Dissecting Service Mesh Overheads

Xiangfeng Zhu[1] Guozhen She[2] Bowen Xue[1]
Yu Zhang[3] Yongsu Zhang[3] Xuan Kelvin Zou[3] Xiongchun Duan[3] Peng He[3]
Arvind Krishnamurthy[1] Matthew Lentz[2] Danyang Zhuo[2] Ratul Mahajan[1,4]

[1]*University of Washington* [2]*Duke University* [3]*Bytedance* [4]*Intentionet*

Abstract

Service meshes play a central role in the modern application ecosystem by providing an easy and flexible way to connect different services that form a distributed application. However, because of the way they interpose on application traffic, they can substantially increase application latency and resource consumption. We develop a decompositional approach and a tool, called MeshInsight, to systematically characterize the overhead of service meshes and to help developers quantify overhead in deployment scenarios of interest. Using MeshInsight, we confirm that service meshes can have high overhead—up to 185% higher latency and up to 92% more virtual CPU cores for our benchmark applications—but the severity is intimately tied to how they are configured and the application workload. The primary contributors to overhead vary based on the configuration too. 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.

1 Introduction

Service meshes are fast becoming the *de facto* communication substrate for cloud applications. A survey of the Cloud Native Computing Foundation (CNCF) community found that 68% of the organizations are already using or planning to use service meshes in the next 12 months [2]. In-production use of service meshes has been growing 40-50% annually [2].

Service meshes are popular because they solve important problems related to communication among loosely coupled (micro) services—the dominant paradigm for modern cloud applications [1, 31, 38]. This includes discovering where services are located, establishing secure connections, and handling communication failures. They also offer many advanced capabilities such as rate limiting, load balancing, and telemetry, via built-in or custom message processing filters.

However, service meshes are not without downsides. A primary one is overhead. All 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 (virtual) CPU cores even at moderate load [18]. These overheads can degrade user experience, increase operational costs, and decrease revenue [30, 39].

Today, outside of a few point studies [4, 18], there is little systematic understanding of service mesh overheads. We do not know even basics such as the amount of overhead for real applications and what factors contribute most to the overhead. This is more than a matter of curiosity. Application developers do not know how much overhead the service mesh adds to their application, a problem exacerbated by the large configuration space of service meshes, each with different performance implications. Thus, they cannot evaluate functionality-performance trade-offs and judge the best way to configure the service mesh for their application. Further, there are several efforts in industry on lowering service mesh overhead [3, 14]. Without a proper accounting of the overhead and its major contributors, it is hard for these developers to quantify the effectiveness of their optimizations, especially as it relates to end-to-end impact on applications.

Our goal is to characterize service mesh overhead in a way that helps application and service mesh developers reason about performance. To support these developers, unlike prior studies, we cannot simply measure the overhead of sidecars as a black box. If we did that, we would have to measure the combinatorial combination of all factors that impact overhead including service mesh configuration and application workload. And we still won't be able to judge the end-to-end impact of optimizing a specific aspect of service meshes.

We develop an approach that models the sidecar's operation as a composition of several 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, we can estimate the end-to-end overhead for the application (without the developer having to deploy the application under that configuration). Similarly, if an optimization reduces the overhead

of some components, we can estimate its end-to-end impact.

We build a tool called MeshInsight that uses the approach above. We use it to quantify the latency and CPU overhead of Envoy [15]. It is the dominant sidecar, used by many service meshes [9, 10, 12, 20, 21, 33], including Istio [19], the most popular service mesh today [2].

Using MeshInsight, we conduct a systematic study of service mesh overheads. We confirm that service meshes can have substantial performance penalty. Across two popular benchmark applications [8, 31], depending on the configuration, request latency increases by 30-185% and CPU usage increases by 41-92%. In a large dataset of microservice-based applications [43], we find that using Envoy increases latency by up to 100 ms and consumes 200 more virtual CPU cores for a quarter of the applications. We also find that, for a given service mesh configuration, the overhead for different applications varies by multiple orders of magnitudes. Such high variation based on service mesh configuration and on application characteristics validate the need for a tool that developers can use for their specific deployment scenarios.

The compositional approach of MeshInsight provides insight into the sources of overheads. We find that when configured in HTTP or gRPC mode, protocol parsing alone represents 62-73% of the total overhead. In TCP mode, most overhead stems from inter-process communication (IPC) and socket write operations. The overhead of individual filters varies a lot. Some add as little as 3% extra latency on top of the baseline overhead, while others add 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 on our microservices dataset [43]. 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 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 three key contributions in our work:

- A decompositional approach to model and predict service mesh overheads in specific deployment scenarios. It is based on our analysis of key components in service mesh datapaths.

- A tool that application developers can use it to quantify overhead and make judicious performance-functionality trade-offs. Service mesh developers can also use the tool to quantify the end-to-end impact of their optimizations.

