Towards a Lightweight Sidecar-based Service Mesh for Serverless

Lazar Cvetković
ETH Zurich
lazar.cvetkovic@inf.ethz.ch

Ana Klimovic
ETH Zurich
aklimovic@ethz.ch

Abstract

Service meshes are ubiquitous in today's cloud settings, providing features such as authentication, retry, and rate-limiting policies. With the advent of serverless computing, in which short-lived functions are invoked at high throughput, service meshes must efficiently handle high network policy churn and high sidecar churn. We find that state-of-the-art service meshes, such as Istio and Linkerd, slow down workloads when enforcing network policies commonly used in serverless settings. When running a production trace, we observe 38.3× higher p95 end-to-end execution times with versus without applying Istio service mesh policies. Furthermore, we observe up to 3.9× higher p95 CPU utilization per worker node. We investigate the root cause of service mesh inefficiencies, and based on insights from our study, we propose a research agenda to redesign service meshes with lightweight sidecar daemons.

CCS Concepts

• **Computer systems organization → Cloud computing**.

Keywords

Serverless, Function as a Service, Cloud Computing, Service Mesh

ACM Reference Format:
Lazar Cvetković and Ana Klimovic. 2025. Towards a Lightweight Sidecar-based Service Mesh for Serverless. In *ACM Symposium on Cloud Computing (SoCC '25), November 19–21, 2025, Online, USA.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3772052.3772210

1 Introduction

Today's serverless platforms, such as Knative [21], rely on service meshes (e.g., Istio [20], Linkerd [26], and Google Cloud Service Mesh [16]) to enforce fine-grain network and security policies such as authentication, retry, and rate-limiting policies. Additionally, service meshes are essential in multi-tenant production-grade cloud deployments to restrict connectivity between sandboxes running workloads from different tenants. Prior work has looked into service mesh performance for traditional microservice cloud environments where requests primarily experience warm starts [39, 44], and has explored performance modeling of policy processing [50]. However, for serverless computing, where users invoke short-running functions at high throughput, causing sandboxes and network policies to be frequently added and removed from the cluster, we find that the performance impact of service meshes is not well understood.

We conduct a characterization study and observe that service meshes significantly impact serverless workload performance and excessively consume cluster resources when enforcing service mesh policies, such as those outlined in Table 1. We experiment with the Knative [21] serverless platform, which runs on top of Kubernetes [23], the most widely used open-source serverless orchestrator [7].[1] We study two popular Kubernetes-based service meshes, Istio [20] and Linkerd [26], which enforce network and security policies through *sidecars* (i.e., containers injected into sandboxes running user functions). We focus on sidecar-based systems since AWS Lambda[2] [33], Huawei FunctionGraph [17, 39], and Google Cloud Run [16] all use sidecar-based designs (see §2.1).

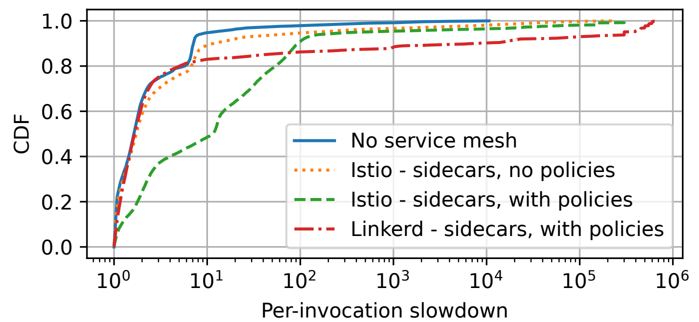

Figure 1: Per-invocation slowdown of Azure trace with 200 functions with different sidecar-based service meshes, with and without policies from Table 1 applied. Slowdown is end-to-end latency divided by the execution time from the trace.

Figure 1 shows the impact of two different service mesh implementations on end-to-end function latency for Azure Functions production trace [43] sampled down to 200 functions with [46], which we ran on a 10-worker node cluster. Applying service mesh policies from Table 1 caused a severe slowdown in the workload, affecting 90% of function invocations compared to a setup where no service mesh was deployed. In this scenario, the end-to-end latency grew 7× and 254× at p50 and p99, respectively. While not applicable for a production-grade setting due to the inability to enforce policies from Table 1, the setup with no service mesh serves as the baseline to show infrastructure costs of the cluster manager (Kubernetes) and serverless layer (Knative). Our additional baseline — service mesh with no policies — shows the overhead of the service mesh infrastructure on top of Kubernetes and Knative, i.e., the cost of sidecars without any policy processing. In this case, the performance degradation mostly affects the top three percentiles of invocations on the slowdown curve, achieving 55× higher p99 latency compared to the no service mesh case.

Our analysis reveals that service mesh overheads from Figure 1 come from two dimensions: warm and cold starts. We find that the

This work is licensed under a Creative Commons Attribution 4.0 International License.
SoCC '25, Online, USA
© 2025 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-2276-9/25/11
https://doi.org/10.1145/3772052.3772210

[1] Knative (commercially known as Google Cloud Run [15]), vHive [47], OpenWhisk [1], OpenFaaS [27], Fission [14], Cloudburst [45], and Kubeless [22] all rely on Kubernetes.

warm start overheads originate from the network stack, L4/L7 protocol parsing and processing, serialization, and kernel-user space crossings. For cold starts, we find that sidecars further reduce the elasticity of already slow-to-scale cluster managers [34, 35], because of the additional processes involved in the sandbox startup, such as sidecar container injection, sequential container startup, and multiple readiness probes. Additionally, we observe that bursty serverless workloads [43] put a lot of pressure on the service mesh control plane to propagate policies to sidecars, further affecting cold start delays and control plane node resource utilization. We observe that upon startup, sidecars receive cumbersome configurations containing the whole state of the world.

