Dissecting Overheads of Service Mesh Sidecars

Xiangfeng Zhu
University of Washington

Guozhen She
Duke University

Bowen Xue
University of Washington

Yu Zhang
ByteDance Inc.

Yongsu Zhang
ByteDance Inc.

Xuan Kelvin Zou
ByteDance Inc.

XiongChun Duan
ByteDance Inc.

Peng He
ByteDance Inc.

Arvind Krishnamurthy
University of Washington

Matthew Lentz
Duke University

Danyang Zhuo
Duke University

Ratul Mahajan
University of Washington

ABSTRACT

Service meshes play a central role in the modern application ecosystem by providing an easy and flexible way to connect microservices of a distributed application. However, because of how they interpose on application traffic, they can substantially increase application latency and its resource consumption. We develop a tool called MeshInsight to help developers quantify the overhead of service meshes in deployment scenarios of interest and make informed trade-offs about their functionality vs. overhead. Using MeshInsight, we confirm that service meshes can have high overhead—up to 269% higher latency and up to 163% more virtual CPU cores for our benchmark applications—but the severity is intimately tied to how they are configured and the application workload. IPC (inter-process communication) and socket writes dominate when the service mesh operates as a TCP proxy, but protocol parsing dominates when it operates as an HTTP proxy. MeshInsight also enables us to study the end-to-end impact of optimizations to service meshes. We show that not all seemingly-promising optimizations lead to a notable overhead reduction in realistic settings.

CCS CONCEPTS

• **Networks → Network performance analysis**.

KEYWORDS

Service Mesh, Microservices, Cloud computing

SoCC '23, October 30–November 1, 2023, Santa Cruz, CA, USA

ACM ISBN 979-8-4007-0387-4/23/11.
https://doi.org/10.1145/3620678.3624652

ACM Reference Format:
Xiangfeng Zhu, Guozhen She, Bowen Xue, Yu Zhang, Yongsu Zhang, Xuan Kelvin Zou, Xiongchun Duan, Peng He, Arvind Krishnamurthy, Matthew Lentz, Danyang Zhuo, Ratul Mahajan. 2023. Dissecting Overheads of Service Mesh Sidecars. In *ACM Symposium on Cloud Computing (SoCC '23), October 30–November 1, 2023, Santa Cruz, CA, USA.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3620678.3624652

1 INTRODUCTION

Service meshes are becoming the *de facto* communication substrate for cloud applications. A survey by the Cloud Native Computing Foundation (CNCF) found that 90% of the organizations are using or evaluating them [29]. Service meshes solve important problems related to communication among loosely-coupled microservices—the dominant paradigm for cloud applications [1, 42, 50]—including discovering where services are located, establishing secure connections, and handling communication failures. They also offer advanced capabilities such as rate limiting, load balancing, and telemetry, via additional message processing filters.

However, the overhead of service meshes is a key concern for both their users and developers [4, 23, 35, 68]. In most common scenarios where service meshes are employed, application traffic traverses software proxies called *sidecars*, which increases request latency and consumes more resources. Service meshes can add tens of milliseconds to request latency in some settings [4], and they can consume multiple CPU cores even at moderate load [23]. These overheads can degrade the end-user experience, increase operational costs, and decrease revenue [40, 52].

Today, the only way users can learn service mesh overhead is by running their application with and without service meshes and measuring the increase in latency or CPU usage. This black box approach, employed by some studies [4, 23, 28, 68] and the MeshMark tool [13], has significant limitations. There are many ways to configure a service mesh,

each with different performance implications, and it is infeasible to measure them all. Thus, application developers cannot evaluate functionality-performance trade-offs and discover the best way to configure the service mesh for their specific application and deployment environment.

The concerns around the performance of service meshes and the lack of alternatives to black box measurements is apparent in the documentation of Envoy (a popular sidecar) [20]:

> We are frequently asked how fast is Envoy? or how much latency will Envoy add to my requests? The answer is: it depends. Performance depends a great deal on which Envoy features are being used and the environment in which Envoy is run. ··· We encourage users to benchmark Envoy in their own environments with a configuration similar to what they plan on using in production.

The black box measurement approach has another limitation. There are several ongoing efforts on optimizing service mesh overhead [3, 12]. Without a detailed accounting of the underlying contributors to overhead, it is hard to quantify the effectiveness of such optimizations, especially as it relates to the end-to-end impact on a wide range of applications.

Instead of a blackbox approach, MeshInsight models the sidecar's operation as a combination of independent components (e.g., read, write, and IPC) and key aspects of the workload. By characterizing individual components, we can estimate the overhead of a sidecar based on the components used in a given configuration. Then, given workload characteristics such as the call graph, request rate, and message sizes, MeshInsight can estimate the end-to-end overhead for the application *without needing to deploy the application under that configuration*. Similarly, if a service mesh developer wants to evaluate an optimization that reduces the overhead of some component(s), MeshInsight can estimate its end-to-end performance impact across a wide variety of application workloads.

Using MeshInsight, we study the overhead of Envoy. Envoy is the dominant sidecar, used by many service meshes [9, 10, 14, 26, 27, 45] including Istio [24], one of the most popular service meshes today [2]. Our experiments confirm that service meshes can have substantial performance penalties. Across two popular benchmark applications [8, 42], depending on the configuration, request latency increases by 27-269% and CPU usage increases by 42-163%. In a large dataset of microservice-based applications [58], we find that using Envoy increases latency by up to 100 ms and consumes 200 more virtual CPU cores for a quarter of the call graphs. We also find that, for a given service mesh configuration, the overhead for different applications varies by multiple orders of magnitude. Such high variation based on service mesh configuration and on application characteristics validates the need for a tool that developers can use for their specific deployment scenarios.

MeshInsight's compositional approach provides insight into the sources of overheads. We find that when configured as an HTTP or gRPC proxy, protocol parsing alone represents 63-77% of the total overhead. When configured as a TCP proxy, most overhead stems from inter-process communication (IPC) and socket write operations. The overhead of individual filters (i.e., network functions) varies significantly. Some increase latency by as little as 3% atop the baseline overhead, while others increase it by as much as 85%.

We also use MeshInsight to evaluate the end-to-end overhead reduction for service meshes using two Linux kernel features: 1) Unix domain sockets in place of TCP connections for IPC, and 2) zero-copy writes for TCP sockets. Given that IPC and socket writes are substantial contributors to overhead, we wanted to understand if these features help reduce overhead. We model the performance of our components in the presence of these two features and evaluate their end-to-end impact using the large microservices dataset [58]. We find that Unix domain sockets are helpful, reducing the average latency overheads by 27% and the CPU overheads by 18%, for TCP proxy. But, surprisingly, zero-copy socket writes have negligible improvements. For small message sizes that are common to microservice workloads, the additional system calls in implementation negate the savings from avoiding the copy operation.

We make two key contributions in our work:

- We develop a tool that application and service mesh developers can use to quantify overhead. It allows application developers to make informed decisions regarding trade-offs between performance and functionality while enabling service mesh developers to assess the overall impact of their optimizations. MeshInsight accomplishes this without requiring the deployment of each setting in a production environment.
- We conduct the first systematic study of service mesh latency and CPU overheads across a wide range of realistic microservices. It confirms that the overhead is generally high, but it also shows that it varies significantly based on the sidecar configuration and application characteristics. Our study also provides a detailed accounting of performance overheads and informs future optimizations.

Our tool and the results of our study will inform work on improving the performance and reducing the resource consumption of service meshes, which now play a central role in the modern application ecosystem. MeshInsight is available at https://github.com/UWNetworksLab/meshinsight

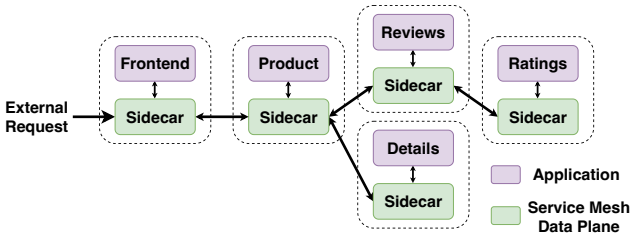


Figure 1: Bookinfo application with a service mesh. Image Redrawn from [11].

2 BACKGROUND

Service meshes emerged to solve challenges that arose when applications moved from monoliths to the microservices architecture. Instead of being a large binary, applications are composed of multiple microservices. Figure 1 shows an example where the BookInfo application from Istio [11] uses five microservices. Each microservice is run as an independent process (or container), often on different hosts, and can be scaled independently. Such decomposition enables agile application lifecycle management, fault-tolerance, scalability, and reuse of building blocks across applications [44]. It is common for modern applications to use tens of services [1, 44, 50].

But decomposing applications into multiple microservices creates new problems as well. What used to be a function call now becomes an RPC over the network. Developers must figure out how services discover, communicate, and authenticate with each other. They must also figure out how to monitor and secure inter-service communication and how to handle failures. Early adopters of microservices built custom communication frameworks to solve these problems [19, 21]. Service meshes emerged to solve them in a reusable manner, and they further evolved to provide other functions such as rate limiting and load balancing.

