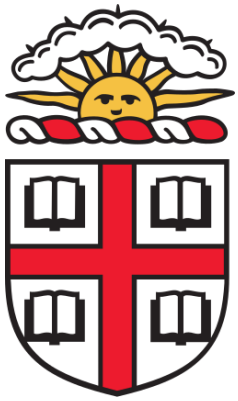
Reducing Instrumentation Overhead in MeshTrek with Userspace eBPF

A thesis submitted in partial fulfillment of the requirements for the degree of Bachelor of Science with Honors in the Computer Science concentration of Brown University.

Hee Su Julie Chung
Brown University
April 29, 2026

Thesis Advisor:
Akshay Narayan, PhD
Assistant Professor of Computer Science
Department of Computer Science

Secondary Reader:
Nicholas DeMarinis, PhD
Assistant Professor of Computer Science
Department of Computer Science



1. Abstract

This thesis presents a modified measurement tool for analyzing service mesh latency by capturing and correlating per-request spans across microservice calls. The original tool, MeshTrek, instruments Envoy using eBPF uprobes to decompose service mesh latency into fine-grained processing stages, including protocol parsing, filter execution, and scheduling-induced wait time on a per-request basis. While MeshTrek enables detailed latency attribution, its reliance on kernel-based eBPF introduces overhead due to context switches between userspace and kernel space. This overhead becomes particularly pronounced at higher request rates, where frequent probe execution amplifies tail latency and distorts measurement accuracy. To address this limitation, this work extends MeshTrek by integrating bpftime, a userspace eBPF runtime framework. bpftime executes eBPF programs directly in userspace via dynamic binary rewriting and runtime injection, eliminating kernel transitions and significantly reducing uprobe overhead. By preserving MeshTrek's fine-grained instrumentation while replacing its kernel-based execution path, this approach minimizes measurement-induced perturbation, particularly in tail latency metrics such as p99. By combining MeshTrek's detailed latency attribution with bpftime's low-overhead execution model, this work enables more accurate and scalable analysis of service mesh performance under high load, while maintaining compatibility with existing eBPF toolchains.

2. Introduction

Service meshes have become a standard abstraction for managing communication in microservice architectures, enabling features such as security, observability, and traffic control without requiring modifications to application code. In this model, sidecar proxies (e.g., Envoy) intercept all service-to-service communication and can also additionally enforce L7 policies through programmable filter chains. While this design provides flexibility and modularity, it introduces non-trivial latency overhead due to additional processing at the proxy sidecar. Prior work has shown that service mesh overhead can vary widely depending on protocol choice, request patterns, and deployed policies, with factors such as proxy scheduling, request queuing, and filter execution contributing significantly to end-to-end latency.

Understanding where this overhead originates is critical for both system designers and operators, particularly for latency-sensitive workloads. While previous research on service mesh latency, such as MeshInsight, demonstrates that aggregate end-to-end latency alone is insufficient for understanding service mesh overhead and proposes a decompositional model across components, it relies on model-based estimation using averaged component costs. Although MeshInsight performs critical path analysis at the application level, it does not capture per-request execution timing or dynamic runtime effects within the proxy, such as scheduling delays, queueing, and contention. As a result, it provides limited visibility into how these factors contribute to latency, particularly under high load where tail behaviors become dominant.

MeshTrek addresses this limitation by attaching kernel-based eBPF (extended Berkeley Packet Filter) uprobes to selected Envoy hookpoints within the request processing pipeline. These probes record timestamps and correlate events across stages such as protocol parsing, filter execution, and scheduling-induced wait time, enabling detailed latency attribution on a

per-request basis. This detailed observation of latency reveals previously underexplored contributing factors to latency, such as wait time, caused by queuing and scheduling effects within the Envoy.