We argue that sidecars can be built more efficiently and outline a research agenda for a lightweight sidecar-based service mesh. The key insight from our service mesh characterization study is that most service mesh policies are pure compute functions. This allows for the design of lightweight sidecars using the recent innovation in low-overhead virtualization and sandbox runtimes such as Hyperlight [18], Dandelion [38], and Fixpoint [36]. These systems avoid booting a heavyweight POSIX interface in function sandboxes, which allows them to achieve high elasticity and fast execution context startup time for pure compute functions. We envision adopting these lightweight sandboxes in the design of a new sidecar-based service mesh to achieve sub-millisecond cold start time, low memory usage, and cheap policy processing for enforcing security and performance isolation in the cloud. We discuss how such a lightweight sidecar design can offer a more general programming model than eBPF, which could enable implementing more complex policy processing than what current eBPF-based service meshes support. Finally, in addition to the service mesh data plane, we propose control plane optimizations for orchestrating lightweight sidecars, such as relaxing the assumption of an all-to-all inter-function communication pattern. Contrary to conventional wisdom, we argue that the community has prematurely given up on sidecars and switched to shared L4/L7 processing daemons [19, 44], without thoroughly examining the underlying root cause of the performance issues and without addressing fundamental problems.

2 Background

In this section, we describe how today's serverless systems are built (§2.1) and discuss the related work (§2.2).

2.1 Today's Serverless Infrastructure

Architecture. Today's serverless stack is complex, consisting of multiple layers, as depicted in Figure 2. Each layer is divided into a *data plane*, through which function invocations flow, and a *control plane*, which configures the corresponding data plane. The first layer in the stack is the cluster manager, which is typically a general-purpose orchestrator, such as Kubernetes [23]. The cluster manager creates, places, and shuts down sandboxes in which functions execute, monitors workload and cluster health, manages cluster state, and provides API services to other layers in the stack. The second layer is the serverless layer (e.g., Knative [21], OpenWhisk [1], OpenFaaS [27]), which runs on top of the cluster manager and provides request buffering for cold invocations and serverless-specific autoscaling. The third layer is the service mesh layer (e.g., Istio [20],

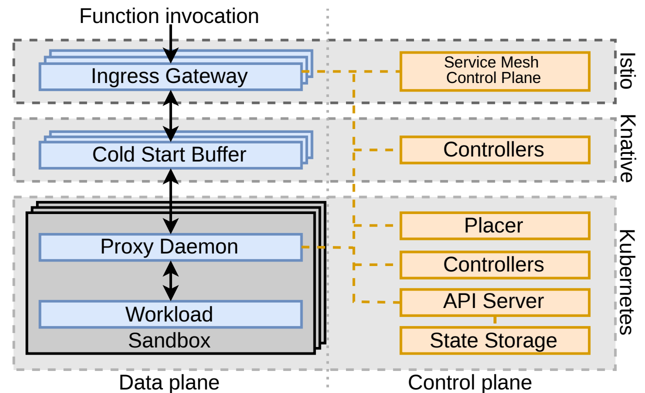

Figure 2: System architecture of a production-grade serverless stack based on Istio service mesh, Knative serverless layer, and Kubernetes cluster manager.

Linkerd [26], Consul [24], Cilium [4]), which provides an ingress gateway to the cluster and enforces network and security policies. The service mesh data plane runs *proxy daemons* on worker nodes, which enforce network and security policies by intercepting the network traffic to the sandbox. A special case of a proxy daemon is a *sidecar*, a per-sandbox container injected into a sandbox with which it shares the lifecycle. Sidecar architectures are used by AWS Lambda[2] [33], Huawei FunctionGraph [17, 39], Google Cloud Run [16], and Knative [21] serverless platforms. Finally, the service mesh control plane distributes policies, manages proxy daemon lifecycle, manages encryption keys, and translates cluster-level policy specifications into daemon-specific configurations.

Proxy daemon sharing and tradeoffs. Proxy daemons can be shared on a per-sandbox, per-function, per-worker node, or per-cluster level. Istio [20] and Consul [24] employ Envoy-based [12] per-sandbox proxy daemons – sidecars. Linkerd independently implements per-sandbox sidecars [31]. To improve resource utilization, Istio offers Ambient mode [19], which separates Layer 4 (L4) and Layer 7 (L7) network processing functionalities into two proxy daemon types. The L4 daemon type is shared on a per-node basis, while the Envoy-based L7 daemon is deployed on a per-workload (per-function) basis. However, cross-user and cross-function proxy daemon sharing must be minimized due to security and performance isolation concerns, i.e., since Envoy was not designed for multi-tenancy [32], and sidecar sharing can lead to performance interference [29, 39, 44]. Hence, sidecar sharing is not applicable for fine-grain serverless workloads, where functions and applications from many users run on the same node, and where, ideally, each serverless invocation should run in a dedicated proxy daemon. Finally, Cilium service mesh [5] is sidecar-free for L4 network processing and a limited set of L7 features with in-kernel eBPF processing. However, eBPF limitations [11, 28] make Cilium use Envoy sidecars to provide advanced L7 processing (e.g., load-balancing, ingress, TLS termination, and L7 rate limiting) [6].