The convenience of service meshes comes at the cost of performance and resource overhead. The overhead could be in the control plane or the data plane. The control plane handles service discovery, metric collection, and certificate management, and it appropriately configures the data plane. For example, in Figure 1, to balance load across multiple instances of the Product service, the Frontend data plane can spread Frontend-to-Product connections across different Product instances (multiple instances not shown in the figure). The service mesh control plane tracks where all the instances are running and configures the Frontend data plane accordingly. We focus on the data plane overhead as it impacts every request and is on the critical path of user experience.

The most common implementations of the service mesh data plane use software proxies called *sidecars*. The sidecar is a separate process that is co-located with each instance of

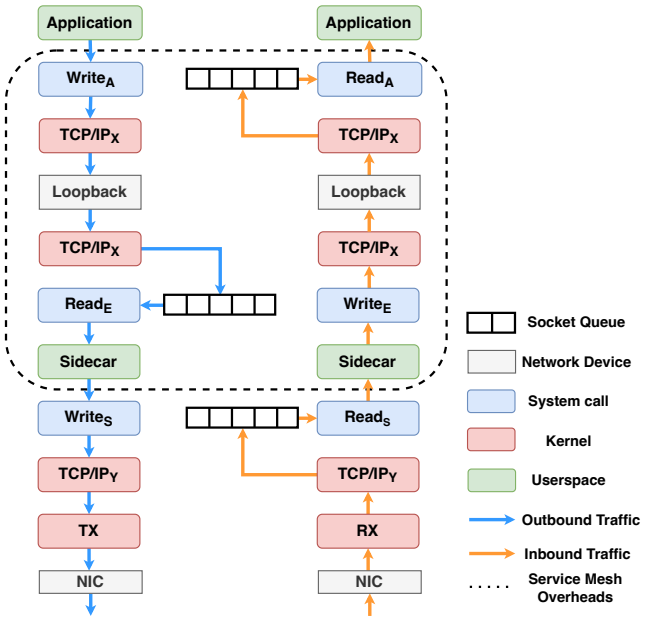


Figure 2: Outbound (left) and inbound (right) messages with a service mesh. Write$_A$/Read$_A$ and Write$_S$/Read$_S$ denote, respectively, the write/read of the application and its sidecar. TCP/IP$_X$ denotes the TCP/IP stack of network namespace X. The figure assumes that the application uses the same event notification interface (i.e., epoll). The extra steps added by the service mesh are in the dashed box.

an application service, which enables the sidecar to mediate all of the service's network access to apply network policies, enforce encryption, and log statistics. Recent service mesh proposals (e.g., Cilium [12] and Ambient Mesh [22]) aim to inline (in the kernel) some data plane functionality and propose per-host proxies. However, kernel acceleration does not eliminate sidecars, which are still needed for complex functionality, and sidecars have some significant benefits over per-host proxies. Thus, sidecars are likely to persist [16, 17] for the foreseeable future. We focus on the sidecar model and leave the analysis of hybrid models to future work.

Sidecar Data Path Figure 2 shows the data path for both outbound and inbound traffic. In this paper, while we apply our modeling approach and analysis to Envoy [18], other sidecar proxies [31] share this architecture. During initialization, the control plane adds iptable rules that redirect all inbound and outbound traffic to the sidecar. As a result, logical connections between microservices are broken into three separate connections: two connections between the sidecars and their associated microservices, and one connection between the sidecars. When the application sends a message,

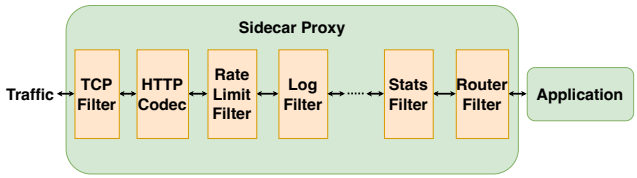


Figure 3: Message processing inside a sidecar. Filters operate on individual messages and form a chain.

it executes a write system call. The kernel network stack and the loopback device process the message and notify the sidecar. The sidecar then reads the data from the kernel, processes it, and writes it back to the kernel. Finally, the NIC driver transmits the message. Similar to the inbound traffic, upon receiving a message, the sidecar intercepts the message before it is passed to the application.

The exact processing done by the sidecar depends on its configuration; Figure 3 shows the general data flow. When a message arrives, it is parsed based on the protocol of choice (e.g., TCP, HTTP, gRPC). In TCP mode, traffic is treated as an opaque TCP stream; in HTTP and gRPC mode, messages are parsed as per the protocol, which enables additional functionality specific to the protocol. After the message is parsed, it is processed by one or more *filters*. Filters are short programs that process individual messages and implement network functions like traffic monitoring, rate limiting and fault injection. Envoy filters can be written using 1) C++ code, 2) Lua scripts, or 3) WebAssembly modules. All of Envoy's 40+ built-in filters are C++-based, while application developers tend to write custom filters using Lua or WebAssembly.

Given the data paths in Figures 2 and 3, we can see a number of sources of overhead. First, without a sidecar, to send a buffer, the kernel copies the application buffer into a kernel buffer, which the NIC can subsequently access through Direct Memory Access (DMA). With a sidecar, the buffer must additionally be copied to the sidecar buffer and then back into a kernel buffer (resulting in two extra copies). Second, there are many additional system call invocations, such as the sidecar waiting for data through epoll and reading and writing buffer from/to kernel. Finally, using sidecars incurs extra IPC invocations (e.g., the loopback interface in Istio). In addition, sidecars may need excessive computation on the buffer, including parsing the data stream into data structures for HTTP, JSON, and RPC data formats.

3 MODELING SERVICE MESH OVERHEAD

Our goal is to characterize the overheads of service meshes in a way that supports two classes of developers. First, we want to enable application developers who are looking to deploy service meshes to understand overhead as a function of service mesh configuration (e.g., proxy and filter configurations), so they can appropriately trade off functionality and overhead. Second, we want to enable service mesh developers to understand the impact of their optimizations (e.g., zero-copy writes) on a wide range of real-world applications without having to necessarily implement them fully. Neither class of developers is well-served by currently available tools and techniques.

A key challenge we face is the large operational space of service meshes. An application may be running Envoy in one of many possible ways, each with different performance implications. There are at least three dimensions of variations: *i*) type of proxy (e.g., TCP, HTTP, and gRPC); *ii*) which filters are used; *iii*) application workload, where the salient characteristics are message sizes and rates. These variations, and their combinatorial combinations, mean that a black-box approach to measuring overhead is a non-starter.

We must instead model the overhead of finer-grained components and then compose the individual component overheads to predict overhead for a given deployment scenario. Our models are a concise representation of performance overheads over the entire operation space, including both the service mesh configuration and application workloads. This modeling approach allows us to reason about a service mesh's large operation space. For example, for an application developer to choose an appropriate configuration for a service mesh, instead of benchmarking every possible configuration, we can use our models to quickly predict the performance overheads for *any* service mesh configuration for their workloads.

This approach is necessarily an approximation. It characterizes each component in isolation, ignoring interactions that may arise when they contend for resources (under high utilization). The overhead of contention is hard to model and sensitive to co-located workloads and background system activity [41, 63]. Consequently, MeshInsight predicts the best-case overhead (i.e., a lower bound) and cannot be used to predict tail latency. As we show later, MeshInsight's predictions track actual overhead well enough to effectively distinguish between the configurations with different overheads.

MeshInsight overview To be practical, we must carefully choose the granularity and nature of the components. If they are too fine-grained, accurately characterizing their overhead (and composing them) will be difficult; if they are too coarse-grained, we'll suffer from the same challenge as with black-box measurements. The overheads of the chosen components should also be largely independent, allowing us to easily compose their overheads to estimate total overhead.

	Component	Description
1	IPC	Data transfer between sidecar and application
2	Read	Read syscall and data copy from kernel to user space
3	Write	Write syscall, data copy from user to kernel space, and network stack's TX processing
4	Notification	I/O event notification processing
5	Sidecar Parsing	Protocol parsing in sidecar
6	Sidecar Filter	Filter chain processing in sidecar
7	Sidecar Other	Baseline processing in sidecar

Table 1: Components in MeshInsight's performance model. The total performance overheads of a service mesh are the sum of these components.

Table 1 shows the component-level breakdown we use. The first four represent the sidecar's interactions with the application and the kernel: 1) inter-process communication (e.g., loopback) between the application and its sidecar proxy, 2-3) writes and reads from the sidecar proxy to the kernel, which also include the data copies (note that write also includes the TX's TCP/IP processing), and 4) blocking waits on socket ready notifications (e.g., from `epoll`). The next three breakdown message processing inside the sidecar: 5) parsing the messaging protocol (e.g., HTTP), and 6) processing done by user-configured filters; and 7) baseline processing by the sidecar to move packets between the two components above.

Figure 4 shows the workflow of MeshInsight. It has an offline profiling phase and an online prediction phase. The offline phase generates performance profiles of components, and the online phase predicts the service mesh overhead based on these profiles, sidecar configuration, and application workload.

MeshInsight models the performance of a component as a function of message size and request rate because these two workload properties are the primary determiners of performance. In our experience, simple linear functions of these two properties suffice (see next section for details). The profile of a component is specific to the *platform*, which includes the hardware (e.g., CPU, memory), OS, and Envoy version, but independent of individual microservices. Overhead predictions are made only for previously profiled platforms.[1] While we focus on Envoy in this work, our approach can be extended to other existing service mesh proxies (e.g., linkerd) with minor configuration changes.