Despite the improved fine-grained latency instrumentation, MeshTrek itself introduces additional measurement overhead due to its reliance on kernel-based eBPF uprobes. Each probe execution incurs context switches between userspace and kernel space, which adds overhead to request processing. At higher request rates, where probes may fire frequently for each request, this overhead accumulates and disproportionately impacts mean latency and tail latency (e.g., p99). Consequently, the measurement system itself perturbs the system being measured, distorting latency distributions and reducing the fidelity of performance analysis.

In this work, we extend MeshTrek by integrating bpftime, a userspace eBPF runtime framework that executes eBPF programs within the userspace. bpftime leverages dynamic binary rewriting and runtime injection to attach inline hookpoints to userspace functions, enabling probe execution within the same address as the target application, which, in this case, is Envoy. By limiting the context switches into the kernel space, this approach significantly reduces uprobe overhead and minimizes measurement-induced perturbation. Importantly, this design preserves MeshTrek's instrumentation logic, including hookpoints, timestamping, and request correlation, while replacing only the execution substrate. This enables more accurate and scalable latency measurement, particularly in tail latency, while maintaining compatibility with existing eBPF toolchains.

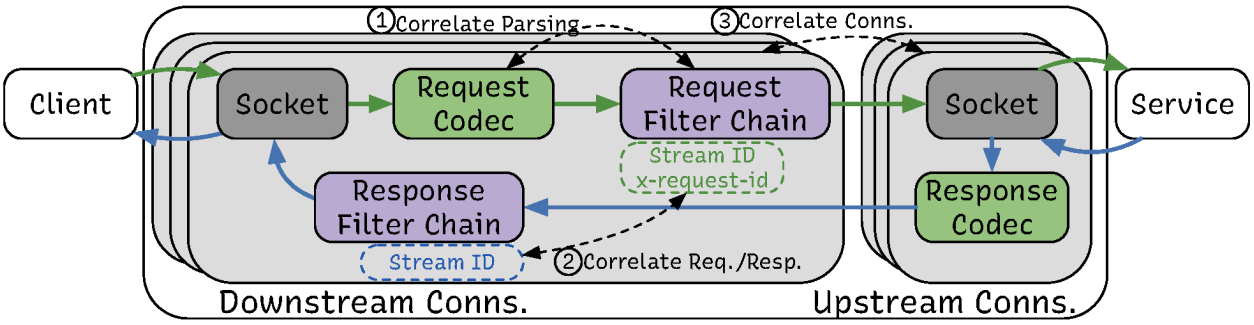


Figure 1) This diagram shows the lifecycle of an Envoy request, starting from the downstream socket, entering the codec stage, processing through the filter chain, and forwarding to the upstream socket,and then doing the reverse for the response while maintaining per-request correlation.

3. Background

In the modern age of the internet, applications transitioned from monolithic architectures to microservices, and developers and operators recognized the need to secure communication by encrypting data in transit. Service meshes emerged as a solution to this problem, providing encryption-in-flight through both the control plane and the data plane. The control plane manages TLS certificates for microservices, while the data plane handles request processing through a service proxy, or a sidecar, which can terminate and authenticates connections between application pods. This design also ensures that the channel between the sidecar and the pod is secure, with no potential adversaries gaining access between them. Due to this intertwined design, proxies also evolved to support additional service mesh policies, also known as filters. Filters would operate on L4 or L7 layers, allowing various functionalities such as load balancing or rate limiting to be supported.

Envoy is a widely used proxy implementation and also the primary focus of MeshTrek. In our experiments, we utilize Kubernetes, a popular container orchestration system. In Kubernetes, a pod is the atomic unit of deployment and can represent a single instance of a microservice

within an application. A pod may consist of one or more containers deployed together, typically including both the application container and its associated sidecar proxy.