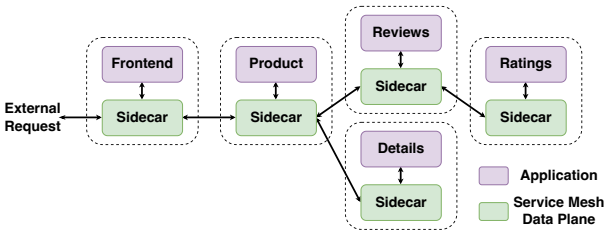


Figure 1: Bookinfo application with a service mesh. Image Redrawn from [11].

- A systematic study of service mesh overheads. It confirms that the overhead can be significant in some settings and reveals which components contribute the most overhead in different settings.

Our tool and findings 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.

2 Background

Service meshes emerged to solve a set of problems that arose when applications moved to the microservices architecture. Instead of being a monolithic unit, applications are composed of multiple microservices. Figure 1 shows an example where the BookInfo application is composed of five individual services. 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 [32]. Modern applications commonly use the microservices architecture, with applications having tens of services [1, 32, 38].

While there are important benefits, splitting application into multiple microservices creates new problems as well. Developers must now figure out how services discover, communicate, and authenticate to 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 [16, 17]. Service meshes solve them in a reusable manner while also providing other functionality such as rate limiting and load balancing.

The operation of service meshes is logically split into a control plane and a data plane. The control plane handles service discovery, metric collection, and certificate management, and it appropriately configures the data plane. The data plane uses software proxies called *sidecars*. A sidecar instance is co-located with each instance of an application service, which enables it to mediate all of service's network access to apply network policies, enforce encryption and log statistics. For example, in Figure 1, to balance load across multiple instances of the Product service, the Frontend sidecar

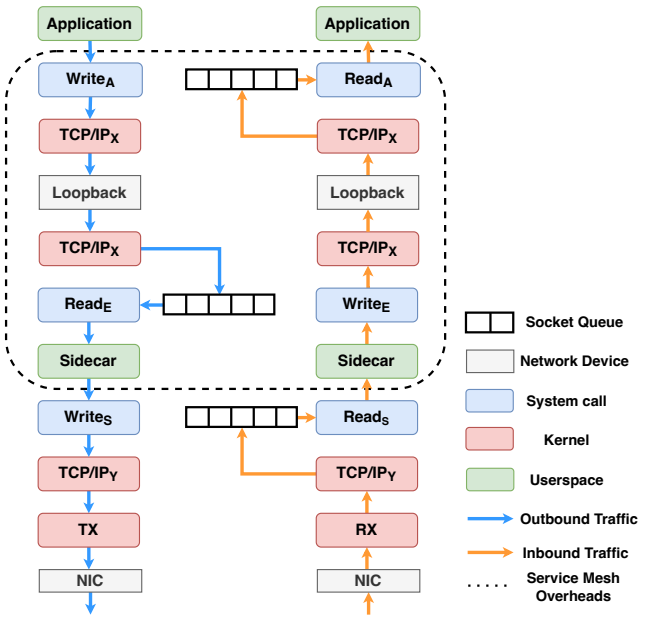


Figure 2: Outbound (left) and Inbound (right) messages with a service mesh. Write_A/Read_A and Write_S/Read_S denote, respectively, the write/read of application and its sidecar. TCP/IP_X denotes the TCP/IP stack of network namespace X. The figure assumes that the application uses the same event notification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

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 sidecar accordingly.

Service Mesh Data Path The advantages of service meshes come at the cost of performance and resource overhead. We focus on the data plane overhead as it impacts every request and is on the critical path of user experience. We use Envoy [15] as our example. It is used by many service meshes [9, 10, 12, 20, 21, 33], including Istio, the most popular service mesh. Other data plane proxies [23] have a similar architecture.

Figure 2 shows data path for both outbound and inbound traffic. 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, 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

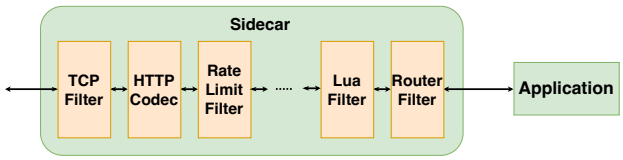


Figure 3: Message processing inside a sidecar.

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 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 performance overheads of service meshes for real-world microservice applications. We want to support two classes of developers. First, application developers who are looking to deploy service meshes should be able to understand overhead as a function of service mesh configuration (e.g., proxy types, filters), so they can appropriately trade off functionality and overhead. Second, service mesh developers should be able to understand the impact of their optimizations on real-world applications. Today, there are several ongoing directions in industry and academia to reduce the service mesh performance overheads [5, 26]. However, these service mesh developers do not have a way to get these insights (without doing their own extensive benchmarking experiments).