[2] Firecracker [33], used by AWS Lambda, describes a per-μVM λ shim process. λ shim is co-located with the μVM, is a network proxy, and a security boundary between the single-tenant μVM and the multi-tenant cluster. The λ shim is essentially a sidecar.

Service mesh policy	OSI layer	Gran.	Cilium eBPF sup.	Linkerd sup.	Description	Behavior without a service mesh
① Request authentication	L7	F	No	No	JSON Web Token (JWT) authentication	No authentication
② Sidecar configuration	L4	F	N/A	Partial	Fine-grain sidecars configuration (ports, protocols, injections, security)	Security vulnerability
③ Authorization policy	L7	F	False	Yes	Controls who can invoke services, which HTTP methods and paths are allowed	All-to-all communication and any method/path allowed
④ Rate limiting	L7	F	No	Per-pod	Cluster-wide quota on request count (global rate limiting)	No limits
⑤ Request tracing	L7	F	Yes	Yes	HTTP header embeddings for distributed tracing and debugging	No tracing
⑥ Access logs	L7	G	Yes	Yes	Record briefly each access to a resource	No logs
⑦ Mutual TLS	L4, L7	G	Yes	Yes	End-to-end mutual authentication	Unencrypted traffic
⑧ Retry policy	L7	F	No	Yes	Retry on 5xx errors from upstream services	2 retries with no timeout
⑨ Circuit breaking	L4, L7	F	No	No	TCP/HTTP connection/request count upper limit, outlier detection (passive health checks)	$2^{32} - 1$ connections, no outlier detection

Table 1: Detailed description of service mesh policies applied in experiments in §3. The third column refers to Granularity: F – per-function; G – common for all functions. All policies are numbered for easier matching with Table 2.

2.2 Related Work

Cooper and Wire [42] explores service meshes from the programming language side, analyzing the expressiveness and complexity of policy specification languages, and proposing a simplified, developer-friendly domain-specific language, along with a compiler for mapping configuration to a few proxy daemon backends. Canal Mesh [44] is Alibaba's closed-source service mesh used in managed Kubernetes offerings for trusted workloads. The work argues for sharing proxy daemons, as Envoy-based sidecars utilize thousands of cores in their clusters. Zhu et al. in [50] propose a static performance analysis tool for Envoy-based sidecar. The work does not investigate the end-to-end performance of policies such as those in Table 1, but focuses on internal overheads such as data copying, protocol parsing, inter-process communication overheads, and the effect of different inputs. All these works focus on processing overheads for warm requests, and run microservice workloads featured by a lower degree of multi-tenancy. They do not analyze service meshes with untrusted workloads with high sandbox churn and high network policy churn, largely overlook the cold start overheads, and lack service mesh control plane characterization.

3 Service Mesh Performance Study

We characterize sidecar-based service mesh systems to evaluate their readiness to run serverless workloads efficiently. We show that the performance impact of sidecars and policy processing in service meshes has been largely underestimated, and that sidecars significantly contribute to the end-to-end latency of serverless functions and higher resource utilization. Also, sidecars negatively affect the elasticity of serverless systems. Finally, we find that the service mesh control plane distributes heavyweight configurations to sidecars. These configurations are not optimized for serverless communication patterns, but more for traditional microservices.

3.1 Service Mesh Policies

To run experiments, we develop a representative set of service mesh policies to match real-world serverless deployments and show them in Table 1. We explored what knobs in serverless systems, such as [2, 8, 15, 21], developers can configure on function registration and in online dashboards. For example, Cloud Run [15] allows users to configure end-to-end request authentication and restrict ports and protocols the function is allowed to use. Google Cloud Console dashboard offers real-time traffic monitoring, request tracing, and access log viewing. AWS Lambda [2] offers global rate limiting [3]. As serverless clusters are multi-tenant, the data plane traffic is encrypted with mutual TLS (mTLS). We find that almost all user-configurable knobs are inherently L7 network processing features, except for a few policies operating at the L4 level. Finally, we match the policy application granularity to what cloud providers offer.

3.2 Experiment Setup

Hardware setup. We experiment on an 11-node `xl170` Cloudlab cluster [9]. Each node has an Intel E5-2640v4 CPU with 10 physical cores (SMT disabled), 64 GB of DDR4-2400 DRAM, Intel DC S3520 480 GB 6G SATA SSD, and Mellanox ConnectX-4 25 GB NIC. One node runs the control plane, and all the other nodes are workers.