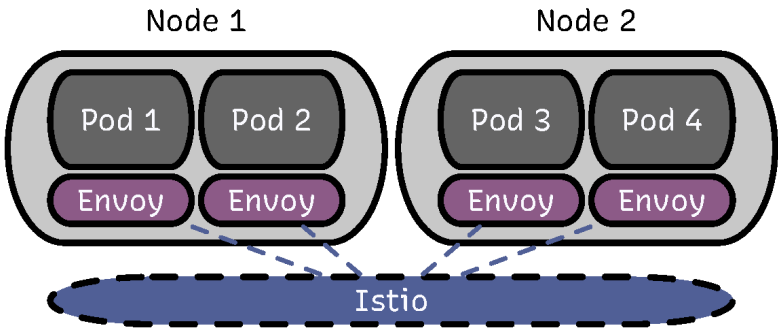


Figure 2) Service mesh architecture in Kubernetes. Each application pod is co-located with an Envoy sidecar proxy, which intercepts all service-to-service communication. The Istio control plane configures these proxies, enabling L7 functionality such as routing, security, and observability.

4. Methodology

4.1 MeshTrek Overview

This work builds upon MeshTrek, an eBPF-based measurement tool that captures per-request latency breakdowns across microservice calls. MeshTrek enables fine-grained latency attribution by placing probes at key stages in Envoy's request processing pipeline, including protocol parsing, filter execution, and scheduling-induced delays.

Unlike prior systems such as DeepFlow or MeshMark, which observe requests at the span level rather than breaking down to processing level, MeshTrek instruments Envoy's internal execution path by attaching probes to selected hookpoints corresponding to different stages of the request lifecycle. While DeepFlow provides end-to-end visibility, it does not attribute latency to specific processing stages within service mesh proxies, focusing more on aggregate latency. On the other hand, by recording timestamps and correlating them using request identifiers,

MeshTrek enables per-request latency attribution across stages such as protocol parsing, request handling, filter execution, and connection establishment.

MeshTrek decomposes latency into stages including Downstream Codec Start, Downstream Codec End, Upstream Codec Start, Upstream Codec End, Request Filters, Response Filters, Establish Upstream, and Stream Lifetime. This approach revealed a previously underexplored source of latency–wait time–which arises from scheduling inefficiencies within Envoy and can account for a significant portion of per-request latency.

MeshTrek's instrumentation pipeline operates by attaching eBPF programs to selected Envoy functions using uprobes, triggering probe execution at function entry and exit. Each probe records timestamps along with request identifiers, enabling correlation across multiple function invocations within the same request. Per-request state is maintained in kernel BPF maps, which track per-request context and timing information throughout execution. Finally, latency is decomposed into fine-grained stages such as processing time and wait time, using critical-path analysis to avoid double-counting overlapping operations.

4.2 Limitations of Kernel-Based instrumentation

Despite its improved observability, MeshTrek relies on kernel-based eBPF uprobes to instrument Envoy functions, which introduces measurement overhead. Each probe execution triggers a trap from userspace into kernel space, where the eBPF program is executed, followed by a return to userspace. This results in two context switches per probe event. At higher request rates, where probes may fire multiple times per request, this overhead accumulates and perturbs the system being measured. In particular, it disproportionately affects tail latency (e.g., p99), leading to distortion in latency measurements under high load. As a result, the measurement

mechanism itself interferes with the system's natural performance characteristics, reducing the fidelity of latency analysis.

