Canal Mesh: A Cloud-Scale Sidecar-Free Multi-Tenant Service Mesh Architecture

Enge Song[†⋆], Yang Song[†⋆], Chengyun Lu[†], Tian Pan[†□], Shaokai Zhang[†], Jianyuan Lu[†], Jiangu Zhao[†], Xining Wang[†], Xiaomin Wu[†], Minglan Gao[†], Zongquan Li[†], Ziyang Fang[†], Biao Lyu[*†], Pengyu Zhang[†], Rong Wen[†], Li Yi[†], Zhigang Zong[†□], Shunmin Zhu[‡†□]
[†]Alibaba Cloud [*]Zhejiang University [‡]Tsinghua University
alibaba_cloud_network@alibaba-inc.com

ABSTRACT

In recent years, service mesh frameworks have gained significant popularity in building microservice-based applications. A key component of these frameworks is a proxy in each K8s pod, named sidecar, which handles inter-pod traffic. Our empirical measurement reveals that such per-pod sidecars cause numerous problems, including intrusion into the user pod, excessive resource occupation, significant overhead in managing many sidecars, and performance degradation caused by passing traffic through the sidecar.

In this paper, we introduce *Canal Mesh*, a cloud-scale sidecar-free multi-tenant service mesh architecture. Canal decouples service mesh functions from the user cluster and deploys a centralized mesh gateway in the public cloud to handle these functions, thus reducing user intrusion and orchestration overhead. Through service consolidation and multi-tenancy, the infra costs of service mesh are also reduced. To address the rising issues due to cloud-based deployment, such as service availability, tenant isolation, noisy neighbor, service elasticity, and additional infra costs, we leverage techniques including hierarchical failure recovery, shuffle sharding, rapid intervention, precise scaling, cloud infra reuse and resource aggregation, *etc.* Our evaluation shows that Canal Mesh's performance, resource consumption, and control plane overhead are significantly better than Istio and Ambient. We also share experiences from years of deploying Istio and Canal in production.

CCS CONCEPTS

• **Networks → Cloud computing**; **Middle boxes / network appliances**; **Network performance analysis**.

KEYWORDS

microservice, service mesh, sidecar, public cloud, multi-tenancy, service consolidation, centralized mesh gateway

ACM Reference Format:

Enge Song, Yang Song, Chengyun Lu, Tian Pan, Shaokai Zhang, Jianyuan Lu, Jiangu Zhao, Xining Wang, Xiaomin Wu, Minglan Gao, Zongquan Li, Ziyang Fang, Biao Lyu, Pengyu Zhang, Rong Wen, Li Yi, Zhigang Zong, Shunmin Zhu. 2024. Canal Mesh: A Cloud-Scale Sidecar-Free Multi-Tenant Service Mesh Architecture. In *Proceedings of ACM SIGCOMM 2024 Conference (ACM SIGCOMM '24).* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3651890.3672221

[⋆]Both authors contributed equally to this work. [□]Co-corresponding authors.

ACM SIGCOMM '24, August 4–8, 2024, Sydney, NSW, Australia

ACM ISBN 979-8-4007-0614-1/24/08
https://doi.org/10.1145/3651890.3672221

1 INTRODUCTION

Service mesh has emerged as an infrastructure that enables service-to-service communication over networks [41, 42, 47, 51, 55, 68]. Major cloud providers, such as AWS [20], Azure [21], GCP [28], Alibaba [19], have launched service mesh-based products that simplify building and managing microservice-based applications. A key component in many service mesh frameworks, such as Istio [30] and Linkerd [32], is a proxy named sidecar. The sidecar takes charge of the network traffic management for a pod, handling tasks such as policy-based routing, percentage-based traffic splitting, rate limiting, *etc.* Decoupling network functions into a sidecar allows flexible traffic management without changing the user's service logic.

After deploying Istio on Alibaba Cloud to serve customers for over four years [19], our production data indicates that such a per-pod sidecar deployment causes the following problems:

- **Intrusion:** The sidecar sits inside the user pod and runs alongside the user app. Its presence in the user pod may cause potential security and stability issues. For example, memory leakage in a sidecar could crash the user app within the same pod.
- **Throughput and latency:** The throughput could degrade by 3x~7x and the latency could increase by 3x~7x after adopting the sidecar [61]. This is because each request needs to be redirected to the sidecar, which introduces extra processing steps [61], significantly increasing the performance overhead [2–4, 11, 61].
- **Resource occupation:** When loaded with complex network and security configurations, the sidecar consumes substantial CPU and memory, which are purchased by users originally for running their apps. Moreover, maintaining optimal performance of user services requires that the sidecar's CPU utilization stay unsaturated (*e.g.*, $< 45\%$), which exacerbates the resource issue.
- **Orchestration:** Since each pod has one sidecar, and a cloud service could have $O(100k)$ pods, the overhead of orchestrating configurations for these sidecars is extremely large [4].