	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	Protocol Parsing	Protocol parsing in sidecar
6	Other Userspace	Other userspace processing in sidecar
7	Filter	Filter chain processing in sidecar

Table 1: Components in MeshInsight's performance model.

A key challenge we face toward meeting our goals 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, gRPC, and MySQL); *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 be able to 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 a 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 his/her workloads. However, this modeling-based approach requires us to 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.

Table 1 shows the components we profile. 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 last three breakdown message processing inside the sidecar: 5) parsing the messaging protocol (e.g., HTTP), 6) additional userspace processing in the sidecar (e.g., default Router filter), and 7) processing done by user-configured filters.

We do not claim that this is the only way to decompose the sidecar's overhead. We chose these components to balance the granularity concern above and because they are mostly independent. This independence allows us to easily compose their overheads to estimate total sidecar overhead. It also means that service mesh optimizations typically impact one or two of these components, which allows us to understand and predict the performance impact through targeted modifications to the profiles of optimized components. We have also chosen to ignore certain sources of overheads such as cache contention and more frequent context switching. Our experiments confirm that the impact of such factors is minimal.

MeshInsight overview 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 individual service mesh components, and the online phase predicts overhead based on these profiles, service mesh configuration, and application workload.

The performance of a component is modeled 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. Overhead predictions are made only for previously profiled platforms. We make no attempt to predict performance for unprofiled platforms. [1]

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 annotated call graph (described in the next section), provided by the user. The annotated call graph encodes the infrastructure and service mesh configuration of each microservice and captures the relevant properties of a microservice application that is handling a particular request.

Overhead measures We consider two primary measures of interest to developers: latency and CPU usage overhead. In particular, we quantify extra latency service mesh adds to the application messages, and extra CPU (number of virtual cores) that message processing consumes. For latency, we consider the average overhead in a lightly or moderately loaded system. Latency-sensitive applications are more likely to be run in this regime. In heavily loaded systems, unpredictable effects

[1] As part of this work, we are developing a shared repository of performance profiles for common platforms such as AWS instances, to enable reuse of profiling data.

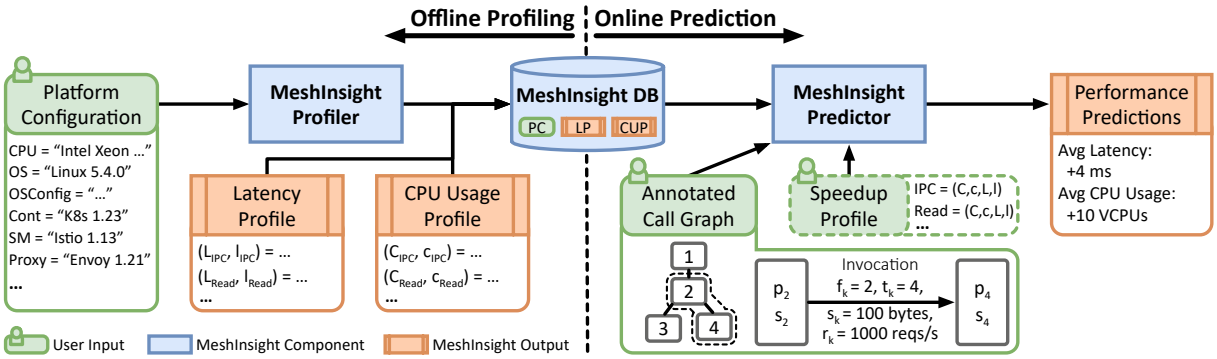


Figure 4: Overview of MeshInsight workflow. The MeshInsight DB stores performance profiles associated with a hardware and software platform configuration, which are generated by the MeshInsight Profiler during the offline profiling stage §4.1. In the online phase §4.2, these profiles are used by the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

due to resource contention kick in, which we do not model. We leave the study of overhead measures such as memory and tail latency to future work.

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. To build these profiles, we exercise these components under a few different settings and then interpolate and extrapolate their performance to settings that are not directly measured.

We conduct the following 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 runs use an echo server paired with a sidecar. We use wrk [24] and wrk2 [25] both as load generators as tools to measure the end-to-end latency with high precision. To mimic a lightly loaded server, 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) corresponds to a specific kernel- or user-space function. We measure the latency of each component using a modified version of BCC's funclatency [13], 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 [34], which allows us to quantify the CPU consumption of any function. We identify the function for a component using a mix of function name, function's 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 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 limit traffic rate on the entire service mesh (global rate limiting) or on per service instance basis (local rate limiting). Likewise, the overhead of Tap filter, which logs traffic, depends on whether the log is written to a file or sent over 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 (see below) of message size.
- **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 [40, 54], we use this linear function:

$$L_x + s \times l_x \quad (1)$$

where L_x is the baseline message processing latency and l_x is the per-byte processing latency. We assume that components' per-message latency does not vary based on request rate.