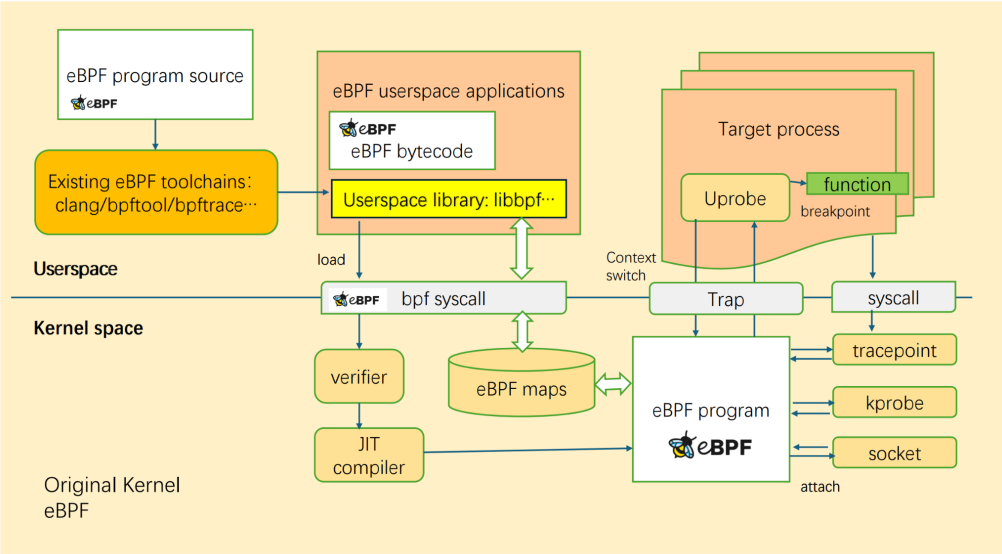


Figure 3) Diagram from bpftime. Kernel-based eBPF execution model. eBPF programs are loaded into the kernel via the `bpf syscall` and attached to user-space functions using uprobes. Each probe triggers a trap into the kernel, where the eBPF program executes, before returning control to userspace, incurring context switch overhead.

4.3 Approach: Userspace eBPF with bpftime

To address this limitation, we replace MeshTrek's kernel-based execution path with bpftime, a userspace eBPF runtime framework. bpftime enables eBPF programs to execute entirely within userspace, which eliminates the need for kernel transitions. Therefore, it provides a complete runtime environment, including program loading, eBPF maps, and event handling while also remaining compatible with pre-existing eBPF toolchains such as clang and libbpf. Internally, bpftime utilizes dynamic binary rewriting and runtime injection to instrument target applications. A shared library agent is injected into the target Envoy process, and selected functions are modified using inline hooks. When these functions are invoked, execution is

redirected to the eBPF runtime within the same address space, allowing probe logic to execute without leaving userspace.

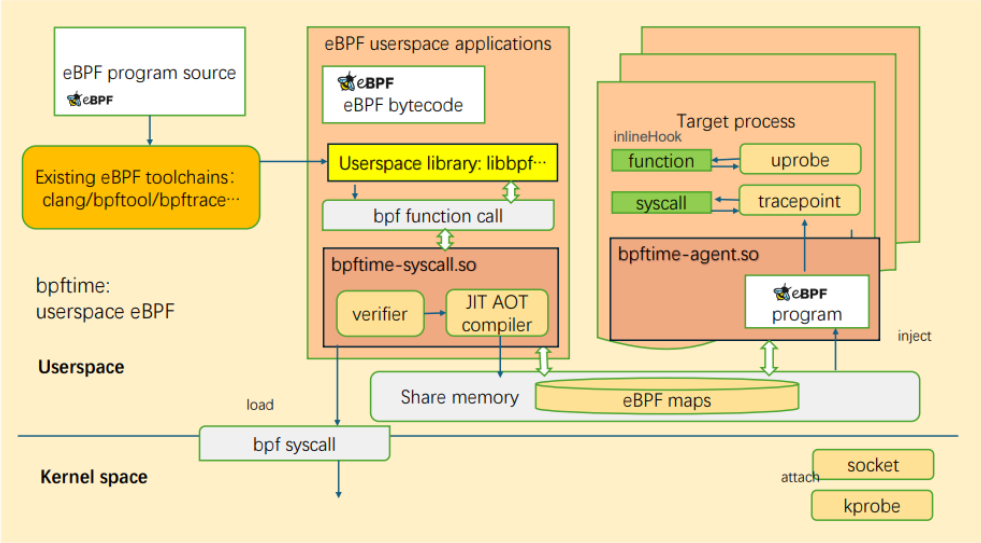


Figure 4) Diagram from bpftime. Userspace eBPF execution model. eBPF programs are executed directly within the target process via runtime injection and inline hooks, eliminating kernel involvement. Shared memory maps enable communication without kernel transitions, reducing the overhead associated with traditional kernel-based uprobes.