To address these problems, Cilium [23] and SPRIGHT [61] use eBPF [25] to reduce packet processing latency and CPU consumption. Despite their advantages, the eBPF-based solutions have limited programmability, making it challenging to support flexible Layer 7 (L7) processing, such as HTTP. According to our investigation with customers, 80%~95% of service mesh users need L7 functions. Therefore, both Cilium and SPRIGHT are insufficient to meet our customers' demands. Ambient [38] is a pioneering

solution addressing these issues. It decouples the complex L7 functions from the sidecar to an optional proxy and deploys a per-node shared proxy for handling L4 functions. Besides, by including Envoy [26], Cilium can also support L7 processing [12]. However, as open-source solutions for single-tenant usage, both Ambient and Cilium with Envoy still have their L4/L7 proxies reside within the user cluster, leaving the issues of intrusion and resource unresolved.

To address the above issues with incomplete decoupling of sidecars from the user cluster, we propose *Canal Mesh*, a cloud-based sidecar-free multi-tenant service mesh with minimal intrusion, high performance, low cost, and minor orchestration overhead. Canal adopts an aggressive decoupling strategy by pulling the service mesh out of the user cluster and introducing a centralized mesh gateway in the public cloud to remotely handle these functions. Through service consolidation and multi-tenancy at the gateway, the costs of using the service mesh can be greatly reduced. Decoupling the service mesh from the user cluster frees us from the constraints imposed by open-source software (*e.g.*, K8s [31]) when developing new features or optimizing its performance. Additionally, cloud providers can better reuse the existing cloud infra and leverage years of experiences in multi-tenant cloud management.

However, such an architecture still faces issues in deployment. For instance, remote service mesh proxies introduce security and observability concerns. Furthermore, the multi-tenant mesh gateway brings about additional issues like service availability, tenant isolation, noisy neighbor, service elasticity, and extra infra costs.

To ensure functional equivalence after remote deployment, we deploy a minimal-feature on-node proxy to address security and observability concerns. In addition, eBPF-based kernel bypass and remote mTLS acceleration are leveraged for higher performance. To address the service availability, tenant isolation, and noisy neighbor issues of the consolidated and multi-tenant service mesh, we employ techniques such as hierarchical failure recovery with replicas across availability zones (AZs), shuffle sharding, and anomaly detection-triggered rapid intervention (via scaling/migration/throttling). To provide sufficient service elasticity for supporting tens of thousands of stateful services in the mesh gateway, we use root cause analysis to pinpoint the services with rapid traffic growth and perform precise scaling of them. To maximally reduce the infra costs for further user expense saving, we conduct load balancer (LB) disaggregation for cloud infra reuse and session aggregation via tunneling.

Our major contributions are summarized as follows:

- We propose the world's first cloud-based multi-tenant service mesh for production deployment. By deploying service mesh functions remotely in the public cloud, we address the issues of intrusion, performance, resource, and orchestration overhead of Istio and Ambient. Specifically, Canal achieves throughput 12.3x and 2.3x higher than Istio and Ambient, with latency 1.7x and 1.3x lower. Canal's CPU consumption is 12x~19x and 4.6x~7.2x lower than Istio and Ambient. Canal's configuration completion time for creating hundreds of pods is 1.5x~2.1x and 1.2x~1.5x smaller than Istio and Ambient. Canal's southbound bandwidth occupation is 9.8x and 4.6x lower than Istio and Ambient.
- We present a production-validated system that addresses the issues due to remote deployment, service consolidation, and multi-tenancy. It achieves low cost, high availability and elasticity. The data collected from multiple regions shows that, in the presence of a noisy neighbor, Canal can reduce gateway backend CPU utilization from 80% to 30% in dozens of seconds with sufficient resources allocated to the noisy neighbor. In the event of a tenant being attacked, Canal supports migrating the affected services to a sandbox within seconds to prevent impact on other tenants. Additionally, through cloud infra reuse and resource aggregation, deployment costs have also seen a reduction of 55%~70%.
- We share our experiences of deploying Istio on Alibaba Cloud for four years and Canal for one year. We observe that with the consolidated mesh gateway, health check probes significantly outnumber user app traffic (up to 515x). We reduce the health checks by 99.6% with multi-level aggregation. We also discuss various abnormal cases of the mesh gateway in the past, such as the "query of death" caused by hotspot social media events and our responses. We delve into deployment issues of technical solutions, such as eBPF-based kernel bypass and crypto offloading with Intel processors. We share insights into customized deployment requirements from our customers with high security demands. We also analyze the overhead and limitations of Canal.