To estimate CPU consumption of a component x for request rate r, we use the following linear function:

$$r \times (C_x + s \times c_x) \quad (2)$$

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 large message sizes when they 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 our modeling is that it will underestimate the overhead for messages that are split; the actual latency and CPU cost is higher for such messages. We quantify this underestimation in §5.5. Fortunately, the vast majority of messages sizes are small [48, 53, 56], and the impact of our modeling approximation is therefore minimal.

To estimate (L_x, l_x) and (C_x, c_x), MeshInsight profiles component for five different message sizes (i.e., 100B, 1KB, 2KB, 3KB, 4KB) and uses linear regression on the resulting data. These two tuples represent the latency and CPU profile of a component for a particular platform.

4.2 Predicting Overhead

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

Formally, an ACG is a tuple (V, P, S, G), where $V = \{v_1, v_2, ..., v_n\}$ is a set of vertices representing microservice instances; 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 edge represents invoked-after relationship. An invocation k is a tuple (f_k, t_k, s_k, r_k), where f_k is the calling microservice (empty if called externally), t_k is the microservice invoked, and s_k, r_k are the expected size (in bytes) and rate of messages (in requests/second) along this invocation.

Figure 5 show 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.

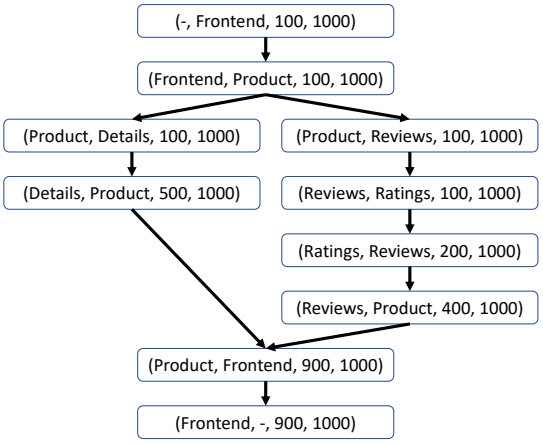


Figure 5: An example call graph for the Bookinfo application.

MeshInsight estimates the overhead of a given ACG. We expect developers have the information necessary to furnish the ACG. If an application has multiple request types (as is the case for our example applications in §5.1), developers can provide multiple ACGs and learn the overhead of each. In cases where an invocation is non-deterministic (e.g., based on cache hits), developers can provide two ACGs, one with the invocation and one without. They will then learn the overhead of each ACG and combine them based on expected probabilities to get the average overhead. Similarly, if a service load balances to multiple instances of a downstream service, the developers can provide multiple ACGs, one for each possible path through the services. In the Bookinfo example, if Frontend balances load across two Product instances, we will have two ACGs. The message rate from Frontend to each Product instance, and downstream of that, will be 500 requests per second.

Generating Predictions. Given an ACG, MeshInsight estimates the latency and CPU overhead. It starts by computing the overhead of each invocation. For an invocation (f_k, t_k, s_k, r_k), the overhead is based on messages of the given size and rate leaving the service at f_k and entering the service at t_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 it is straightforward to extend our model to account for them.

MeshInsight computes end-to-end request-level overheads using sidecar-level overheads as computed above. For CPU, this is simply summing all the sidecar-level overheads. For

latency, simple summation does not work because computations can happen in parallel, and thus critical path analysis is needed. 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. We thus compute the latency overhead using critical path analysis, essentially reporting the highest latency path in G. In this analysis, we are assuming that the critical path computed based on latency overheads is the same as the one with application processing; this path will often be the longest path in the invocation chain. If information on application processing latency were available, MeshInsight could compute a better estimate of end-to-end latency overhead in cases where this assumption does not hold. In the future, we will allow developers to optionally provide this information.

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 MeshInsight profiling after implementing the optimization.

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

5 Characterizing Service Mesh Overhead

We now use MeshInsight to characterize the overhead of service meshes in realistic deployments scenarios and shed light on major sources of overhead in different scenarios. Our experiments use Cloudlab [27] 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 TurboBoost, CPU C-states and dynamic CPU frequency scaling to reduce measurement variance.

5.1 Application Benchmarks

To characterize the overhead of service meshes on realistic applications, we consider two popular microservices benchmarks: Online Boutique [8] and Hotel Reservation [31]. 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

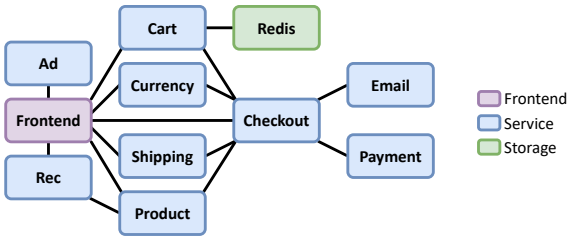


Figure 6: Online Boutique application [8].

