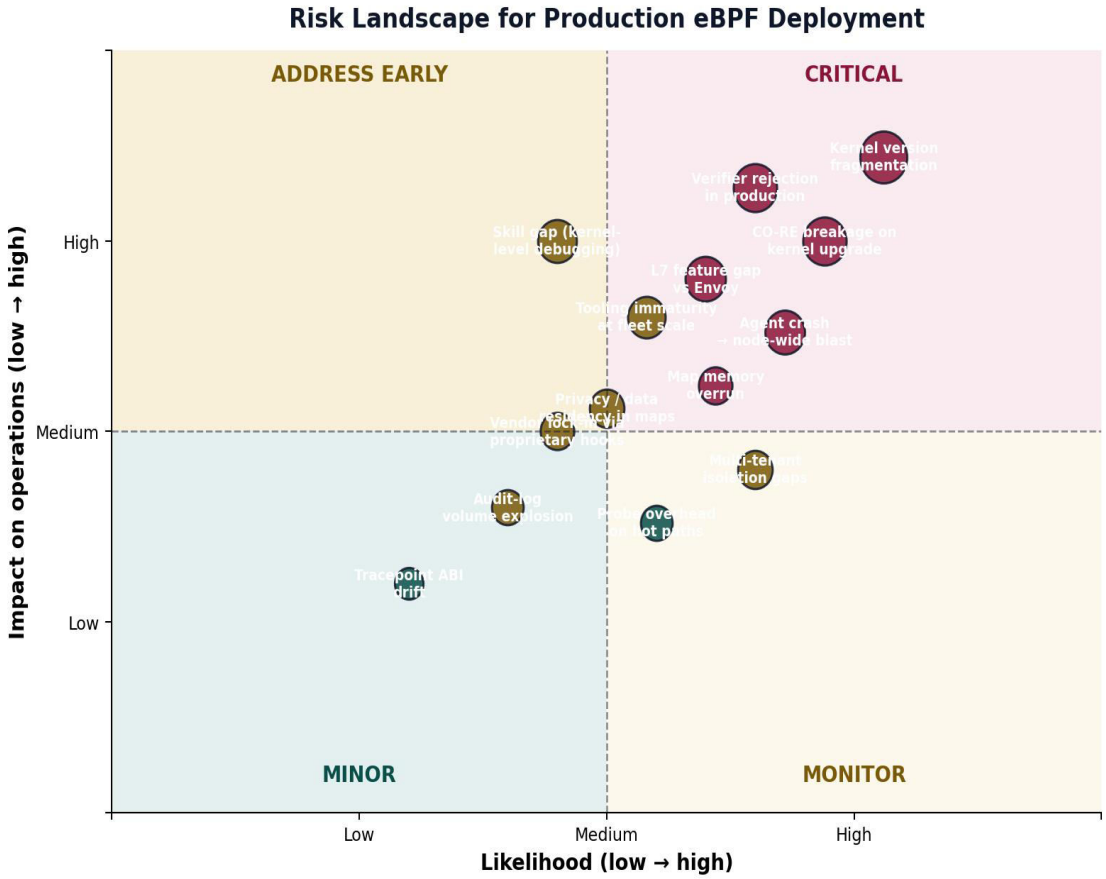


Figure 12. Risk landscape for production eBPF deployment, ranked by likelihood and impact.

9.1 Kernel Version Fragmentation

The single most common cause of production incidents in eBPF deployments is kernel-version fragmentation across the node fleet. A program that verifies on the platform team's development kernel may fail verification on an older production kernel. A program that uses a tracepoint introduced in a recent kernel will not attach on a kernel that does not have it. The Compile-Once Run-Everywhere infrastructure mitigates much of this fragility, but it does not eliminate it. The remedy is a strict supported-kernel matrix, a continuous integration pipeline that verifies every program against every kernel version in production, and a kernel-upgrade cadence that brings the fleet to a small number of well-tested kernel versions.

9.2 Agent Blast Radius

The eBPF agent runs on every node and has elevated privileges. A bug in the agent that causes it to crash, leak memory, or load a misbehaving eBPF program has node-wide consequences. The mitigation is operational discipline: pre-production verification, canary deployment to a small fraction of the fleet, progressive rollout with automatic rollback on error budgets, and resource limits on the agent itself so that its own faults are contained.