The online prediction phase uses performance profiles from the offline phase to provide performance predictions in the context of a specific application deployment scenario. These predictions are based on an extended call graph (described in Section 4.2) that captures application behavior.

4 MESHINSIGHT DESIGN

We now describe the design of MeshInsight in more detail.

4.1 Building Component Profiles

In the offline phase, for each component in Table 1, MeshInsight builds performance profiles that characterize components' message processing latency and CPU usage as a function of message size and rate. The profiler exercises the components under a few different settings and then extrapolates their performance to other settings.

We conduct two types of profiling runs: *i*) sidecar configured as TCP, HTTP, or gRPC proxy, with no additional filters; and *ii*) sidecar configured with only the filter(s) of interest. Profiling uses an echo server paired with a sidecar. We use wrk [32] and wrk2 [33] for load generation as well as high-precision measurement of end-to-end latency. To measure request processing latency and CPU without contention effects, the wrk client generates requests such that at most one request is outstanding at any time.

To quantify the overhead of individual components during a run, we exploit the fact that all components (except filters–see below) correspond to specific kernel- or user-space functions. We measure the latency of each component using a modified version of BCC's funclatency [15], an eBPF-based tool that uses Kprobe and Uprobe to monitor time spent on a function. We measure CPU usage of each component using the standard sampling technique [46], which allows us to quantify the CPU consumption of any function. We identify the function for a component using a mix of the name, the input/output, and process ID.

Filters present two wrinkles in this process. First, most of them do not have a known function. We measure the overhead of a filter by subtracting the overhead of a setup without the filter from an otherwise identical setup with it. Second, the overhead of some filters depends significantly on

[1]This does not impact application developers, who can use MeshInsight on their own platforms. But service mesh developers may want to understand overhead on platforms that they do not have access to. To support them, we are developing a shared repository of profiles for common platforms (e.g., AWS instances) to enable the reuse of profiling data.

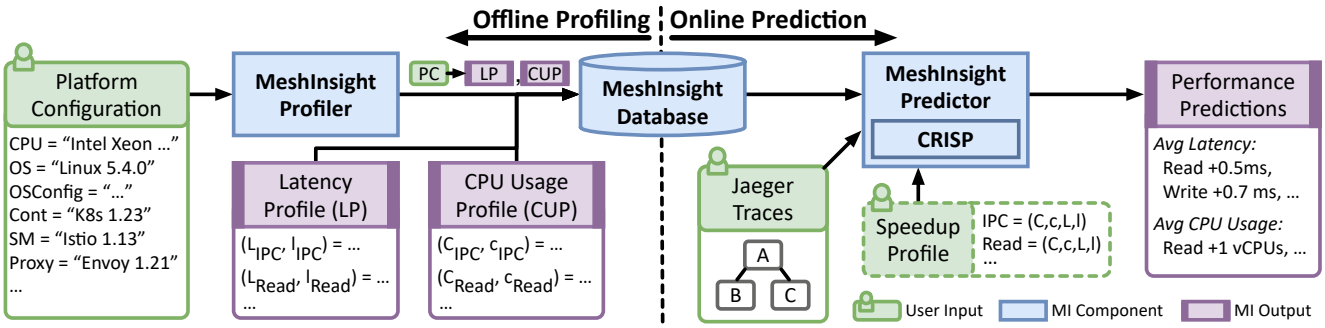


Figure 4: Overview of MeshInsight (MI) workflow. The MeshInsight Database stores performance profiles associated with a hardware and software platform configuration, which are generated by the MeshInsight Profiler during the offline profiling stage §4.1. In the online phase §4.2, these profiles are used by the MeshInsight Predictor, in conjunction with Jaeger Traces provided by the user, to compute latency and CPU performance predictions for application deployment. The user can optionally provide a speedup profile, which MeshInsight uses to adjust the predictions accordingly.

certain configuration parameters. For instance, the overhead of the Rate Limit filter, which limits the network traffic to a service, depends on whether the developer needs to limit traffic rate on the entire service mesh (global rate limiting) or on per service instance basis (local rate limiting). Likewise, the overhead of the Tap filter, which logs traffic, depends on whether the log is written to a file or sent over the network. In our profiling, we treat such filters as different components.

Data from the profiling runs and the following assumptions (empirically validated in §5) enable us to estimate the sidecar's overhead in any setting.

- **A1:** The total latency and CPU overhead of the sidecar is the sum of components' overhead.
- **A2**: Latency overhead is a linear function of message size (see below).
- **A3**: CPU overhead is a linear function of message size and proportional to message rate.

Thus, to estimate the overhead of the sidecar in a configuration with multiple filters, which has not been measured directly, we can add the overhead of the base configuration without filters and the overheads of filters that are employed.

To estimate the latency of a component x for a message size s, per prior work [54, 72], we use this linear function: $L_x + s \times l_x$, where L_x is the base message processing latency and l_x is the per-byte latency. We assume that latency for processing a message does not vary based on the request rate.

To estimate CPU consumption of a component x for request rate r, we use the following linear function: $r \times (C_x + s \times c_x)$, where C_x denotes the baseline per-message CPU usage and c_x denotes the per-byte CPU usage.

The impact of message size captured in the two equations above assume that a message is processed by each component (e.g., read or written) as one unit. This assumption may be violated for huge messages that are split into multiple units. Size threshold at which a message is split may be overridden by applications or Envoy, but is typically at least a few KB; it was always above 4KB for platforms that we have experimented with. The implication for MeshInsight is that it will underestimate the overhead for messages that are split; the actual latency and CPU cost are higher for such messages. We quantify this underestimation in §5.3. Fortunately, the vast majority of message sizes are small [61, 69, 74], and the impact of our modeling approximation is therefore minimal.

To estimate (L_x, l_x) and (C_x, c_x), MeshInsight profiles components using five different message sizes (100B, 1KB, 2KB, 3KB, 4KB) and four different request rates (25%, 50%, 75%, and 100% of the application's maximum achievable throughput). We then apply linear regression to the collected data. These two tuples represent the latency and CPU profile of a component for a particular platform. The message size and rate are chosen empirically; we find that these choices have minimal impact on profile accuracy given sufficient data points.

4.2 Predicting Overhead

Application developers can use MeshInsight to estimate service mesh overhead in any deployment scenario of interest by providing an *extended call graph* (ECG). The ECG captures the details of the deployment and interactions among microservices in response to a request.

Formally, an ECG is a tuple (V, P, S, G), where $V = \{v_1, v_2, ..., v_n\}$ is a set of vertices representing microservice instances;

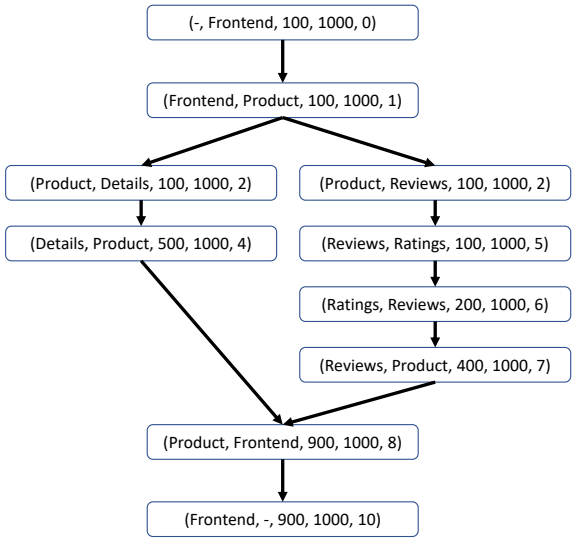


Figure 5: An example call graph for the Bookinfo application.

P is a map from microservice instances to platforms; and S is map from microservices instances to configurations (i.e., protocol and filters). G is a DAG (directly acyclic graph) of microservice invocations. Each node in the graph is an invocation, and the edge represents an invoked-after relationship.

An invocation k is a tuple $(u_k, d_k, s_k, r_k, t_k)$, where u_k is the calling microservice (empty if called externally), d_k is the microservice invoked, s_k, r_k are the expected size (in bytes) and rate of messages (in requests/second) along this invocation, and t_k is the timestamp of the invocation without a service mesh. ECGs are extracted automatically with an openTelemetry-standard distributed tracing tool like Jaeger [25] and a trace analysis tool CRISP[76]. Since Jaeger traces do not contain the message size and rate for each invocation, we collect these using custom scripts. MeshInsight can also handle dynamic call graphs by using all available Jaeger traces and generates a probability distribution of the overhead for an API.

Figure 5 shows an example of G for the Bookinfo application in Figure 1. An external client calls Frontend, which in turn calls Product. The Product service calls Reviews and Details in parallel. Reviews calls Ratings and responds to Product after getting the response. Product responds to Frontend after both Details and Reviews respond. Finally, Frontend responds to the external client.