Software setup. None of the serverless layers natively supports all features from Table 1. We rely on service meshes to provide and enforce them. We use Istio as a sidecar-based service mesh with the richest feature set [42], which is compatible with Knative-on-Kubernetes serverless system. We deploy Linkerd service mesh as another baseline. Linkerd lacks an ingress gateway; hence, we use Istio as an ingress gateway. Not all policies were portable from Istio to Linkerd (see Table 1), as the latter does not support all the features Istio offers (e.g., global rate-limiting, circuit breaking). Also,

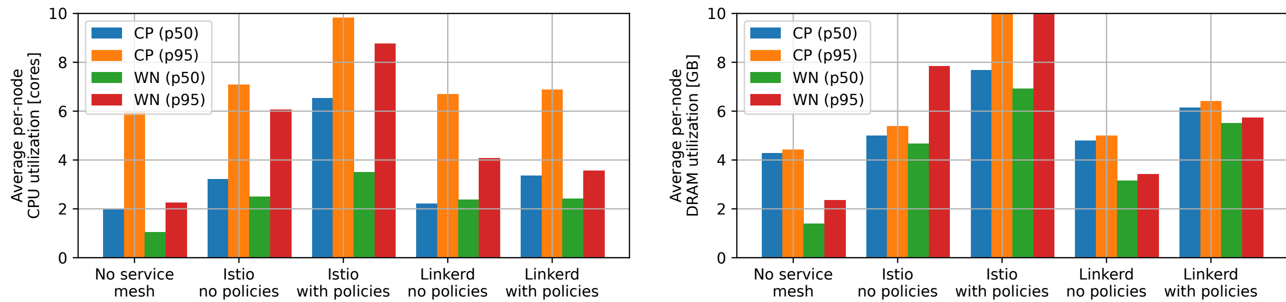

Figure 3: Average *per-node* CPU utilization (left) and *per-node* DRAM utilization (right) while running the 200-function trace. The cluster has 1 control plane node and 10 worker nodes. CP and WN refer to the control plane and worker node, respectively.

since Knative is not compatible with Ambient service mesh [30], we could not run experiments in a non-sidecar setting. We run Kubernetes v1.29.1, Knative v1.13.1, Istio v1.20.2, and Linkerd v25.5.2 with the default configurations and resource reservations (100m vCPU and 128 MB DRAM for Istio). We pre-cache container images on worker nodes and use InVitro for load generation [46].

Workload. We run Azure Functions trace [43] for 30 minutes, which we sample down using InVitro methodology [46] from 44.7K functions (day 6, hour 8) to 200 functions with 32.5K invocations. This is the biggest trace we can run without cluster saturation in any experiment scenario. The workload executes the SQRTSD x86 instruction for the amount of time given in the trace. We conduct experiments in three settings: with no service mesh deployed; with service mesh deployed, but without any policies applied (to show the sidecar overheads); and with service mesh deployed and with policies from Table 1 applied (to show the policy processing and configuration overheads on top of the service mesh overhead).

3.3 Data Plane Performance

Data plane components are shown in blue in Figure 2, whose performance we assess by measuring the per-invocation slowdown defined as end-to-end execution time divided by the function execution time given in the trace. We separately consider cold and warm starts. We define cold starts as scenarios that trigger the creation of a new sandbox in which the workload and the sidecar execute.

Warm starts. Figure 1 shows the per-invocation slowdown for the 200-function trace under multiple scenarios. Once policies from Table 1 are applied in the Istio service mesh, the workload performance severely degrades, raising the p50 and p95 per-invocation slowdowns by 7× and 38.3×, respectively, and affecting 90% of function invocations compared to when the service mesh is not deployed. The overheads primarily come from network stack, L4/L7 protocol parsing and processing, serialization, and data copying between the user and kernel mode, as Envoy-based sidecars do not use kernel bypass [10]. However, when Istio sidecars are deployed, but with no service mesh policies from Table 1 applied, the performance degradation affects warm starts mildly. In this case, the median slowdown is 1.06× higher compared to the scenario with no service mesh deployed, which we attribute to one extra network stack traversal on each invocation's critical path. The warm start experiments highlight an opportunity to optimize service mesh policy processing cost, which we further elaborate on in §4.

To understand the cost of individual service mesh policies, we conduct a policy ablation study. We find that mutual TLS (⑦), authorization (③), and request authentication (①) policies cause the biggest per-invocation slowdown, since they involve extra work on the cryptography side and/or costly L7 header parsing. Furthermore, we reveal that there is a base cost of deploying service mesh policies, regardless of whether one or more policies are applied in the cluster. The reason is the activation of the packet processing pipeline inside the sidecar, which includes L4/L7 processing and Envoy filter execution. With no policies applied, the sidecar is just a proxy that takes a cheaper code path, forwarding packets to their final destination without additional processing and decision-making.

We run an additional baseline, Linkerd service mesh, which lacks support for some policies from Table 1. Without policies applied, Linkerd matches the performance with no service mesh deployed. With policies applied, 80% of invocations exhibit performance as with no sidecars, outperforming Istio. This is because Linkerd does not use Envoy and optimizes policy processing within sidecars. However, the other 20% of invocations exhibit long tail latency, hugely underperforming Istio. With Linkerd, multiple experiment runs exhibited a failure rate of ∼11%, compared to just 2% with Istio.

Cold starts. The top-most three percentiles from Figure 1, i.e., the tail, are cold starts. Istio and Linkerd achieve 254× and 1008× higher p99 per-invocation slowdown compared to when the service mesh is not deployed, respectively. These slowdowns are unacceptably high, since serverless functions are bursty, sporadically invoked, and short-lived (half of Azure trace functions complete within a second [43]), i.e., cold starts are frequent [34, 35]. While the poor elasticity of serverless infrastructure has been described in [34, 35], we find that sidecars further negatively affect elasticity, which has not been addressed by prior works such as [39, 42, 44, 50].