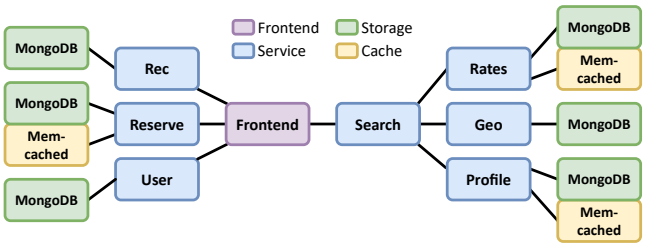


Figure 7: Hotel Reservation application [31].

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.

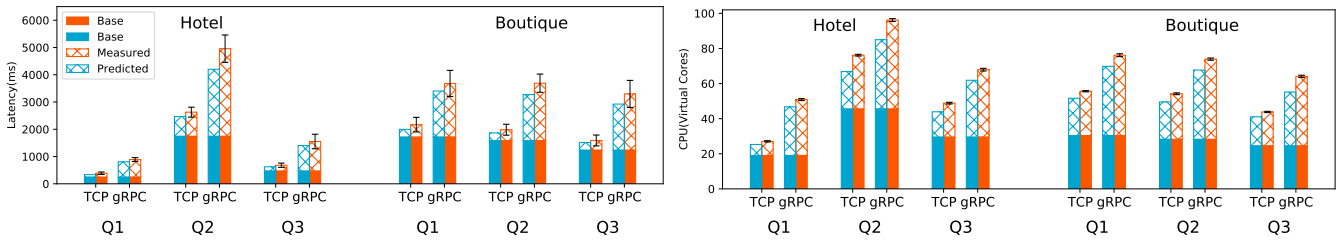
We deploy these applications on multiple Cloudlab hosts and consider two deployment scenarios: TCP and gRPC. (HTTP cannot be used because the applications are gRPC-based.) In TCP mode, all sidecars are configured as a TCP proxy that relays the message to the application service. In gRPC mode, the sidecars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

We consider three queries for each benchmark. For hotel reservation, 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
3. **(Q3) Reservation**: Reserves a hotel room. Calls Reserve, User and their MongoDB and Memcached storage. 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.



(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.

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

5.2 End-to-end Overhead

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.

We see that service meshes can be a significant source of overheads. When operating in gRPC mode, it can increase latency by up to 185% and consume 92% more CPU. In TCP mode, the overhead is lower but still substantial–latency increases by 22% and CPU usage by 43%. 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 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. The predictions are generally on the lower end of measured values. This underestimation stems from random noise in complex, running systems and MeshInsight ignoring components with minimal overheads (e.g., kernel scheduling).

5.3 Sources of 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) to which a sidecar can be attached in TCP, HTTP, or gRPC modes. It uses 100 byte messages at 30K requests per second.

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 62-73% of the total overhead.

Parsing overhead is unfortunate because the application code will spend resources on parsing as well. Because of the way in service mesh data path is organized today, there is no opportunity to reuse parsing work across Envoy and 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 (34% 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 most pronounced for "Protocol Other", where its contribution to CPU overhead is far greater than its contribution to latency, but hold for other components as well (e.g., Write). It implies that some optimizations may impact one type of overhead and not the other, and developers need to be careful that optimizing for latency does not hurt CPU and vice versa.

5.4 Impact of Filters

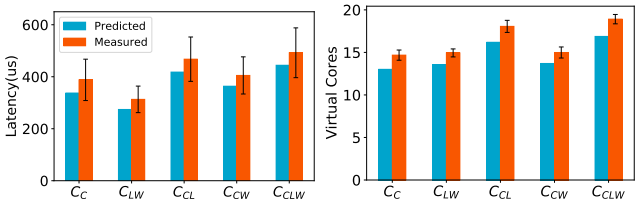
We now characterize the impact of filters on overhead. We find that filters can be quite expensive (even configured as a

	Latency (us)			CPU Usage (Virtual Cores)		
	TCP	**HTTP**	**gRPC**	**TCP**	**HTTP**	**gRPC**
IPC	11.59 (30%)	12.75 (8%)	13.04 (7%)	0.49 (15%)	0.51 (5%)	0.55 (4%)
Read	8.14 (16%)	9.01 (5%)	9.37 (5%)	0.26 (8%)	0.29 (3%)	0.30 (2%)
Write	13.22 (34%)	13.80 (8%)	14.35 (7%)	0.45 (14%)	0.48 (5%)	0.57 (4%)
Notification	1.33 (3%)	1.27 (1%)	1.35 (1%)	0.26 (8%)	0.27 (3%)	0.26 (2%)
Protocol Parsing	-	117.35 (70%)	142.38 (73%)	-	6.00 (62%)	9.76 (71%)
Protocol Other	4.25 (11%)	13.07 (8%)	14.39 (7%)	1.79 (55%)	2.09 (22%)	2.34 (17%)
Total	38.63	167.25	194.79	3.25	9.65	13.79