4.4 Integration with MeshTrek

Our approach integrates bpftime into MeshTrek to optimize while preserving the measurement logic. The key design principle is that only the execution substrate is modified and all higher-level logic remains unchanged. In particular, the same Envoy functions are instrumented, the same timestamping and correlation logic is used, and the same latency attribution model is preserved. The primary difference lies in how the probes are executed. In the original MeshTrek design, probes are implemented using kernel-based uprobes and store data in kernel eBPF maps. On the other hand, this work replaces kernel-based uprobes with userspace inline hooks and stores measurement data in shared memory maps provided by the bpftime

runtime. This design ultimately eliminates kernel involvement during probe execution and reduces measurement overhead.

4.5 Measurement Workflow

The modified measurement pipeline begins by loading the bpftime runtime and attaching it to the Envoy process, enabling userspace execution of eBPF programs. The instrumentation logic is defined in binary program `trace.bpf.o`, which specifies the probe behavior at selected Envoy hookpoints. A Rust-based loader is used to load the compiled eBPF object, resolve target symbols within the Envoy binary (e.g., `http1_hookpointDispatch`), and attach probes through the bpftime runtime.

To enable stable execution, we modify the pipeline to maintain a persistent Frida-based instrumentation. In contrast to the default behavior of bpftime, where the Frida module attaches briefly and exits, this implementation ensures that the injected runtime remains active for the duration of the measurement. Frida is used to inject trampolines at target hookpoints, redirecting execution from the original Envoy function into the corresponding eBPF handlers executing in userspace.

An additional challenge arose from discrepancies in function address resolution. The static address used by the eBPF program corresponds to offset-adjusted values (`real_address - c`), whereas the Frida module requires the actual runtime address of the function. Therefore, we apply a correction factor during attachment, adding the constant offset `c` to recover the true function address before installing the probe.

Once attached, execution of the instrumented function triggers the eBPF program directly within the Envoy process. The current implementation validates successful attachment and

execution at a single hookpoint, focusing on function entry events. While this establishes the feasibility of userspace instrumentation within Envoy, full end-to-end measurement of per-request latency and span reconstruction remains ongoing work.

By shifting probe execution to userspace, this approach eliminates kernel context switches and has the potential to reduce instrumentation overhead. These results demonstrate that fine-grained instrumentation of the service mesh proxy is feasible, though further work is required to fully realize accurate and complete latency measurement under load.

5. Evaluation

We evaluate the impact of tracing on service latency using a synthetic microservice application deployed within a service mesh environment. The application consists of a simple multi-service request chain with fixed per-service processing times, allowing for controlled and repeatable analysis of end-to-end request latency under varying load conditions. This design minimizes application-level variability and isolates the effects of the service mesh and instrumentation.

Workloads are generated using wrk2, which produces constant-rate traffic at specified offered loads, ensuring stable and comparable measurements across experiments. Each configuration is executed for 180 seconds and repeated three times to account for run-to-run variability.

We evaluate three tracing configurations to isolate the effect of instrumentation overhead:

- No-trace: baseline system without instrumentation
- Kernel trace: MeshTrek's original design using kernel-based eBPF uprobes
- Userspace trace (bpftime): userspace execution model (under development)

Experiments are conducted across offered loads ranging from 2000 to 5000 requests per second (RPS) to capture system behavior under increasing load.