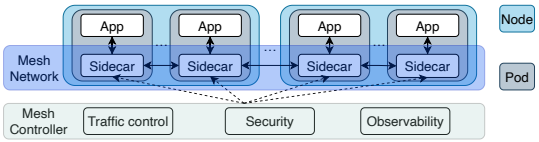


Figure 1: Per-pod sidecar deployment (*e.g.*, Istio).

2 BACKGROUND AND MOTIVATION

2.1 Problems of Per-Pod Sidecar

We have identified the following problems with Istio, a service mesh with per-pod sidecar. Some of these problems have also been discussed in the open-source and research communities [38, 69]. After deploying Istio for our customers in production for four years [19], we observed that some problems escalate at cloud-scale.

High intrusiveness into user pods. As shown in Fig. 1, for per-pod sidecar deployment, a sidecar is embedded within the pod of each user app to handle communication tasks. The sidecar and app coexist, sharing pod resources. To ensure uninterrupted communication for the app and prevent resource waste from an isolated sidecar, both the sidecar and app are designed to be created, destroyed, and scaled together, sharing the same lifecycle [5, 37]. However, such a design introduces stability and security issues, *e.g.*, memory leakage in the sidecar may cause the app to crash, and upgrading the sidecar will require a pod restart, disrupting the app.

Performance degradation due to extra steps. As traffic needs to be processed by the sidecar, outbound traffic from the user app is redirected to the sidecar (*e.g.*, using iptables [10, 29]), introducing extra processing steps, as shown in Fig. 21 in the Appendix. On both the client and server sides, the traffic redirection introduces two extra times of context switch, memory copy, and protocol stack processing [61]. Moreover, the sidecar needs to perform complex L7 tasks such as CPU-intensive TLS crypto, which may also lead to significant performance degradation [2–4, 11, 61].

Excessive resource consumption. As the sidecar is deployed within the user pod, it consumes resources initially allocated for user apps [2, 3, 69]. Our quantitative findings reveal that, when deploying K8s clusters with Istio in production, the sidecar consumes a considerable amount of resources. As shown in Table 1, for a major customer with a K8s cluster of 500 nodes and 15k pods,

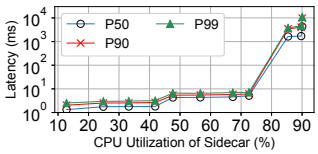


Figure 2: Sidecar CPU usage *vs* end-to-end latency.

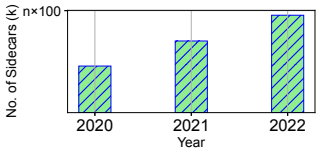


Figure 3: #Sidecars for a major customer in our cloud.

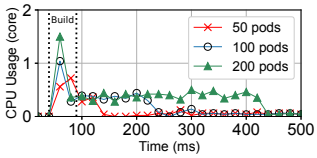


Figure 4: Controller CPU usage and pod update time.

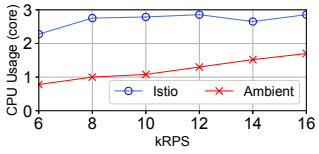


Figure 5: CPU usage of Istio and Ambient.

Table 1: Resource usage of Istio in production.

Cluster size		*Resource usage of sidecar*	
Node	Pod	CPU	Memory
500	15k	1500core, 10%	5000GB, 10%
200	8k	1000core, 8%	1200GB, 5%
100	1k	32core, 4%	150GB, 5%
60	2k	400core, 10%	300GB, 6%
60	400	150core, 30%	300GB, 25%

sidecars consume 1500 CPU cores (10% of the total) and 5000GB memory (10% of the total). In extreme cases, due to the rich functions provided by the sidecar, its CPU and memory usage grow even higher than that of the app (3x for CPU and 5.54x for memory). This has led to customer complaints, as the purchased pod resources are not fully utilized for running their apps. Moreover, we also find that achieving optimal latency requires resource over-provisioning for the sidecar. For example, if the CPU utilization exceeds 45%, the latency doubles, and if the utilization exceeds 75%, the latency experiences significant spikes (100x~1000x), as shown in Fig. 2.