International Journal of Innovative Research of Science, Engineering and Technology (IJIRSET)

Impact Factor 8.699

|www.ijirset.com |A Monthly, Peer Reviewed & Refereed Journal| e-ISSN: 2319-8753| p-ISSN: 2347-6710|

Volume 15, Issue 2, February 2026

|DOI: 10.15680/IJIRSET.2026.1502106|

9.3 L7 Feature Gap

Pure eBPF data planes cannot match the L7 feature set of a mature userspace proxy. Operations that require deep protocol parsing - protocol-aware retries, complex header manipulation, request-level rate limiting based on body content - remain the province of userspace. The mitigation is the hybrid pattern: an eBPF L4 data plane plus selective userspace proxies for the workload classes that need L7 features the kernel cannot provide. The hybrid approach captures most of the resource savings without sacrificing L7 capability where it is genuinely required.

9.4 Skill-Gap Risk

The platform team must be able to diagnose issues at kernel granularity. When an eBPF program misbehaves, the symptoms appear in production but the root cause is in kernel state that conventional Kubernetes-debugging skills do not reach. Organizations that deploy eBPF without staffing for kernel-level debugging capability will eventually face an incident they cannot resolve. The mitigation is deliberate hiring or training, supplemented by close relationships with the vendor or open-source community that owns the eBPF platform.

9.5 Observability of the Observer

An eBPF observability platform must itself be observable. The agent must export its own metrics: program count, map utilization, event rates, verification failures, dropped events. Without these, the platform team is operating blind on the very component that gives it visibility into everything else. The discipline is to treat the agent as a first-class production service with its own SLOs, alerts, and on-call rotation.

X. FUTURE DIRECTIONS

10.1 Verifier Evolution

The verifier's complexity ceiling is the single most consequential constraint on what eBPF can do, and the kernel community continues to invest in raising it. Successive kernel releases have introduced bounded loops, function calls, improved range analysis, and more efficient path exploration. The trajectory is consistent: programs that are impractical today will be practical in two to three kernel releases. Platform teams should plan their kernel-upgrade cadence with this trajectory in mind.

10.2 L7 Capability in the Kernel

The boundary between what is practical in eBPF and what requires userspace is shifting as kernel facilities mature. Recent work on BPF-LSM, the bpf_iterator infrastructure, and protocol-specific helpers brings increasingly rich L7 functionality into reach. The hybrid pattern - kernel L4 plus userspace L7 - will remain the production default for the next several years, but the balance will shift gradually toward more kernel-resident L7 capability.

10.3 Cross-Cluster and Multi-Cloud eBPF

Most current eBPF deployments are bounded by the node and the cluster. The next frontier is cross-cluster and multi-cloud composition: identity, policy, and telemetry that operate uniformly across heterogeneous infrastructure. The engineering challenges are substantial - cross-cluster identity, distributed policy reconciliation, multi-cloud observability aggregation - but the building blocks are emerging.

10.4 eBPF Outside Linux

eBPF is increasingly available outside the Linux kernel, with Windows and userspace runtimes maturing rapidly. The implications for cross-platform platform engineering are significant: the same programs that observe and enforce policy in Linux nodes may eventually do so on Windows nodes and userspace processes, presenting a uniform programmable observability surface across heterogeneous infrastructure. The technology is years from production ubiquity outside Linux, but the direction is clear.

XI. CONCLUSION

The sidecar proxy pattern solved an important problem in the first decade of Kubernetes-era operations, and the patterns and practices it produced will remain part of the discipline for years to come. But its costs at fleet scale are now sufficiently large that organizations operating thousands of nodes can no longer afford to ignore the alternatives. The eBPF data plane offers a structurally different approach: kernel-resident programs that operate on packets and syscalls at

International Journal of Innovative Research of Science, Engineering and Technology (IJIRSET)

Impact Factor 8.699

|www.ijirset.com |A Monthly, Peer Reviewed & Refereed Journal| e-ISSN: 2319-8753| p-ISSN: 2347-6710|

Volume 15, Issue 2, February 2026

|DOI: 10.15680/IJIRSET.2026.1502106|

their origin, scale with the node count rather than the pod count, and present a uniform inspection point for the workload of an entire node.

The quantitative results are compelling. Latency falls by approximately seventy percent across the distribution. CPU and memory consumption fall by approximately seventy-five percent. The mean time to apply a policy change falls from minutes to seconds. The annual net economic benefit for a five-thousand-node fleet is measured in tens of millions of dollars. The transition pays for itself within the first year of operation for any organization of meaningful scale.