Generating Predictions. Given an ECG, MeshInsight estimates the latency and CPU overhead. It starts by computing the overhead of each invocation. For an invocation $(u_k, d_k, s_k, r_k, t_k)$, the overhead is based on messages of the given size and rate leaving the service at u_k and entering the service at d_k. The sidecar configuration for these services tell us which components are exercised. We compute the component-level overhead (using s_k and r_k) and then sidecar-level overhead by summing component overheads. In some configurations, not all components are subjected to the same s_k and r_k. For instance, if a fault injection filter, which can be configured to drop some messages, is present, downstream filters will see a lower message rate. Currently, we ignore such intra-sidecar variations, though our model can be extended to account for them if needed.

MeshInsight computes end-to-end request-level overheads using sidecar-level overheads computed above. For the CPU, it simply adds all the sidecar-level overheads. For latency, simple addition does not work because computations can happen in parallel, and we thus need critical path analysis. In the Bookinfo example above, the end-to-end latency depends on which of Details or Reviews is slower to respond to Product, and the latency of the faster one is not critical. To this end, MeshInsight uses t_k to compute the critical path of the application and only report the end-to-end latency overheads along the critical path. It is important to note that introducing a service mesh might alter an application's critical path. To accommodate such changes, MeshInsight executes a critical path analysis (i.e., CRISP) after adding the overhead predictions for each hop to the ECG.

Quantifying the impact of service mesh optimizations. The prediction techniques described above can be used by service mesh developers to estimate the end-to-end impact of their optimizations. To enable this estimation, service mesh developers need to provide information on the impact of their optimization for the component(s) they have optimized. That is, they need to update the performance profiles. This update may be based on the estimated impact of their planned optimization (which has not been implemented yet). For example, the developer may estimate that their optimization will lower the baseline write overhead by 50%. Alternatively, new performance profiles may be based on running the MeshInsight profiler after implementing the optimization.

Once information on new performance profiles is provided, MeshInsight can estimate the overhead of the new system and how much improvement it brings compared to the original.

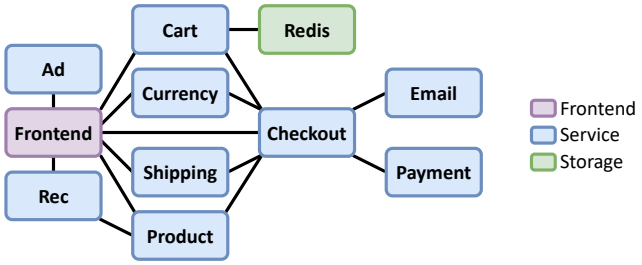


Figure 6: Online Boutique application [8].

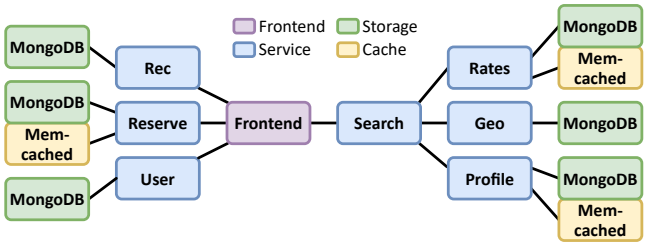


Figure 7: Hotel Reservation application [42].

5 CHARACTERIZING SERVICE MESH OVERHEAD

We now use MeshInsight to characterize the overhead of service meshes for benchmark applications and validate the accuracy of its predictions. We study overhead for both basic deployments and when filters are added. We also study how individual components contribute to the overhead and the impact of message size and rate. In the next section, we demonstrate how MeshInsight can help application and service mesh developers.

Our experiments use Cloudlab [37] machines with two 16-core Intel Xeon Gold 6142 CPUs (2.6 GHz) and 384GB RAM, Ubuntu 20.04 LTS (Linux kernel v5.4.0), Kubernetes v1.12.5, Istio v1.13.0, and Envoy 1.21.0. We disable Turbo-Boost, CPU C-states, and dynamic CPU frequency scaling to reduce measurement variance.

Application Benchmarks. To characterize the overhead of service meshes on realistic applications, we consider two popular microservices benchmarks: Online Boutique [8] and Hotel Reservation [42]. Online Boutique (Figure 6) has 11 microservices. It is a web-based e-commerce application where users can browse items, add them to the cart, and purchase them. Microservices are written in different languages (Python, C#, Java, and Go) that communicate using gRPC. Hotel Reservation (Figure 7) has 17 microservices and supports searching for hotels using geolocation, making reservations, and providing hotel recommendations. All microservices are implemented using Go and communicate using gRPC.

We deploy these applications on multiple Cloudlab hosts and consider two deployment scenarios: TCP and gRPC. (HTTP cannot be used because the applications are gRPC-based.) In TCP mode, all sidecars are configured as a TCP proxy that relays the message to the application service. In gRPC mode, the sidecars parse the gRPC stream and collect basic application-level metrics.

We consider three queries for each benchmark. For hotel reservations, the queries are:

(1) **(Q1) User**: Checks the username and password. Calls User and its MongoDB.

(2) **(Q2) Search**: Returns available hotels based on location and check-in/check-out dates. Calls Search, Rates, Geo, Profile, Reserve and their MongoDB and Memcached storage.

(3) **(Q3) Reservation**: Reserves a hotel room. Calls Reserve, User and their MongoDB and Memcached storage.

For online boutique, the queries are:

(1) **(Q1) Index**: Returns the home page. Calls Currency, Products, Cart and Ad.

(2) **(Q2) Browse_Product**: Returns product details. Calls Product, Currency (twice), and Cart.

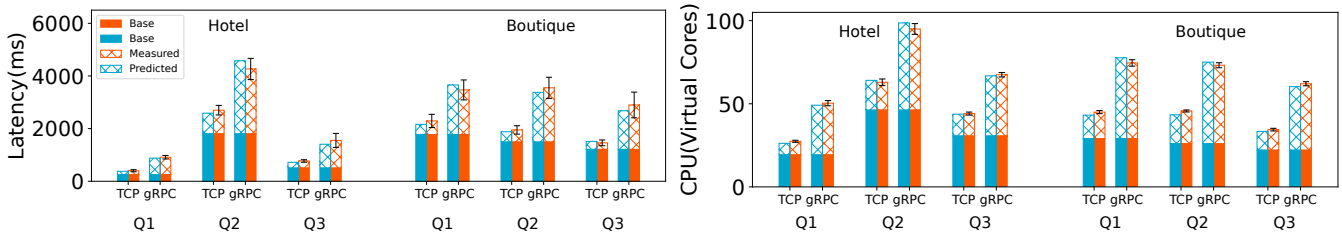
(3) **(Q3) View_Cart**: Returns the user's shopping cart. Calls Cart, Shipping, Product, and Currency.

We derive the ECG (extended call graph) for each query and intentially force a cache miss on each memcached access for a deterministic call graph. The message sizes in each ECG are based on the actual traffic of the application; we find that most messages are small (a few hundred bytes). We set the request rates close to the maximum the machine can sustain in gRPC mode for each query.

Figure 8 shows the base (without the service mesh) latency and CPU usage of the application along with predicted and measured overheads. The measured overhead is the latency or CPU usage of running the application with the service mesh minus that of running it without the service mesh. Latency is end-to-end, measured at the client.

We see that service meshes can be a significant source of overheads. When operating in gRPC mode, it can increase latency by up to 269% and consume 163% more CPU. In TCP mode, the overhead is lower but still substantial–latency increases by up to 61% and CPU usage by up to 92%. The next section sheds light on why these two modes behave differently.

Service mesh overhead will increase further as filters are added (see below). These high overheads, and differences in



(a) Latency.

(b) CPU Usage.

Figure 8: Predicted and measured overhead for Online Boutique and Hotel Reservation applications. Base denotes latency and CPU usage when the application is run without a service mesh. The error bars for measured overhead are standard deviations.

overheads in different settings, is why MeshInsight is needed to enable developers to appropriately trade-off performance and functionality. For instance, TCP mode might be enough for most services, and gRPC mode is limited only to services where extra control or visibility is required.

In Figure 8, we can also see that MeshInsight predictions track well the overhead for all queries across both benchmarks even though their individual performance varies significantly. Predicted overhead tends to be lower than measured overhead. Underestimation is more pronounced for gRPC. The primary factor is that in gRPC (and HTTP) mode Envoy adds extra headers and increases the response size. Larger responses slightly increase the latency and CPU use of downstream components that are outside of the service mesh (e.g., client-side processing). We currently do not model this effect.

5.1 Breaking down the Overhead

To shed light on sources of overhead, Table 2 shows how much each component contributes when Envoy is run in different protocol modes and without any filters. This experiment uses a synthetic application (echo server) with 100-byte messages at 30K requests per second. We use a synthetic application so that we can completely control the workload and can study the HTTP mode as well (which benchmark applications do not allow). For benchmark applications, relative contributions of different components in TCP and gRPC modes are roughly similar to those in Table 2.

We can draw several conclusions from this data. First, using HTTP and gRPC is substantially more expensive than TCP. The additional overhead of HTTP is roughly 4x for latency and 3x for CPU; for gRPC it is 5x for latency and 4x for CPU. The bulk of this additional overhead stems from protocol parsing, accounting for 63-77 % of the total overhead.

Parsing overhead is unfortunate because the application code will spend resources on parsing as well. Because of the way sidecar data path is organized, its parsing effort cannot be shared with the application. If enabled, such reuse will have a notable impact on the architecture of service meshes.