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 (3.1%)	0.20 (1.9%)
Rate Limit	8.19 (4.5%)	0.21 (2.0%)
Tap	156.09 (85.0%)	2.95 (8.0%)
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.

no-op) and, as assumed by MeshInsight, validate that 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 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 request 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 high overhead of Tap (file) is high because of its interaction with the file system. On the other head, even no-op Lua or WebAssembly filters have substantial latency and CPU overheads, with Lua being 3x more expensive for latency and nearly 5x more expensive for CPU.

To study the composability of filters, 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.

5.5 Impact of Message Size and Rate

We now characterize the impact of message size and rate. We will show that, consistent with our modeling assumptions, the overhead increases with each of these factors. To study the impact of message size, we vary it from 100 bytes to 16KB. The upper end of this range is well beyond the maximum size that we directly profile (4KB). To study the impact of message rate, we vary it from 10K to 50K requests per second.

Latency 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

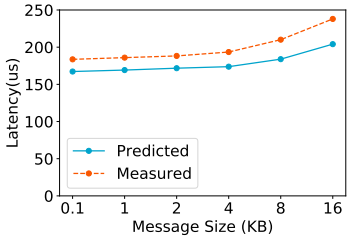


Figure 10: Impact of message size on service mesh latency overhead. X-axis is on log scale.

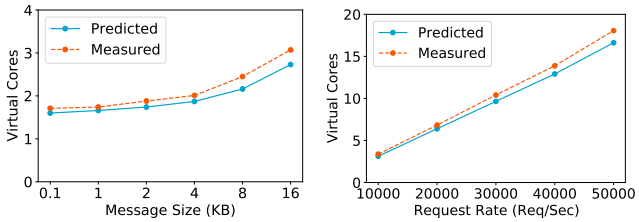


Figure 11: Impact of message size and rate on service mesh CPU overhead of service meshes.

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 drops for very large messages (which is uncommon [48, 53]). 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 has higher latency and CPU cost.

CPU usage. 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.

Similar to latency, large messages have relatively low impact on CPU overhead, compared to message rate. CPU usage increases by 44% when going from 100 bytes to 16 KB.

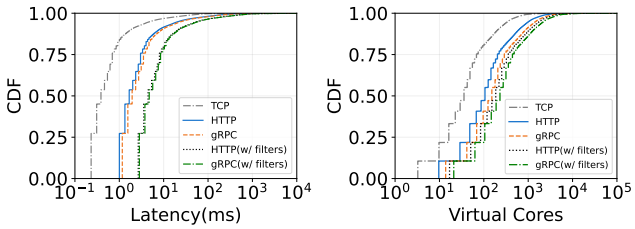
6 Helping Developers Predict Overhead

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 under different configurations. With this knowledge, they can make the right trade-off between desired functionality and overhead.

We demonstrate this use case using the Alibaba microservice traces [43]. These contain over 20M call graphs from



(a) Absolute latency overhead. (b) Absolute CPU overhead.

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

microservices-based applications, collected over 7 days in an Alibaba cluster. While most call graphs have a small number of microservices, 10% of the 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 moderate load.

We consider five possible service mesh configurations, three protocol modes without filters, HTTP with filters, and gRPC with filters (all five filters described in §5.4). Figure 12 shows the latency and CPU overheads. 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. In any given configuration, the overhead varies by multiple orders of magnitude across applications. Even for simplest configuration (TCP), latency overhead varies from 0.2 to 100 ms and CPU overhead varies from 3 to 1000 virtual cores.

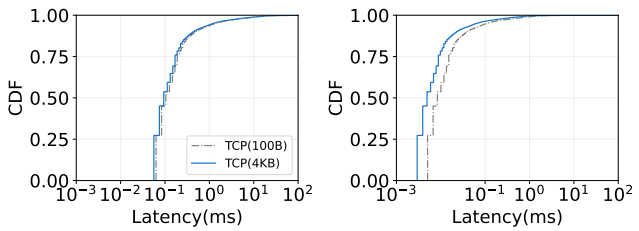
We also see that the overhead of different service mesh configurations varies by an order of magnitude for both latency and CPU. The median latency overhead is 0.2ms in TCP mode but it is 2 ms in gRPC mode with filters, and the 75th percentile varies from 1 to 10 ms. Similarly, the median CPU overhead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

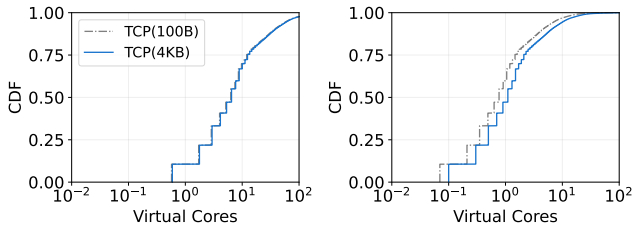
MeshInsight enables service mesh developers to judge the impact of potential optimizations. There are several ongoing works in the industry today to optimize the service mesh performance overheads [3, 14]. While such optimizations can be benchmarked in isolation, it is difficult to understand the end-to-end impact on real-world applications.