To extend the cold start study, we run a microbenchmark at low load (1 cold start per second) and measure the end-to-end cold start latency. Without deploying a service mesh, cold start latency is 1.2 seconds, similar to reports in [34]. With sidecars deployed, we observe an end-to-end cold start time of ∼4 seconds. We notice that Istio and Linkerd inject two additional containers into users' workloads (in addition to Knative's queue-proxy sidecar). The first container to run and terminate is the initialization container (init

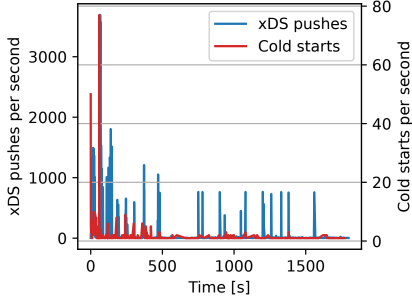

Figure 4: xDS protocol transaction load.

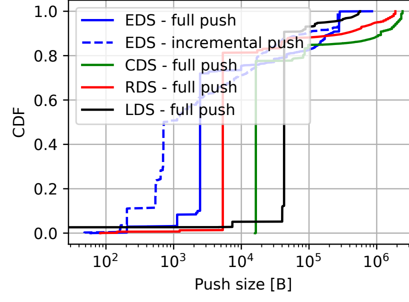

Figure 5: xDS transaction payload size.

Policy	Input	Output	State
①	L7 hdr.	A/D	No
②	L4/L7 hdr.	A/D	No
③	L7 hdr.	A/D	No
④	L7 hdr.	A/D	Yes
⑤, ⑥	L4/L7 hdr.	None	No
⑦	L4/L7 hdr.	A/D	No
⑧, ⑨	L7 hdr.	A/D	Yes

Table 2: Policies from Table 1, their inputs and outputs, and the associated state. A/D stands for Allow/Deny; hdr. for header.

container), which configures the network namespace and routing rules to forward all the traffic through the Envoy sidecar that will be created shortly after. Traffic forwarding is done with `iptables`, which is not the state-of-the-art solution. We measure it takes 95 ms to create and start the init container, excluding the network configuration. The second container to run is the Envoy sidecar, which takes 81 ms to create and start. It is only then that the user's workload starts. We discuss the sidecar policy configuration overheads separately in §3.4. The cold start time is higher because each service mesh container needs to be injected into the sandbox, pass the readiness probe, and update the cluster state database. These steps happen sequentially for each container inside the sandbox, and require numerous interactions with the control plane, which bottlenecks cold starts [34]. Consequently, since the cold start process is more complex in the service mesh scenario, we observe the cold start throughput reduction from 2.5 to 2 cold starts per second.

Worker node resource utilization. Figure 3 shows the per-worker node CPU utilization averaged across machines. Istio sidecars with policies applied consume 2.5 and 6.5 additional CPU cores per worker node, at p50 and p95, respectively, compared to the baseline with no service mesh. In a 10-worker node cluster, this maps to 25 to 65 additional CPU cores for which the cloud provider needs to provision resources. Figure 3 additionally shows the per-worker node DRAM utilization averaged across machines, highlighting the major drawback of today's sidecars — high memory usage. Linkerd experiments show sidecars can come at a much lower CPU and DRAM cost compared to Envoy-based sidecars. When comparing all scenarios to the baseline with no service mesh, the results show the opportunity to further improve resource efficiency with sidecars.

3.4 Control Plane Performance

In this section, we characterize the Istio service mesh control plane. We monitor the control plane traffic, payload size, and resource utilization. Control plane components are shown in orange in Figure 2.

Service mesh component configuration. Sidecars receive configuration updates from the control plane through a family of discovery services [13], collectively called xDS. Cluster Discovery Service (CDS) informs sidecars about changes in Kubernetes services (logical groups of pods). Through Endpoint Discovery Service (EDS), sidecars get to know about changes in pods (e.g., lifecycle, IP address, and port changes). Route Discovery Services (RDS) instruct service mesh members what network rules and policies to apply, whereas the Listener Discovery Service (LDS) configures Envoy proxies for how to listen and accept the traffic. When sidecars start, they register with the service mesh control plane, fetch the configuration from it, and apply policies before serving the traffic.

While running the 30-minute trace with Istio policies applied, we monitor the number of xDS transactions over time and present them in Figure 4. We also plot the cold start count over time and find that cold start spikes are correlated with the configuration push spikes. The bursty nature of serverless functions [43] puts pressure on the cluster manager to create new sandboxes and on the control plane to inject the sidecars into the sandboxes and push configurations to sidecars. This led to more than 3000 transactions for handling just 80 cold starts, which increases CPU and DRAM utilization of the control plane node, as shown in Figure 3. At p50, sidecar orchestration in Istio consumes 1.2 more CPU cores and 4.6 extra CPU cores when policies are applied, compared to the scenario when the service mesh is not deployed.

We further examine the size of configurations exchanged via xDS. Figure 5 shows that the payload size distribution varies greatly, with 80% of transactions ranging from 1 kB to 14 kB, whereas the other 20% go up to 3 MB in size. We notice that configuration traffic going to the sidecars is much less variable in size, and is responsible for the vertical lines on Figure 5, whereas the fifth quintile is the traffic going to the ingress gateway. Huge payloads paired up with bursts from Figure 4 lead to serialization costs, on both the sender and receiver side, manifesting as high resource utilization.