Second, we see that IPC overhead is notable (30% for TCP) and notification overhead is small (3% for TCP). This observation implies that asynchronous processing between the application and sidecar is not expensive by itself, but the default IPC mechanism in Envoy is expensive. We can tackle this overhead by either putting the sidecar in the same process as the application. However, this can have security implications because a malicious application may circumvent the network policies. In addition, upgrading a sidecar more complicated would require recompiling the application. Another option is to use a more lightweight IPC mechanism. We will consider the second option in the next section.

Third, some components have disparate impacts on latency and CPU. This disparity is most pronounced for "Sidecar Other", where its contribution to CPU overhead is far greater than its contribution to latency, but occurs for other components as well (e.g., Write). It implies that some optimizations may impact one type of overhead but not the other, and developers need to be careful that optimizing for latency does not hurt CPU and vice versa.

5.2 Impact of Filters

We find that sidecar filters can be quite expensive, even when configured as a no-op. We also empirically confirm that, as assumed by MeshInsight's model, their overhead is additive.

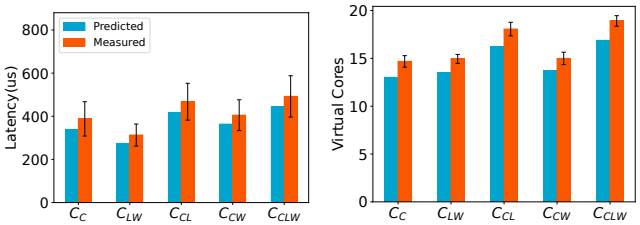
We study five different filters, covering all three ways to write an Envoy filter: 1) Fault Injection: a built-in, C++ filter that helps test the resilience to communication failures; 2) (Local) Rate Limit: a built-in, C++ filter that rate limits traffic to a service instance. 3) Tap (File): a built-in, C++ that records

	Latency (us)			CPU Usage (Virtual Cores)		
	TCP	HTTP	gRPC	TCP	HTTP	gRPC
IPC	12.17 (30%)	12.73 (8%)	13.40 (7%)	0.46 (14%)	0.50 (5%)	0.57 (4%)
Read	8.71 (21%)	9.19 (6%)	9.24 (5%)	0.27 (8%)	0.28 (3%)	0.31 (2%)
Write	14.11 (34%)	13.57 (8%)	14.91 (8%)	0.48 (14%)	0.49 (5%)	0.57 (4%)
Notification	1.27 (3%)	1.32 (1%)	1.28 (1%)	0.26 (8%)	0.27 (3%)	0.26 (2%)
Sidecar Parsing	-	122.07 (74%)	147.03 (77%)	-	6.07 (63%)	9.26 (69%)
Sidecar Other	4.83 (12%)	6.29 (4%)	5.81 (3%)	1.88 (56%)	2.09 (22%)	2.39 (18%)
Total	41.09	165.17	191.67	3.35	9.70	13.36

Table 2: Contribution of different components to the overhead of a single sidecar instance in different protocol modes. The numbers report both inbound and outbound overheads.

	Latency (us)	Virtual Cores
Fault Injection	5.74 (3.5%)	0.20 (1.9%)
Rate Limit	8.19 (5.0%)	0.21 (2.0%)
Tap	156.09 (85.0%)	2.95 (30.4%)
Lua	80.59 (43.9%)	3.18 (30.2%)
WebAssembly	26.30 (14.3%)	0.69 (6.6%)

Table 3: Latency overhead of five filters. The percentage in parentheses denotes the additional overhead atop baseline HTTP mode (without any filters).



(a) Latency. **(b) CPU Usage.**

Figure 9: Prediction results of different filters configurations.

traffic and is configured to log to a file; 4) Lua: a custom, no-op filter written as a Lua script; 5) WebAssemtly: a custom, no-op filter written as a WebAssembly module. We add these filters on Envoy configured in HTTP mode.

Table 3 shows the overhead of each filter inferred by MeshInsight when subjected to the same workload as the previous section (100 byte messages, 30K requests per second). We see that different filters have widely different overheads. The baseline overhead of C++ filter is low, as evidenced by the low overhead of Fault Injection and Rate Limit filters. The overhead of Tap (file) is high because of its interaction with the file system. On the other hand, even no-op Lua or WebAssembly filters have substantial overheads, with Lua being 3x more expensive for latency and nearly 5x more expensive for CPU.

To study the composability of filters' overheads, we consider five different filter configurations, each with a different way to combine filter types: 1) C_C: combines all three types of C++ filters; 2) C_{LW} combines the Lua and WebAssembly filters; 3) C_{CL}: combines the Lua filter with all three C++ filters; 4) C_{CW}: combines the WebAssembly filter with all three C++ filters; and 5) C_{CLW}: combines all five filters.

Figure 9 shows both predicted and measured overheads of each of these combinations. The measured overhead denotes latency and CPU usage with the filters minus that without the filters. We see that filter combination overheads can be quite high when multiple expensive filters are employed (something that the developers must avoid). We also see the predictions of MeshInsight, based on adding individual overheads on top of base HTTP proxy overhead in Table 3, are quite accurate.

5.3 Impact of Message Size and Rate

We now show how sidecar overhead increases with message size and rate. We vary message sizes from 100 bytes to 16KB. The upper end of this range is well beyond the maximum size that we directly profile (4KB). We vary message rates from 10K to 50K requests per second.

Figure 10 plots latency overhead for HTTP proxy without filters. The latency increase is similar for other protocols. We see that latency overhead increases slowly with message size. Going from 100 bytes to 16 KB (which represents a very large message), the latency overhead increases by 53 ms. This increase represents only a 30% increase for HTTP. The presence of filters does not significantly change the impact of message size on latency, as most filters operate on message headers, not payload (which has the most bytes).

We also see in Figure 10 that MeshInsight models the impact of latency increase well, though its prediction accuracy

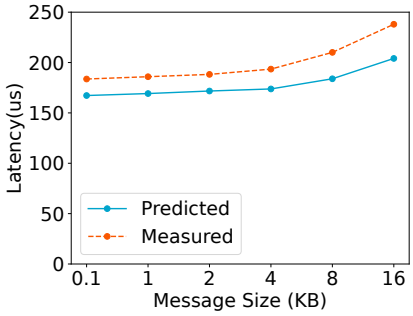
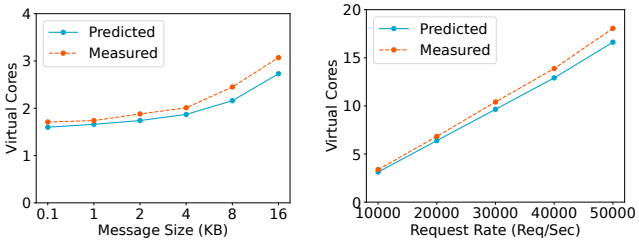


Figure 10: Impact of message size on service mesh latency overhead. X-axis is on log scale.



(a) Message Size. **(b) Message Rate.**

Figure 11: Impact of message size and rate on service mesh CPU overhead of service meshes.

drops for very large messages (which are uncommon [61, 69]). As mentioned earlier (§4.1), the reason for this lower accuracy is that messages larger than a few KB are split into multiple units, which have higher latency and CPU costs.

Figure 11 shows CPU usage for HTTP proxy. We see that the CPU overhead increases linearly with message sizes (for a fixed rate) and linearly with message rate (for a fixed size), and that MeshInsight tracks this increase well. Large messages have a relatively higher impact on CPU usage than they did on latency. CPU usage increases by 62% when the message size increases from 100 bytes to 16 KB.

6 HELPING DEVELOPERS PREDICT OVERHEAD

In addition to characterizing service mesh overhead in detail, MeshInsight has two more use cases: (1) helping application developers determine how to configure service meshes, and (2) helping service mesh developers evaluate potential optimizations.

6.1 Application Developers

Given the application call graph, MeshInsight can predict service mesh overhead for different configurations. With this knowledge, application developers can make the right trade-off between desired functionality and overhead.

We demonstrate this use case using the Alibaba microservice traces [58]. These contain over 20M call graphs from microservices-based applications, collected over 7 days in an Alibaba cluster. While most call graphs have a small number of microservices, 10% of them have over 40 microservices and the largest ones have thousands of microservices. We randomly select 1M call graphs for our experiments. The traces do not contain message sizes or rates; we assume these to be 100 bytes (because service mesh messages tend to be small) and 30K requests per second, which represents a moderate load.

Figure 12 shows the latency and CPU overheads for five different service mesh configurations: three protocol modes without filters, HTTP with filters, and gRPC with filters (all five filters described in §5.2). Since a call graph can have multiple paths, the latency overhead is computed for the critical paths, which we extract from the Alibaba trace based on both invocation's timestamp and response time. We see in Figure 12 that in any given configuration, the sidecar overhead across applications varies by multiple orders of magnitude. Even for the simplest configuration (TCP), latency overhead varies from 0.2 to 100 ms and CPU from 3 to 1000 virtual cores. Thus, different applications are impacted disparately by service meshes.