To demonstrate this use case, we consider using two Linux



(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.

kernel features in Envoy. Porting Envoy to use new kernel features is quite a bit of implementation effort (and may also introduce some functional limitations), so the Envoy developers may want to estimate the impact even before taking on this work. For this estimation, 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 (Figure 2). 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 §5.3, 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 [22], 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. 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 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 microservice 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 Unix 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 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 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 overheads 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 message .

7 Discussion and Future Work

Our work 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. There are several ongoing works in the industry targeting at reducing service mesh performance overheads, and our work brings new insights on them.

Will In-Kernel Sidecars Reduce Service Mesh Overheads? The Linux kernel has increasing support for extensibility using eBPF. Researchers have also proposed using safe languages, such as Rust, for kernel extension development [46, 47]. One approach to reduce 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 overheads in those configurations.

Will Hardware Offloading Reduce Service Mesh Overheads? Another direction researchers are currently exploring is to offload the sidecar logic to a programmable network hardware [26]. 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, NAT, and layer-4 load balancers [41, 45, 57]. 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 logics to programmable network hardware use UDP as the transport to circumvent this issue [51].

New Directions on Reducing Service Mesh Performance Overheads. Our work shed the light on some new directions that worth exploration. 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 an 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.

Performance of the host network stack. Understanding the host networking stack performance is a common goal in many previous works. Peter et al. [50] breaks down the latency overheads of Linux network stack. Neugebauer et al. [49] studies how PCIe affects network performance in host networking. Farshin et al. [29] 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 [35] 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 network proxies. Many works have investigated the performance of network proxies and developed improvements such as hardware offloading [45, 52, 57] and rehoming TCP connections [36]. However, these works mostly focus on layer-4 network proxies, while sidecars are layer-7 proxies. Our measurements of sidecars provide insights into performance bottlenecks of layer-7 proxies. In the future, 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.

Maltzahn et al. [44] study the performance characteristics of Web proxies. The performance of Web proxies mostly depends on how proxies cache web contents, where sidecars' performance depends on IPC and buffer parsing.

Reducing inter-process communication overheads. Reducing IPC overheads is one of the oldest research topics in the operating system community. Immich et al. [37] and Venkataraman et al. [55] study the existing IPC mechanisms' performance on Linux. IPC performance is a critical design aspect for microkernels [28, 42]. 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

MeshInsight is a tool to systematically quantify the overhead of service meshes. Its decompositional 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 (which would be intractable). 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. Our tool and findings can thus also help service mesh developers as they work to lower the overhead of service meshes, which are now a central component of the modern application ecosystem.

References

[1] Decomposing twitter: Adventures in service-oriented architecture. https://www.infoq.com/presentations/twitter-soa/, 2013.

[2] CNCF survey 2020. https://www.cncf.io/wp-content/uploads/2020/11/CNCF_Survey_Report_2020.pdf, 2020.

[3] Accelerate istio-cni with ebpf. https://events.istio.io/istiocon-2021/sessions/accelerate-istio-cni-with-ebpf/, 2021.

[4] Benchmarking Linkerd and Istio: 2021 redux. https://linkerd.io/2021/11/29/linkerd-vs-istio-benchmarks-2021/, 2021.

[5] eBPF-based networking, observability, and security. https://cilium.io/, 2021.

[6] Katran: A high performance layer 4 load balancer. https://github.com/facebookincubator/katran, 2021.

[7] MSG ZEROCOPY. https://www.kernel.org/doc/html/latest/networking/msg_zerocopy.html, 2021.

[8] Online boutique - a cloud-native microservices demo application. https://github.com/GoogleCloudPlatform/microservices-demo, 2021.

[9] Aspen mesh. https://aspenmesh.io/, 2022.

[10] AWS App Mesh: Application-level networking for all your services. https://aws.amazon.com/app-mesh/, 2022.

[11] Bookinfo application. https://istio.io/latest/docs/examples/bookinfo/, 2022.

[12] Consul service mesh. https://www.consul.io/docs/connect, 2022.

[13] Demonstrations of funclatency, the linux eBPF/BCC version. https://github.com/iovisor/bcc/blob/master/tools/funclatency_example.txt, 2022.

[14] ebpf-powered cilium service mesh. https://cilium.io/blog/2021/12/01/cilium-service-mesh-beta, 2022.