Finally, we reveal that initial configuration pushes, i.e., configuration updates on sidecar creations, cost more than incremental (delta) pushes. They transfer the whole state of the world to sidecars, as service meshes assume all-to-all inter-function communication patterns. However, this pattern is not necessarily the case for serverless applications, since workloads are organized into standalone functions, linear pipelines, or workflows [40, 43]. Unfortunately, initial configuration pushes are common for serverless workloads, since sandboxes are short-lived [43] and function invocation patterns result in high sandbox churn, i.e., high sidecar churn [34].

3.5 Generalizability Analysis

While we focus on the Kubernetes environment, the system challenges we explore are orthogonal to the choice of cluster manager.

Any alternative orchestrator, such as [34, 37, 48], would still need sidecars for inter-sandbox communication. Figures 1, 3, and 4 show service mesh costs, which exist regardless of whether Kubernetes is the orchestrator. However, each orchestrator has a different base cost, on top of which service mesh costs are added. Although our study focuses on sidecar-based design, which is used by commercial platforms, we do not claim the results generalize to commercial systems, as we cannot measure internal metrics. However, these systems are likely to encounter some of the challenges we describe.

4 Towards A Lightweight Service Mesh for Serverless Computing

The characterization study in §3 revealed numerous optimization opportunities. We now discuss how we can leverage recent advancements in sandbox runtimes to design lightweight sidecars with cheap L4/L7 policy processing, high elasticity, and lower resource footprint. We make a case for rethinking design decisions and optimizations built into the control plane of today's systems.

4.1 Data Plane

Service meshes designs have shifted in recent years towards proxy daemon sharing, primarily to reduce resource consumption [19, 44]. Although high resource consumption has traditionally been used as an argument against per-sandbox sidecars, we argue that sidecars do not fundamentally need to be as expensive as §3 shows.

Our insight is based on recent work, which shows that *pure compute functions* can be securely isolated in sandboxes that are quick to boot [18, 38, 49]. The key observation is that the slow part of booting a traditional sandbox in serverless environments today is not the μVM (one can boot a KVM instance in ~50 μs [38]). Rather, the slow part is initializing a fully-compliant POSIX environment inside the sandbox, as this requires loading and initializing device drivers and the complete network stack. Since pure compute functions do not rely on POSIX calls, they can execute in secure sandboxes that are much faster to boot. This insight allows systems, such as Hyperlight, to boot a fresh VM within 2 ms [18], and Dandelion to achieve a sub-millisecond execution context startup time [38], in contrast to Firecracker μVM startup time of 125 ms [33]. Furthermore, Dandelion showed to reduce the average memory usage by 96% on average when running the Azure production trace compared to Firecracker and to be able to support complex applications such as serverless data analytics and agentic AI query processing [38].

To assess the feasibility of expressing service mesh policies as pure compute functions, we study the inputs, outputs, and cross-invocation state of each policy in Table 1. As Table 2 shows, most L4 and L7 policies are pure compute functions, with L4 and/or L7 headers as inputs, and binary outputs (either allow or deny the request). However, some policies maintain cross-invocation state (e.g., rate limiting), which the sandbox runtime can handle with an in-memory caching layer, such as the Faa$T cache [41]. Also, the runtime transparently injects the data into the execution context and participates in cross-node data/state exchange and aggregation.

We therefore propose to leverage technologies, such as Hyperlight [18], Dandelion [38], Virtines [49], and Fixpoint [36] to build a lightweight sidecar-based service mesh. The fast startup time and high elasticity of these runtimes make it practical to boot a new sandbox on each request's critical path. This opens an opportunity for enforcing policies in more secure sidecars, spawned separately for *each request*, compared to today's sidecars that are *shared* between requests targeting the same sandbox. Since sidecar warm starts no longer exist, memory usage would be reduced. The proposed sidecars are compatible with existing FaaS platforms and runtimes. We discuss Dandelion [38] as an example. It already integrates with the Dirigent cluster manager [34] and we see no fundamental obstacles for Knative integration. Also, the sidecars are compatible with user functions running in legacy runtimes (e.g., containerd, Firecracker). In this scenario, the sidecar runtime maintains long-lived connections with user functions controlled by the legacy runtime, and the sidecar runtime does all the network receive/transmit operations (e.g., Dandelion's communication functions). Running a user function in a Dandelion sandbox yields another benefit — user function and inbound and outbound policy processing can be organized into a workflow, which Dandelion natively supports, allowing for more efficient execution and scheduling. Finally, the proposed model, combined with kernel bypass network stacks, could become an alternative to eBPF packet processing, as it could support a richer set of L4 and L7 policies compared to what eBPF can do today due to the verifier constraints [10, 11, 28].

4.2 Control Plane

Figure 2 shows the loosely-coupled architecture of today's serverless systems, in which each layer issues decisions separately, and whose control plane can be optimized. Although critical for serverless, cross-layer information exchange is known to be expensive in Kubernetes-based systems [34]. Inspired by Dirigent [34], we propose layer fusing and optimizing transfers of critical cluster state, which includes sandbox and sidecar configurations, routing information, readiness and autoscaling signals, and placement decisions. Also, Dirigent can be leveraged to orchestrate a large number of sidecars at a high throughput, which no longer need to undergo the injection process in the Kubernetes API Server on cold starts.