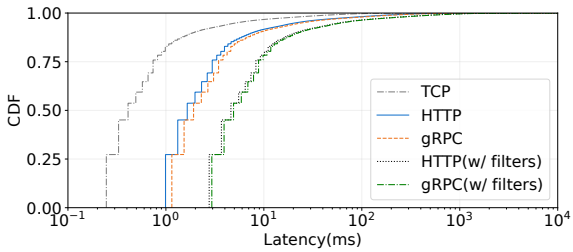
We also see that the overhead of different service mesh configurations varies by an order of magnitude for both latency and CPU. The median latency overhead is 0.2ms in TCP mode but it is 2 ms in gRPC mode with filters, and the 75th percentile varies from 1 to 10 ms. Similarly, the median CPU overhead varies from 20 virtual CPU cores in TCP mode to over 200 in gRPC mode with filters.

These massive variations based on service mesh configuration and application characteristics are why we need a tool like MeshInsight using which application developers can learn the overhead of their specific deployment scenarios of interest.

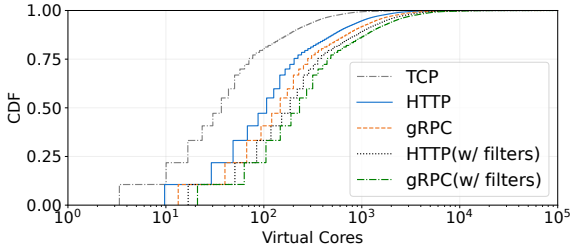
6.2 Service Mesh Developers

MeshInsight enables service mesh developers to judge the impact of potential optimizations. There are several ongoing efforts to optimize service meshes [3, 12]. While such optimizations can be benchmarked in isolation, it is difficult to understand the end-to-end impact on real-world applications. MeshInsight allows service mesh developers to estimate the potential performance changes without excessive software prototyping effort.

To demonstrate this use case, we consider the potential use of two Linux kernel features in Envoy. Porting Envoy to use new kernel features is a big effort and may also introduce



(a) Absolute latency overhead.



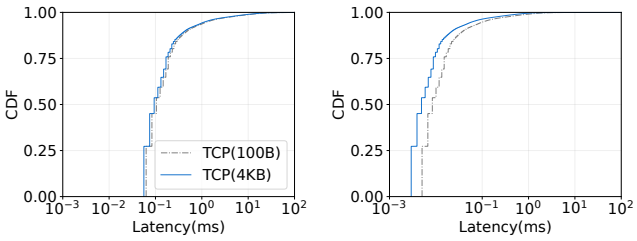
(b) Absolute CPU overhead.

Figure 12: Latency and CPU overhead for application call graphs in the Alibaba trace.

some functional limitations, so the Envoy developers may want to estimate the impact even before taking on this work. All they need to provide MeshInsight is an estimate of new performance profiles for various components. To estimate these speedups we enable these features in the context of a simple sidecar (with 676 lines of C++) that has the same data path architecture as Envoy. We then profile the components to learn their new performance profiles when these features are active. The Linux features we study are:

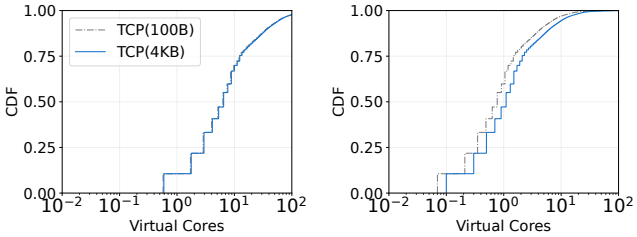
Unix domain sockets In §5.1, we saw that IPC overhead is a significant contributor to overhead, adding at least 11 microseconds to latency and consuming 0.5 virtual cores, representing 30% of latency overhead and 15% of CPU overhead in TCP mode. By default, Envoy-to-application IPC uses a TCP connection, which traverses the TCP/IP network stack for the loopback interface. One potential option to reduce overhead is via Unix domain sockets [30], which is lighter weight than TCP sockets while providing the same API. When using a Unix domain socket, the kernel copies the data to kernel space and directly puts the constructed socket buffer on the receiving side's socket queue, avoiding the expensive network stack processing.

Reducing Write Overhead A second significant source of Envoy overhead is the latency and CPU usage of copying data, which is embedded in the Read and Write component.



(a) Unix Domain Socket. (b) Zero Copy Writes.

Figure 13: End-to-end latency reduction in the Alibaba trace with Linux features.



(a) Unix Domain Socket. (b) Zero Copy Writes.

Figure 14: End-to-end CPU usage reduction in the Alibaba trace with Linux features.

It is already noticeable with 100 byte messages (Table 2) and gets worse with larger messages. Linux kernel supports zero-copy TCP sockets starting from version 4.14. For write calls, applications can pin memory buffers in userspace; the kernel signals to the application after sending the buffers to enable garbage collection or re-use of the buffers. This eliminates the need for copying the data from the userspace to the kernel. Linux does not support zero-copy for read calls.

We use MeshInsight to evaluate the performance implications of these optimizations using the same Alibaba workloads as above. We consider TCP mode sidecars. The absolute savings for HTTP or gRPC mode will be similar but the relative advantage in those modes will be small because their performance is bottlenecked by parsing rather than IPC or data copy.

Figure 13 shows the latency overheads when using Unix domain socket and zero-copy write. We observe substantial improvement from the domain socket when message size is 100 Bytes: the average speedup is 0.71 ms. For 4KB messages, the average speedup 0.78 ms. However, because the latency prior to the optimization is higher (4.01 ms versus 2.88 ms), the relative decrease in latency is lower with 4KB. This result

is in line with IPC overheads constituting less to the total latency overheads when the message size is large.

We also see that zero-copy writes bring negligible performance improvement in our setting. This optimization has additional performance cost for buffer lifecycle management [7]. For the message size regime of most interest for microservices, the gains of avoiding a copy is negated by this additional cost.

Figure 14 shows the CPU overheads with the two optimizations. When messages are 100 bytes, the average CPU overhead reduces from 86 to 71 virtual cores with Unix domain sockets. The difference is quite substantial. When messages are 4KB, Unix domain sockets reduce the CPU overheads from 107 to 91 virtual cores. As for latency, zero-copy writes do not improve the CPU overheads notably, reducing the CPU overheads from 86 ms to 84 ms for 100-byte messages.

7 DISCUSSION

MeshInsight provides a systematic way to understand the overhead of service meshes and confirms that they can significantly increase latency and CPU usage of distributed, microservices applications. While we focused on the currently-dominant sidecar architecture, our work has implications also for alternative architectures being considered in the industry.

In-Kernel Proxies The Linux kernel has increasing support for extensibility using eBPF. Researchers have also proposed using safe languages, such as Rust, for kernel extension development [60]. One approach to reducing the system call overheads and the data copy overheads (across userspace and kernel) is to implement a sidecar's functionality inside the kernel. Katran [6] offloads layer-4 load balancers into the kernel using BPF. It is now possible for Envoy to run a limited set of filters directly in Cillium [5], a popular framework for using eBPF on the network data path. However, our study shows that while removing the system call and data copy overhead can be useful for TCP proxies, it will offer limited performance improvement for HTTP and gRPC proxies because protocol parsing is the major overhead in those configurations.

Hardware Offloading Another direction being currently explored is to offload the sidecar logic to programmable network hardware [36]. While this is a promising direction, there are substantial challenges. For example, it is still questionable if programmable network hardware can do complex layer-7 protocol processing efficiently. There are a variety of layer-7 protocols (e.g., HTTP, gRPC, MySQL). These protocols are more complex than the fixed functions people typically offload to programmable network hardware, such as firewalls, NATs, and layer-4 load balancers [55, 59, 75]. In addition, many filters in sidecars (e.g., encryption) require reconstructing the original data stream, and this means programmable network hardware also needs to run TCP/IP packet processing (e.g., packet loss recovery, congestion control). Prior works on offloading application logic to programmable network hardware use UDP as the transport to circumvent this issue [66].

New Directions on Reducing Service Mesh Performance Overheads. Our work shed light on some new directions that are worth exploring. We observe that protocol parsing is a major overhead for HTTP and gRPC proxies. Unfortunately, using a sidecar today means that there is duplicated protocol processing. A sidecar parses an HTTP request from TCP streams, optionally modifies it, and serializes it into a TCP stream again. The application service receiving this stream has to parse the HTTP request yet again.

There are two potential methods to eliminate this double parsing. The first is by linking sidecars to libraries that applications use to parse various protocols. Another is to create a new transport protocol for efficient parsing by sidecars, which is possible in this context where both ends of the communication are sidecars. Both of these methods, however, have limitations. The first one does not work with unmodified applications, and the second one does not help when filters need to inspect HTTP headers added by applications. In future work, we will investigate these and other methods.

8 RELATED WORK

Our work is related to several threads of prior work.

Microservices. The microservice architecture has introduced many additional challenges to the research community, and many recent works have been trying to address various aspects of the microservice architecture, including reducing tail latencies [51, 70, 71], fault-tolerance [50], debugging [43], isolation mechanisms [34], energy efficiency [57], monitoring [53], and ensuring correctness [64]. Because the application logic is decomposed into many independent microservices, this increases the amount of communication between various microservices within an application.

Today, operators increasingly use service mesh to deploy microservice applications for enabling better visibility and control over the communication between microservices. Unfortunately, the networking aspect of the microservice architecture and the service mesh has received little attention. We provide the first systematic study of the performance aspects of a service mesh, and we show that the service mesh can lead to major performance overheads.