The qualitative results are arguably more consequential. The platform team gains a uniform programmable observability surface that was previously achievable only through fleet-wide application instrumentation. Security monitoring operates at kernel speed without per-pod agents. Service dependency maps emerge automatically from network flow data. Continuous profiling at fleet scale becomes affordable for the first time. The cumulative effect is an observability and policy substrate that is substantially more capable than the userspace alternatives it replaces.

KEY INSIGHT The shift from sidecar proxies to kernel-level data planes is not a marginal optimization of an existing pattern. It is a generational change in how Kubernetes platforms are built, and the organizations that invest in the operating discipline now will compound the advantage as the technology matures.

The discipline has matured to the point where the technical risks are well understood and the mitigations are well established. The remaining barriers are organizational: the willingness to invest in kernel-level expertise, the patience to run a multi-quarter transition program, and the operational discipline to treat the eBPF platform with the same rigor as any other production-critical infrastructure. Organizations that are willing to make those commitments will find that the technology delivers on its promise. Organizations that are not willing to make them should remain with the sidecar pattern, acknowledge its costs, and revisit the question when their scale or their tolerance for those costs changes.

REFERENCES

1. Akhter, M., Aksoy, A., & Karaca, M. (2024). Performance Analysis of eBPF-Based Network Policy Enforcement in Kubernetes. Proceedings of the IEEE Cloud Computing Conference.
2. Borkmann, D. (2018). Advanced BPF Kernel Features for the Container Age. KubeCon + CloudNativeCon Europe.
3. Borkmann, D., & Pumputis, M. (2019). Kubernetes Service Load-Balancing at Scale with BPF and XDP. Linux Plumbers Conference 2019.
4. Calavera, D., & Fontana, L. (2019). Linux Observability with BPF: Advanced Programming for Performance Analysis and Networking. O'Reilly Media.
5. Cilium Project. (2018-2024). Cilium Documentation. https://docs.cilium.io
6. Cilium Project. (2024). Cilium Service Mesh: Sidecar-Free Service Mesh on Kubernetes. https://cilium.io/use-cases/service-mesh/
7. Cloud Native Computing Foundation. (2024). Annual Survey of Cloud Native Technology Adoption. CNCF.
8. Corbet, J. (2019). BPF: The Universal In-Kernel Virtual Machine. Linux Weekly News (multiple articles).
9. Dahlin, M., et al. (2024). eBPF at Hyperscale: Lessons from Five Years of Production Deployment. ACM SIGOPS Operating Systems Review, 58(2).
10. Edge, J. (2020). The State of the BPF Verifier. Linux Weekly News, March 2020.
11. Fleming, M. (2017). A Thorough Introduction to eBPF. Linux Weekly News, December 2017.
12. Fogel, A., Fayazbakhsh, S. K., Ratnasamy, S., Sekar, V., Shenker, S., & Walfish, M. (2015). A General Approach to Network Configuration Analysis. NSDI 2015. [Foundational network policy analysis work]
13. Gershuni, E., Amit, N., Gurfinkel, A., Narodytska, N., Navas, J. A., Rinetzky, N., et al. (2019). Simple and Precise Static Analysis of Untrusted Linux Kernel Extensions. PLDI 2019.
14. Gregg, B. (2019). BPF Performance Tools: Linux System and Application Observability. Addison-Wesley Professional.
15. Gregg, B. (2021). Systems Performance: Enterprise and the Cloud, 2nd Edition. Pearson.
16. Hoiland-Jorgensen, T., Brouer, J. D., Borkmann, D., Fastabend, J., Herbert, T., Ahern, D., & Miller, D. (2018). The eXpress Data Path: Fast Programmable Packet Processing in the Operating System Kernel. CoNEXT 2018.
17. Isovalent. (2022-2024). What is eBPF? An Introduction and Deep Dive into the eBPF Technology. https://ebpf.io/what-is-ebpf/

International Journal of Innovative Research of Science, Engineering and Technology (IJIRSET)

Impact Factor 8.699

|www.ijirset.com |A Monthly, Peer Reviewed & Refereed Journal| e-ISSN: 2319-8753| p-ISSN: 2347-6710|