We also propose optimizing sidecar configuration protocols, such that only the bare minimum information is passed to sidecars. Today's systems, such as Istio, broadcast the whole state-of-the-world to every sidecar on startup, as they assume allowed communication between all sandboxes in the cluster, i.e., optimize for all-to-all communication [25]. However, serverless workloads do not require all-to-all connectivity; hence, we advocate for all communication being forbidden by default, except for what is explicitly allowed.

5 Conclusion

Service meshes enforce network and security policies and are ubiquitous in today's cloud. We characterized sidecar-based service meshes on a serverless workload and found that today's systems feature expensive policy processing, long sidecar startup time, and high resource consumption. Finally, we presented a vision of how to build a more efficient, lightweight sidecar-based service mesh.

Acknowledgments

We thank the anonymous reviewers and our shepherd, Divyanshu Saxena, for their feedback and guidance. Lazar Cvetković is supported by the Swiss National Science Foundation (project TMSGI2_218019).

References

[1] Apache OpenWhisk. Available at https://openwhisk.apache.org/.
[2] AWS Lambda. Available at https://aws.amazon.com/lambda/.
[3] AWS Lambda Quotas. Available at https://docs.aws.amazon.com/lambda/latest/dg/gettingstarted-limits.html.
[4] Cilium. Available at https://cilium.io/.
[5] Cilium Service Mesh. Available at https://docs.cilium.io/en/stable/network/servicemesh/index.html.
[6] Cilium Service Mesh – Everything You Need To Know. Available at https://isovalent.com/blog/post/cilium-service-mesh/.
[7] Cloud Native Computing Foundation Survey – 2020. Available at https://www.cncf.io/wp-content/uploads/2020/11/CNCF_Survey_Report_2020.pdf.
[8] Cloud Run for Anthos. Available at https://cloud.google.com/anthos/run.
[9] Cloudlab. Available at https://www.cloudlab.us/.
[10] eBPF and Sidecars – Getting the Most Performance and Resiliency out of the Service Mesh. Available at https://tetrate.io/blog/ebpf-and-sidecars-getting-the-most-performance-and-resiliency-out-of-the-service-mesh.
[11] eBPF on Linux. Available at https://docs.ebpf.io/linux/.
[12] Envoy Proxy. Available at https://www.envoyproxy.io/.
[13] Envoy Proxy – xDS REST and gRPC protocol. Available at https://www.envoyproxy.io/docs/envoy/latest/api-docs/xds_protocol.
[14] Fission. Available at https://github.com/fission/fission.
[15] Google Cloud Run. Available at https://cloud.google.com/run.
[16] Google Service Mesh. Available at https://cloud.google.com/products/service-mesh.
[17] Huawei FunctionGraph. Available at https://www.huaweicloud.com/eu/product/functiongraph.html.
[18] Introducing Hyperlight: Virtual machine-based security for functions at scale. Available at https://opensource.microsoft.com/blog/2024/11/07/introducing-hyperlight-virtual-machine-based-security-for-functions-at-scale/.
[19] Istio Ambient Mode. Available at https://istio.io/latest/docs/ambient/.
[20] Istio considerations for large clusters. Available at https://www.istio.io/.
[21] Knative. Available at https://knative.dev/.
[22] Kubeless. Available at https://kubeless.io/.
[23] Kubernetes. Available at https://kubernetes.io/.
[24] Kubernetes considerations for large clusters. Available at https://www.consul.io/.
[25] Kubernetes Network Model. Available at https://kubernetes.io/docs/concepts/services-networking/#the-kubernetes-network-model.
[26] Linkerd. Available at https://linkerd.io/.
[27] OpenFaaS. Available at https://www.openfaas.com/.
[28] Pitfalls of relying on eBPF for security monitoring (and some solutions). Available at https://blog.trailofbits.com/2023/09/25/pitfalls-of-relying-on-ebpf-for-security-monitoring-and-some-solutions/.
[29] Service Mesh Deployment Best Practices for Security and High Availability. Available at https://tetrate.io/blog/service-mesh-deployment-best-practices-for-security-and-high-availability.
[30] Support for Istio Ambient Mesh – GitHub issues. Available at https://github.com/knative-extensions/net-istio/issues/1360.
[31] The Linkerd Proxy. Available at https://github.com/linkerd/linkerd2-proxy.
[32] Why no L7 processing on the local node? Available at https://istio.io/latest/blog/2022/introducing-ambient-mesh/.
[33] Agache, A., Brooker, M., Iordache, A., Liguori, A., Neugebauer, R., Piwonka, P., and Popa, D.-M. Firecracker: Lightweight Virtualization for Serverless Applications. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)* (2020), pp. 419–434.
[34] Cvetković, L., Costa, F., Djokic, M., Friedman, M., and Klimovic, A. Dirigent: Lightweight Serverless Orchestration. In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles* (New York, NY, USA, 2024), SOSP '24, Association for Computing Machinery, p. 369–384.