Performance of the host network stack. Understanding the host networking stack performance is a common goal in many previous works. Peter et al. [65] breaks down the latency overheads of the Linux network stack. Neugebauer et al. [62] studies how PCIe affects network performance in host networking. Farshin et al. [39] examines how Intel Data Direct I/O technology (for NIC to access CPU's last-level cache directly) speeds up host networking performance. More recently, NSight [47] uses Intel Processor Tracing to diagnose latency in network stacks.

While we share data gathering primitives from these works, our focus is on the data path of service meshes (which traverses the network stack multiple times and has a substantial userspace processing component). We decompose the overhead of a sidecar proxy in the datapath of host networking, and we identify the key contributors to high overhead such as IPC and protocol parsing.

Performance of online services and network proxies. Many works have investigated the performance of network proxies and developed improvements such as hardware offloading [59, 67, 75], kernel offloading [12, 68] and re-homing TCP connections [48]. However, these works mostly focus on layer-4 proxies, while sidecars are layer-7 proxies. Our measurements of sidecars provide insights into performance bottlenecks of layer-7 proxies. We plan to combine insights from our study and techniques for improving the performance of layer-4 proxies to develop high-performance layer-7 proxies.

Similar to our approach, Stewart et al. [72] uses linear models to profile the performance characteristics of online services, treating them as black boxes. Conversely, our model breaks down the overhead of sidecars into distinct components, highlighting the main contributors to service mesh overhead. This not only spotlights the primary contributors but also empowers service mesh developers to assess the cumulative impact of their improvements.

Reducing inter-process communication overheads. Reducing IPC overheads is one of the oldest research topics in the operating system community. Immich et al. [49] and Venkataraman et al. [73] study the existing IPC mechanisms' performance on Linux. IPC performance is a critical design aspect for microkernels [38, 56]. The goal of our work is not to develop techniques to lower IPC overhead but to build a tool that helps evaluate the impact of such techniques on the end-to-end performance of service meshes.

9 CONCLUSION

This paper presents a tool, MeshInsight, that systematically quantifies the overhead of service mesh sidecars. Its compositional approach can analyze a wide range of deployment scenarios (i.e., the combination of service mesh configuration and application characteristics), without the need to directly measure them. This ability can help application developers pick the appropriate service mesh configuration for their specific application needs; as we showed using a large dataset of microservice applications, the overhead of service meshes can vary by orders of magnitude based on the configuration, and in a given configuration, the overhead can again vary by orders of magnitude across applications.

MeshInsight can also identify the primary contributors to the overhead in any scenario. We find, for instance, that IPC and socket writes are the main contributors when the service mesh is configured in TCP mode but protocol parsing dominates in other modes. This ability can help service mesh developers as they work to lower the overhead of service meshes.

ACKNOWLEDGEMENT

We thank the anonymous reviewers and our shepherd, Karla Saur, for their comments and feedback. This work was supported by UW FOCI and its partners (Alibaba, Amazon, Cisco, Google, Microsoft, and VMware).

REFERENCES

[1] 2013. Decomposing Twitter: Adventures in Service-Oriented Architecture. https://www.infoq.com/presentations/twitter-soa/.
[2] 2020. CNCF SURVEY 2020. https://www.cncf.io/wp-content/uploads/2020/11/CNCF_Survey_Report_2020.pdf.
[3] 2021. Accelerate Istio-CNI with ebpf. https://events.istio.io/istiocon-2021/sessions/accelerate-istio-cni-with-ebpf/.
[4] 2021. Benchmarking Linkerd and Istio: 2021 Redux. https://linkerd.io/2021/11/29/linkerd-vs-istio-benchmarks-2021/.
[5] 2021. eBPF-based Networking, Observability, and Security. https://cilium.io/.
[6] 2021. Katran: A high performance layer 4 load balancer. https://github.com/facebookincubator/katran.
[7] 2021. MSG ZEROCOPY. https://www.kernel.org/doc/html/latest/networking/msg_zerocopy.html.
[8] 2021. Online Boutique - a cloud-native microservices demo application. https://github.com/GoogleCloudPlatform/microservices-demo.
[9] 2022. Aspen Mesh. https://aspenmesh.io/.
[10] 2022. AWS App Mesh: Application-level networking for all your services. https://aws.amazon.com/app-mesh/.
[11] 2022. Bookinfo Application. https://istio.io/latest/docs/examples/bookinfo/.
[12] 2022. Cilium Service Mesh – Everything You Need to Know. https://isovalent.com/blog/post/cilium-service-mesh/.
[13] 2022. Cloud Native Value Measurement. https://smp-spec.io/meshmark.
[14] 2022. Consul Service Mesh. https://www.consul.io/docs/connect.
[15] 2022. Demonstrations of funclatency, the Linux eBPF/BCC version. https://github.com/iovisor/bcc/blob/master/tools/funclatency_example.txt.
[16] 2022. eBPF and the Service Mesh: Don't Dismiss the Sidecar Yet. https://www.infoq.com/articles/ebpf-service-mesh/.
[17] 2022. eBPF, sidecars, and the future of the service mesh. https://buoyant.io/blog/ebpf-sidecars-and-the-future-of-the-service-mesh.

[18] 2022. Envoy. https://www.envoyproxy.io/.

[19] 2022. Finagle. https://github.com/twitter/finagle.

[20] 2022. How fast is Envoy?

[21] 2022. Hystrix: Latency and Fault Tolerance for Distributed Systems. https://github.com/Netflix/Hystrix.

[22] 2022. Introducing Ambient Mesh. https://isovalent.com/blog/post/cilium-service-mesh/.

[23] 2022. Istio Performance and Scalability. https://istio.io/latest/docs/ops/deployment/performance-and-scalability/.

[24] 2022. The Istio service mesh. https://istio.io/.

[25] 2022. Jaeger: open source, end-to-end distributed tracing. https://www.jaegertracing.io/.

[26] 2022. Open Service Mesh. https://openservicemesh.io/.

[27] 2022. OpenShift Service Mesh. https://cloud.redhat.com/learn/topics/service-mesh.

[28] 2022. Performance Benchmark Analysis of Istio and Linkerd. https://kinvolk.io/blog/2019/05/performance-benchmark-analysis-of-istio-and-linkerd/.

[29] 2022. Service meshes are on the rise — but greater understanding and experience are required. https://www.cncf.io/wp-content/uploads/2022/05/CNCF_Service_Mesh_MicroSurvey_Final.pdf.

[30] 2022. Unix domain socket. https://en.wikipedia.org/wiki/Unix_domain_socket.

[31] 2022. The world's lightest, fastest service mesh. https://linkerd.io/.

[32] 2022. wrk. https://github.com/wg/wrk.

[33] 2022. wrk2. https://github.com/giltene/wrk2.

[34] Sol Boucher, Anuj Kalia, David G Andersen, and Michael Kaminsky. 2018. Putting the" micro" back in microservice. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. 645–650.

[35] Jingrong Chen, Yongji Wu, Shihan Lin, Yechen Xu, Xinhao Kong, Thomas Anderson, Matthew Lentz, Xiaowei Yang, and Danyang Zhuo. 2023. Remote Procedure Call as a Managed System Service. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. 141–159.

[36] Tianyi Cui, Wei Zhang, Kaiyuan Zhang, and Arvind Krishnamurthy. 2021. Offloading load balancers onto SmartNICs. In *Proceedings of the 12th ACM SIGOPS Asia-Pacific Workshop on Systems*. 56–62.

[37] Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, Kirk Webb, et al. 2019. The Design and Operation of CloudLab. In *2019 USENIX annual technical conference (USENIX ATC 19)*. 1–14.

[38] Kevin Elphinstone and Gernot Heiser. 2013. From L3 to SeL4 What Have We Learnt in 20 Years of L4 Microkernels?. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. 133–150.

[39] Alireza Farshin, Amir Roozbeh, Gerald Q Maguire Jr, and Dejan Kostić. 2020. Reexamining Direct Cache Access to Optimize I/O Intensive Applications for Multi-hundred-gigabit Networks. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. 673–689.

[40] Tobias Flach, Nandita Dukkipati, Andreas Terzis, Barath Raghavan, Neal Cardwell, Yuchung Cheng, Ankur Jain, Shuai Hao, Ethan Katz-Bassett, and Ramesh Govindan. 2013. Reducing web latency: the virtue of gentle aggression. In *Proceedings of the ACM SIGCOMM 2013 conference on SIGCOMM*. 159–170.

[41] Joshua Fried, Zhenyuan Ruan, Amy Ousterhout, and Adam Belay. 2020. Caladan: Mitigating interference at microsecond timescales. In *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation*. 281–297.

[42] Yu Gan, Yanqi Zhang, Dailun Cheng, Ankitha Shetty, Priyal Rathi, Nayan Katarki, Ariana Bruno, Justin Hu, Brian Ritchken, Brendon Jackson, et al. 2019. An Open-source Benchmark Suite for Microservices and Their Hardware-software Implications for Cloud & Edge Systems. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. 3–18.