6. Results

6.1 Baseline Behavior

In the absence of tracing, the system exhibits stable and predictable latency behavior as load increases. Median latency (p50) remains relatively low across all offered loads, and tail latency (p99) increases gradually with higher request rates. For example, p99 latency grows from approximately 25 ms at 2000 RPS to around 38 ms at 5000 RPS, indicating a smooth and near-linear scaling trend.

This behavior is consistent with expected system dynamics under increasing utilization, where modest increases in queuing and processing delays lead to gradual tail growth. Overall, the no-trace configuration reflects the natural performance characteristics of the service mesh with measurement-induced perturbation.

6.2 Effect of Kernel Tracing

When kernel-based tracing is enabled, the system exhibits a different behavior in tail latency. While median latency (p50) remains relatively stable, high-percentile latency increases significantly compared to the baseline. At 2000 RPS, p99 latency rises to approximately 35 ms, and continues to grow sharply with load, reaching 49 ms at 4000 RPS.

Furthermore, the system is unable to sustain stable operation at 5000 RPS under kernel tracing, suggesting that instrumentation overhead contributes to early saturation.

7. Discussion

These results demonstrate a clear divergence between baseline and instrumented system behavior. In the no-trace configuration, latency scales smoothly with load, reflecting the inherent performance characteristic of the service mesh. In contrast, kernel-based tracing introduces additional variability in request processing times, disproportionately impacting slower requests and amplifying tail latency.

The behavior is consistent with the overhead of eBPF uprobes. Each probe execution results in context switches between userspace and kernel space, and at high request rates, these per-event costs accumulate significantly. As a result, tracing overhead exacerbates queuing and scheduling delays within the proxy, leading to non-linear growth in tail latency.

Importantly, these findings highlight that the measurement system itself perturbs the system being measured. The inflated tail latency observed under kernel tracing does not solely reflect application or service mesh behavior, but is instead partially induced by in the instrumentation mechanism. This distortion reduces the fidelity of latency measurements, particularly in high-load scenarios where accurate tail latency characterization is critical.

These observations motivate the use of userspace eBPF execution via bpftime. By eliminating kernel transitions, bpftime is expected to reduce instrumentation overhead and mitigate the observed tail latency amplification, enabling more accurate and scalable latency measurement.

References

[1] Y. Zheng et al., “bpftime: Userspace eBPF runtime for uprobe, syscall, and kernel-user interactions,” arXiv:2311.07923, 2023.

[2] Envoy Project Authors, “Life of a request,” [Online]. Available: https://www.envoyproxy.io/docs/envoy/latest/intro/life_of_a_request. [Accessed: Apr. 24, 2026].

[3] Envoy Project Authors, “Envoy Proxy,” [Online]. Available: https://www.envoyproxy.io/. [Accessed: Apr. 24, 2026].

[4] J. Shen et al., “Network-Centric Distributed Tracing with DeepFlow: Troubleshooting Your Microservices in Zero Code,” in Proc. ACM SIGCOMM, 2023. https://doi.org/10.1145/3603269.3604823.

[5] Service Mesh Performance Authors, “MeshMark,” [Online]. Available: https://smp-spec.io/meshmark. [Accessed: Apr. 24, 2026].

[6] eBPF Foundation, “What is eBPF? An introduction and deep dive into the eBPF technology,” [Online]. Available: https://ebpf.io/what-is-ebpf/. [Accessed: Apr. 24, 2026].

[7] X. Zhu et al., “Dissecting Overheads of Service Mesh Sidecars,” in Proc. ACM Symp. Cloud Computing (SoCC), 2023, pp. 142–157. doi: 10.1145/3620678.3624652.

[8] Google, “Kubernetes,” [Online]. Available: https://kubernetes.io/. [Accessed: Apr. 24, 2026].

[9] Istio Authors, “Istio,” [Online]. Available: https://istio.io/. [Accessed: Apr. 24, 2026].

[10] X. Zhu et al., “High-level programming for application networks,” in Proc. USENIX NSDI, 2025, pp. 915–935.