[35] Cvetković, L., Fonseca, R., and Klimovic, A. Understanding the Neglected Cost of Serverless Cluster Management. In *Proceedings of the 4th Workshop on Resource Disaggregation and Serverless* (2023), WORDS '23, p. 22–28.
[36] Deng, Y., Montemayor, A., Levy, A., and Winstein, K. Computation-centric networking. In *Proceedings of the 21st ACM Workshop on Hot Topics in Networks* (New York, NY, USA, 2022), HotNets '22, Association for Computing Machinery, p. 167–173.
[37] Hindman, B., Konwinski, A., Zaharia, M., Ghodsi, A., Joseph, A. D., Katz, R. H., Shenker, S., and Stoica, I. Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center. In *Proceedings of the 8th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2011, Boston, MA, USA, March 30 - April 1, 2011* (2011), D. G. Andersen and S. Ratnasamy, Eds., USENIX Association.
[38] Kuchler, T., Li, P., Zhang, Y., Cvetković, L., Goranov, B., Stocker, T., Thomm, L., Kalbermatter, S., Notter, T., Lattuada, A., and Klimovic, A. Unlocking True Elasticity for the Cloud-Native Era with Dandelion. In *Proceedings of the ACM SIGOPS 31st Symposium on Operating Systems Principles* (New York, NY, USA, 2025), SOSP '25, Association for Computing Machinery, p. 944–961.
[39] Liu, Q., Du, D., Xia, Y., Zhang, P., and Chen, H. The Gap Between Serverless Research and Real-world Systems. In *Proceedings of the 2023 ACM Symposium on Cloud Computing, SoCC 2023, Santa Cruz, CA, USA, 30 October 2023 - 1 November 2023* (2023), ACM, pp. 475–485.
[40] Mahgoub, A., Yi, E. B., Shankar, K., Elnikety, S., Chaterji, S., and Bagchi, S. ORION and the Three Rights: Sizing, Bundling, and Prewarming for Serverless DAGs. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)* (Carlsbad, CA, July 2022), USENIX Association, pp. 303–320.
[41] Romero, F., Chaudhry, G. I., Goiri, I. n., Gopa, P., Batum, P., Yadwadkar, N. J., Fonseca, R., Kozyrakis, C., and Bianchini, R. Faa$T: A Transparent Auto-Scaling Cache for Serverless Applications. In *Proceedings of the ACM Symposium on Cloud Computing* (New York, NY, USA, 2021), SoCC '21, Association for Computing Machinery, p. 122–137.
[42] Saxena, D., Zhang, W., Pailoor, S., Dillig, I., and Akella, A. Copper and Wire: Bridging Expressiveness and Performance for Service Mesh Policies. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1, ASPLOS 2025, Rotterdam, The Netherlands, 30 March 2025 - 3 April 2025* (2025), L. Eeckhout, G. Smaragdakis, K. Liang, A. Sampson, M. A. Kim, and C. J. Rossbach, Eds., ACM, pp. 233–248.
[43] Shahrad, M., Fonseca, R., Goiri, I., Chaudhry, G., Batum, P., Cooke, J., Laureano, E., Tresness, C., Russinovich, M., and Bianchini, R. Serverless in the Wild: Characterizing and Optimizing the Serverless Workload at a Large Cloud Provider. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, USENIX Association, pp. 205–218.
[44] Song, E., Song, Y., Lu, C., Pan, T., Zhang, S., Lu, J., Zhao, J., Wang, X., Wu, X., Gao, M., Li, Z., Fang, Z., Lyu, B., Zhang, P., Wen, R., Yi, L., Zong, Z., and Zhu, S. Canal Mesh: A Cloud-Scale Sidecar-Free Multi-Tenant Service Mesh Architecture. In *Proceedings of the ACM SIGCOMM 2024 Conference* (New York, NY, USA, 2024), ACM SIGCOMM '24, Association for Computing Machinery, p. 860–875.
[45] Sreekanti, V., Wu, C., Lin, X. C., Schleier-Smith, J., Gonzalez, J., Hellerstein, J. M., and Tumanov, A. Cloudburst: Stateful Functions-as-a-Service. *Proc. VLDB Endow. 13*, 11 (2020), 2438–2452.
[46] Ustiugov, D., Park, D., Cvetković, L., Djokic, M., Hè, H., Grot, B., and Klimovic, A. Enabling In-Vitro Serverless Systems Research. In *Proceedings of the 4th Workshop on Resource Disaggregation and Serverless* (New York, NY, USA, 2023), WORDS '23, Association for Computing Machinery, p. 1–7.
[47] Ustiugov, D., Petrov, P., Kogias, M., Bugnion, E., and Grot, B. Benchmarking, analysis, and optimization of serverless function snapshots. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (2021), pp. 559–572.
[48] Vavilapalli, V. K., Murthy, A. C., Douglas, C., Agarwal, S., Konar, M., Evans, R., Graves, T., Lowe, J., Shah, H., Seth, S., Saha, B., Curino, C., O'Malley, O., Radia, S., Reed, B. C., and Baldeschwieler, E. Apache Hadoop YARN: yet another resource negotiator. In *ACM Symposium on Cloud Computing, SOCC '13, Santa Clara, CA, USA, October 1-3, 2013* (2013), G. M. Lohman, Ed., ACM, pp. 5:1–5:16.
[49] Wanninger, N. C., Bowden, J. J., Shetty, K., Garg, A., and Hale, K. C. Isolating Functions at the Hardware Limit with Virtines. In *Proceedings of the Seventeenth European Conference on Computer Systems* (2022), EuroSys '22.
[50] Zhu, X., She, G., Xue, B., Zhang, Y., Zhang, Y., Zou, X. K., Duan, X., He, P., Krishnamurthy, A., Lentz, M., Zhuo, D., and Mahajan, R. Dissecting Service Mesh Overheads, 2022.