[43] Yu Gan, Yanqi Zhang, Kelvin Hu, Dailun Cheng, Yuan He, Meghna Pancholi, and Christina Delimitrou. 2019. Seer: Leveraging Big Data to Navigate the Complexity of Performance Debugging in Cloud Microservices. In *Proceedings of the twenty-fourth international conference on architectural support for programming languages and operating systems*. 19–33.

[44] A. Gheith, R. Rajamony, P. Bohrer, K. Agarwal, M. Kistler, B. L. White Eagle, C. A. Hambridge, J. B. Carter, and T. Kaplinger. 2016. IBM Bluemix Mobile Cloud Services. *IBM J. Res. Dev.* 60, 2–3 (March 2016), 7:1–7:12. https://doi.org/10.1147/JRD.2016.2515422

[45] Google. 2022. Anthos Service Mesh. https://cloud.google.com/anthos/service-mesh.

[46] Brendan Gregg. 2022. Linux perf Examples. https://www.brendangregg.com/perf.html.

[47] Roni Haecki, Radhika Niranjan Mysore, Lalith Suresh, Gerd Zellweger, Bo Gan, Timothy Merrifield, Sujata Banerjee, and Timothy Roscoe. 2022. How to diagnose nanosecond network latencies in rich end-host stacks. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*. 861–877.

[48] Yutaro Hayakawa, Michio Honda, Douglas Santry, and Lars Eggert. 2021. Prism: Proxies without the pain. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*. 535–549.

[49] Patricia K Immich, Ravi S Bhagavatula, and Ravi Pendse. 2003. Performance analysis of five interprocess communication mechanisms across UNIX operating systems. *Journal of Systems and Software* 68, 1 (2003), 27–43.

[50] Gopal Kakivaya, Lu Xun, Richard Hasha, Shegufta Bakht Ahsan, Todd Pfleiger, Rishi Sinha, Anurag Gupta, Mihail Tarta, Mark Fussell, Vipul Modi, et al. 2018. Service fabric: a distributed platform for building microservices in the cloud. In *Proceedings of the thirteenth EuroSys conference*. 1–15.

[51] Ram Srivatsa Kannan, Lavanya Subramanian, Ashwin Raju, Jeongseob Ahn, Jason Mars, and Lingjia Tang. 2019. GrandSLAm: Guaranteeing SLAs for Jobs in Microservices Execution Frameworks. In *Proceedings of the Fourteenth EuroSys Conference 2019*. 1–16.

[52] Ron Kohavi and Roger Longbotham. 2007. Online experiments: Lessons learned. *Computer* 40, 9 (2007), 103–105.

[53] Joshua Levin and Theophilus A Benson. 2020. ViperProbe: Rethinking Microservice Observability with eBPF. In *2020 IEEE 9th International Conference on Cloud Networking (CloudNet)*. IEEE, 1–8.

[54] Bojie Li, Tianyi Cui, Zibo Wang, Wei Bai, and Lintao Zhang. 2019. Socksdirect: Datacenter Sockets Can Be Fast and Compatible. In *Proceedings of the ACM Special Interest Group on Data Communication* (Beijing, China) *(SIGCOMM '19)*. 90–103.

[55] Bojie Li, Kun Tan, Layong (Larry) Luo, Yanqing Peng, Renqian Luo, Ningyi Xu, Yongqiang Xiong, Peng Cheng, and Enhong Chen. 2016. ClickNP: Highly Flexible and High Performance Network Processing with Reconfigurable Hardware. In *Proceedings of the 2016 ACM SIGCOMM Conference* (Florianopolis, Brazil) *(SIGCOMM '16)*. Association for Computing Machinery, New York, NY, USA, 1–14. https://doi.org/10.1145/2934872.2934897

[56] Jochen Liedtke. 1993. Improving IPC by kernel design. In *Proceedings of the fourteenth ACM symposium on Operating systems principles*. 175–188.

[57] Ming Liu, Simon Peter, Arvind Krishnamurthy, and Phitchaya Mangpo Phothilimthana. 2019. E3: Energy-efficient Microservices on SmartNIC-accelerated Servers. In *2019 {USENIX} Annual Technical Conference ({USENIX}{ATC} 19)*. 363–378.

[58] Shutian Luo, Huanle Xu, Chengzhi Lu, Kejiang Ye, Guoyao Xu, Liping Zhang, Yu Ding, Jian He, and Chengzhong Xu. 2021. Characterizing Microservice Dependency and Performance: Alibaba Trace Analysis. In *Proceedings of the ACM Symposium on Cloud Computing*. 412–426.

[59] Rui Miao, Hongyi Zeng, Changhoon Kim, Jeongkeun Lee, and Minlan Yu. 2017. SilkRoad: Making Stateful Layer-4 Load Balancing Fast and Cheap Using Switching ASICs. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication* (Los Angeles, CA, USA) *(SIGCOMM '17)*. Association for Computing Machinery, New York, NY, USA, 15–28. https://doi.org/10.1145/3098822.3098824

[60] Samantha Miller, Kaiyuan Zhang, Danyang Zhuo, Shibin Xu, Arvind Krishnamurthy, and Thomas Anderson. 2019. Practical Safe Linux Kernel Extensibility. In *HotOS*.

[61] Behnam Montazeri, Yilong Li, Mohammad Alizadeh, and John Ousterhout. 2018. Homa: A receiver-driven low-latency transport protocol using network priorities. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*. 221–235.

[62] Rolf Neugebauer, Gianni Antichi, José Fernando Zazo, Yury Audzevich, Sergio López-Buedo, and Andrew W Moore. 2018. Understanding PCIe performance for end host networking. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*. 327–341.

[63] Minh Nguyen, Zhongwei Li, Feng Duan, Hao Che, and Hong Jiang. 2016. The tail at scale: how to predict it?. In *8th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 16)*.

[64] Aurojit Panda, Mooly Sagiv, and Scott Shenker. 2017. Verification in the Age of Microservices. In *Proceedings of the 16th Workshop on Hot Topics in Operating Systems*. 30–36.

[65] Simon Peter, Jialin Li, Irene Zhang, Dan R. K. Ports, Doug Woos, Arvind Krishnamurthy, Thomas Anderson, and Timothy Roscoe. 2014. Arrakis: The Operating System is the Control Plane. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. USENIX Association, Broomfield, CO, 1–16. https://www.usenix.org/conference/osdi14/technical-sessions/presentation/peter

[66] Phitchaya Mangpo Phothilimthana, Ming Liu, Antoine Kaufmann, Simon Peter, Rastislav Bodik, and Thomas Anderson. 2018. Floem: A Programming System for NIC-Accelerated Network Applications. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. USENIX Association, Carlsbad, CA, 663–679. https://www.usenix.org/conference/osdi18/presentation/phothilimthana

[67] Salvatore Pontarelli, Roberto Bifulco, Marco Bonola, Carmelo Cascone, Marco Spaziani, Valerio Bruschi, Davide Sanvito, Giuseppe Siracusano, Antonio Capone, Michio Honda, Felipe Huici, and Giuseppe Siracusano. 2019. FlowBlaze: Stateful Packet Processing in Hardware. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. USENIX Association, Boston, MA, 531–548. https://www.usenix.org/conference/nsdi19/presentation/pontarelli

[68] Shixiong Qi, Leslie Monis, Ziteng Zeng, Ian-chin Wang, and KK Ramakrishnan. 2022. SPRIGHT: extracting the server from serverless computing! high-performance eBPF-based event-driven, shared-memory processing. In *Proceedings of the ACM SIGCOMM 2022 Conference*. 780–794.

[69] Feng Qian, Alexandre Gerber, Zhuoqing Morley Mao, Subhabrata Sen, Oliver Spatscheck, and Walter Willinger. 2009. TCP revisited: a fresh look at TCP in the wild. In *Proceedings of the 9th ACM SIGCOMM conference on Internet measurement*. 76–89.

[70] Haoran Qiu, Subho S Banerjee, Saurabh Jha, Zbigniew T Kalbarczyk, and Ravishankar K Iyer. 2020. {FIRM}: An Intelligent Fine-grained Resource Management Framework for SLO-Oriented Microservices. In *14th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 20)*. 805–825.

[71] Akshitha Sriraman and Thomas F Wenisch. 2018. μTune: Auto-Tuned Threading for {OLDI} Microservices. In *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*. 177–194.

[72] Christopher Stewart and Kai Shen. 2005. Performance modeling and system management for multi-component online services. In *Proceedings of the 2nd Conference on Symposium on Networked Systems Design & Implementation-Volume 2*. 71–84.

[73] Aditya Venkataraman and Kishore Kumar Jagadeesha. 2015. Evaluation of Inter-Process Communication Mechanisms. *Architecture* 86 (2015), 64.

[74] Juncheng Yang, Yao Yue, and KV Rashmi. 2020. A large scale analysis of hundreds of in-memory cache clusters at Twitter. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. 191–208.

[75] Kaiyuan Zhang, Danyang Zhuo, and Arvind Krishnamurthy. 2020. Gallium: Automated software middlebox offloading to programmable switches. In *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication*. 283–295.

[76] Zhizhou Zhang, Murali Krishna Ramanathan, Prithvi Raj, Abhishek Parwal, Timothy Sherwood, and Milind Chabbi. 2022. {CRISP}: Critical Path Analysis of {Large-Scale} Microservice Architectures. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*. 655–672.