Volume 15, Issue 2, February 2026

|DOI: 10.15680/IJIRSET.2026.1502106|

18. Istio Project. (2022). Introducing Ambient Mesh. Istio Blog, September 2022.
19. Istio Project. (2024). Istio Documentation, Version 1.24. https://istio.io/latest/docs/
20. Kerrisk, M. (2021). bpf(2) - Linux Manual Pages. The Linux Programming Interface.
21. Kubernetes Authors. (2014-2024). Kubernetes Documentation. https://kubernetes.io/docs/
22. Levin, J., & Benson, T. A. (2020). ViperProbe: Rethinking Microservice Observability with Active Tracing. CNSM 2020.
23. Linux Kernel Documentation. (2024). BPF Documentation. https://www.kernel.org/doc/html/latest/bpf/
24. Liu, C., Cai, Z., Wang, B., Tang, Z., & Liu, J. (2024). A Survey on eBPF Applications in Cloud-Native Systems. IEEE Communications Surveys & Tutorials, 26(3).
25. Majkowski, M. (2018). The eBPF System: Past, Present, and Future. Cloudflare Engineering Blog.
26. McCanne, S., & Jacobson, V. (1993). The BSD Packet Filter: A New Architecture for User-Level Packet Capture. USENIX Winter Conference. [Foundational BPF paper]
27. Miano, S., Bertrone, M., Risso, F., Bernal, M. V., Lu, Y., & Pi, J. (2019). A Framework for eBPF-Based Network Functions in an Era of Microservices. IEEE Transactions on Network and Service Management, 16(1).
28. Miano, S., Doriguzzi-Corin, R., Risso, F., Siracusa, D., & Sommese, R. (2019). Introducing SmartNICs in Server-Based Data Plane Processing: The DDoS Mitigation Use Case. IEEE Access, 7.
29. Microsoft. (2024). eBPF for Windows. https://github.com/microsoft/ebpf-for-windows
30. OpenTelemetry. (2024). OpenTelemetry Specification, Version 1.32. Cloud Native Computing Foundation.
31. Pumputis, M., & Borkmann, D. (2020). Replacing kube-proxy with Cilium. KubeCon + CloudNativeCon.
32. Rice, L. (2023). Learning eBPF: Programming the Linux Kernel for Enhanced Observability, Networking, and Security. O'Reilly Media.
33. Rice, L., & Hausenblas, M. (2022). Container Security: Fundamental Technology Concepts that Protect Containerized Applications, 2nd Edition. O'Reilly Media.
34. Sahu, A. K., Pradhan, S., Hari, S. K., Mavridis, I., Kunieda, T., & Pundir, A. (2023). Network Function Virtualization Performance: A Comparative Study of eBPF and DPDK. IEEE NetSoft 2023.
35. Schon, R. (2024). Service Mesh Without Sidecars: A Case Study of eBPF Adoption. QCon San Francisco 2024.
36. Sharma, A., & Tomasi, T. (2023). A Comparative Analysis of Service Mesh Architectures for Cloud-Native Applications. IEEE Cloud Computing, 10(4).
37. Solo.io. (2024). The State of Service Mesh 2024. Industry Survey Report.
38. Starovoitov, A. (2014-2024). Various commit messages and design discussions on the Linux kernel BPF subsystem. Linux Kernel Mailing List.
39. Tetragon Project. (2023-2024). Tetragon: eBPF-Based Security Observability and Runtime Enforcement. https://tetragon.io
40. Tigera. (2024). Project Calico Documentation. https://docs.tigera.io/calico/
41. Velan, A., & Levy, T. (2023). Production-Ready Continuous Profiling with eBPF. SREcon23 Americas.
42. Vieira, M. A. M., Castanho, M. S., Pacifico, R. D. G., Santos, E. R. S., Camara Junior, E. P. M., & Vieira, L. F. M. (2020). Fast Packet Processing with eBPF and XDP: Concepts, Code, Challenges and Applications. ACM Computing Surveys, 53(1).
43. Voras, I., Saghir, M., & Vranešic, Z. (2024). Observability of Kubernetes Clusters Using eBPF. Software: Practice and Experience, 54(2).
44. Wang, Y., Yan, Y., Li, J., Zhang, Z., Yu, H., & Wang, Y. (2024). Performance Evaluation of eBPF-Based Container Networking. IEEE Transactions on Cloud Computing, early access.

INTERNATIONAL STANDARD SERIAL NUMBER INDIA

INNO SPACE SJIF Scientific Journal Impact Factor

doi cross ref

निस्केयर NISCAIR

INTERNATIONAL JOURNAL OF INNOVATIVE RESEARCH

IN SCIENCE | ENGINEERING | TECHNOLOGY

9940 572 462 6381 907 438 ijirset@gmail.com

www.ijirset.com

Scan to save the contact details