[15] Envoy. https://www.envoyproxy.io/, 2022.

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

[17] Hystrix: Latency and fault tolerance for distributed systems. https://github.com/Netflix/Hystrix, 2022.

[18] Istio performance and scalability. https://istio.io/latest/docs/ops/deployment/performance-and-scalability/, 2022.

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

[20] Open service mesh. https://openservicemesh.io/, 2022.

[21] OpenShift service mesh. https://cloud.redhat.com/learn/topics/service-mesh, 2022.

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

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

[24] wrk. https://github.com/wg/wrk, 2022.

[25] wrk2. https://github.com/giltene/wrk2, 2022.

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

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

[28] Kevin Elphinstone and Gernot Heiser. 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*, page 133–150, 2013.

[29] Alireza Farshin, Amir Roozbeh, Gerald Q Maguire Jr, and Dejan Kostić. Reexamining direct cache access to optimize I/O intensive applications for multi-hundred-gigabit networks. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 673–689, 2020.

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

[31] Yu Gan, Yanqi Zhang, Dailun Cheng, Ankitha Shetty, Priyal Rathi, Nayan Katarki, Ariana Bruno, Justin Hu, Brian Ritchken, Brendon Jackson, et al. 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*, pages 3–18, 2019.

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

[33] Google. Anthos service mesh. `https://cloud.google.com/anthos/service-mesh`, 2022.

[34] Brendan Gregg. Linux perf examples. `https://www.brendangregg.com/perf.html`, 2022.

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

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

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

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

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

[40] Bojie Li, Tianyi Cui, Zibo Wang, Wei Bai, and Lintao Zhang. Socksdirect: Datacenter sockets can be fast and compatible. In *Proceedings of the ACM Special Interest Group on Data Communication*, SIGCOMM '19, page 90–103, 2019.

[41] Bojie Li, Kun Tan, Layong (Larry) Luo, Yanqing Peng, Renqian Luo, Ningyi Xu, Yongqiang Xiong, Peng Cheng, and Enhong Chen. Clicknp: Highly flexible and high performance network processing with reconfigurable hardware. In *Proceedings of the 2016 ACM SIGCOMM Conference*, SIGCOMM '16, page 1–14, New York, NY, USA, 2016. Association for Computing Machinery.

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

[43] Shutian Luo, Huanle Xu, Chengzhi Lu, Kejiang Ye, Guoyao Xu, Liping Zhang, Yu Ding, Jian He, and Chengzhong Xu. Characterizing microservice dependency and performance: Alibaba trace analysis. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 412–426, 2021.

[44] Carlos Maltzahn, Kathy J Richardson, and Dirk Grunwald. Performance Issues of Enterprise Level Web Proxies. *ACM SIGMETRICS Performance Evaluation Review*, 25(1):13–23, 1997.

[45] Rui Miao, Hongyi Zeng, Changhoon Kim, Jeongkeun Lee, and Minlan Yu. 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*, SIGCOMM '17, page 15–28, New York, NY, USA, 2017. Association for Computing Machinery.

[46] Samantha Miller, Kaiyuan Zhang, Mengqi Chen, Ryan Jennings, Ang Chen, Danyang Zhuo, and Thomas Anderson. High velocity kernel file systems with bento. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*, pages 65–79. USENIX Association, February 2021.

[47] Samantha Miller, Kaiyuan Zhang, Danyang Zhuo, Shibin Xu, Arvind Krishnamurthy, and Thomas Anderson. Practical safe linux kernel extensibility. In *HotOS*, 2019.

[48] Behnam Montazeri, Yilong Li, Mohammad Alizadeh, and John Ousterhout. 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*, pages 221–235, 2018.

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

[50] Simon Peter, Jialin Li, Irene Zhang, Dan R. K. Ports, Doug Woos, Arvind Krishnamurthy, Thomas Anderson, and Timothy Roscoe. Arrakis: The Operating System is the Control Plane. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 1–16, Broomfield, CO, October 2014. USENIX Association.

[51] Phitchaya Mangpo Phothilimthana, Ming Liu, Antoine Kaufmann, Simon Peter, Rastislav Bodik, and Thomas Anderson. Floem: A programming system for NIC-Accelerated network applications. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 663–679, Carlsbad, CA, October 2018. USENIX Association.

[52] 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. FlowBlaze: Stateful Packet Processing in Hardware. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 531–548, Boston, MA, February 2019. USENIX Association.

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

[54] Christopher Stewart and Kai Shen. 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*, pages 71–84, 2005.

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

[56] Juncheng Yang, Yao Yue, and KV Rashmi. 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)*, pages 191–208, 2020.

[57] Kaiyuan Zhang, Danyang Zhuo, and Arvind Krishnamurthy. 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*, pages 283–295, 2020.