High control plane orchestration overhead. With the growing popularity of service mesh, more customers are deploying microservices with it [8]. As shown in Fig. 3, from 2020 to 2022, the number of sidecars for a major customer nearly doubles. This has significantly raised both the cost and the frequency of individual updates, as the control plane needs to manage all sidecars and any sidecar configuration change triggers a global pod update.

Southbound bandwidth overhead. A sidecar needs many configurations, such as routing policies and security admissions. However, individually orchestrating service-dependent configurations for each sidecar is time-consuming and error-prone. For instance, the connectivity configuration for a K8s cluster with hundreds of services based on their dependencies is complex, requiring manual orchestration for each sidecar, where any misconfiguration can affect service continuity. As users may modify the service logic or scale their apps on demand, such connectivity configuration occurs from time to time. To reduce complexity, a common practice is to download the same configuration set to all sidecars. This set includes all configurations that may take effect, ensuring that any pod can freely communicate with others if needed.

However, pushing full configurations to all pods at each update greatly increases southbound bandwidth overhead, because any update to a sidecar needs to be pushed to all other sidecars, even if they are irrelevant [16]. Assuming there are N pods, the size of the full configurations for each pod's sidecar is $O(N)$, containing routing and security rules for communicating with all other pods. To update a policy, the total southbound data transmission is $O(N^2)$ as pushing the full configurations to all pods is needed. Such overhead grows rapidly as the K8s cluster expands (while incremental update would be preferable, Istio currently lacks good support for it). In the past, we addressed this by dividing a large cluster into smaller ones (*e.g.*, hundreds of clusters for a customer). However, cross-cluster communication is unavoidable, causing synchronization issues.

Table 2: Configuration update frequency by cluster.

Cluster size		*Configuration update frequency (times/min)*
Node	Pod	
3 ~ 10	100 ~ 500	1 ~ 5
30 ~ 60	700 ~ 1100	10 ~ 20
100 ~ 300	1500 ~ 3000	40 ~ 70

In cross-region or cross-cloud deployment of K8s clusters (*e.g.*, for on-premises deployment or disaster recovery), high southbound bandwidth overhead could cause configuration delays or even losses. Since cross-region/cloud communication is expensive due to the requirement for VPNs or dedicated lines, most customers choose not to invest in higher bandwidth. Hence, when managing cross-region/cloud clusters, updates issued by the controller to the sidecars in remote locations may saturate cross-region/cloud bandwidth, causing packet delays or losses. In a real case, a customer deployed a controller hosted on our public cloud to manage their K8s cluster in their on-premises IDC. The initial cross-region VPN bandwidth purchased was 100Mbps. However, due to the large scale of the cluster (with thousands of pods), the peak update rate reached 120Mbps. To eliminate the risk of configuration failures, the customer incurred additional expenses to upgrade the VPN bandwidth to 1Gbps.

Controller CPU usage and pod update time. The sidecar configuration process can be divided into two steps: *building* and then *pushing* the updates. As each sidecar requires the full configurations associated with all pods, the controller CPU usage for building the full configurations is proportional to the cluster size, as shown in Fig. 4. After that, the updates need to be pushed to all sidecars. According to our measurement in Fig. 4, the pushing is I/O-intensive rather than CPU-intensive, as the CPU usage for pushing is not significantly affected by the cluster size, while the update completion (until CPU usage reaches 0) takes much longer for larger clusters.

Update frequency. Table 2 shows the configuration update frequency of different clusters in our cloud. The update frequency increases with the cluster size because larger clusters tend to host a greater number of services. The cumulative update rate of these services leads to an overall increase in the update frequency as the cluster size grows. This further exacerbates the control plane overhead.

2.2 Ambient and its Open Issues

In Sep 2022, Ambient Mesh [38] proposed a novel service mesh architecture that eliminates per-pod sidecars. Specifically, it splits the service mesh functions into two layers with different resource-sharing strategies: a per-node proxy for L4 tasks and a service-level proxy for L7 tasks shared by all pods of that service. Ambient addresses the problems of Istio: (1) Ambient no longer intrudes into user pods, avoiding issues due to the shared lifecycle between the sidecar and app. (2) Ambient avoids resource overuse and idleness associated with reserving a dedicated sidecar for each pod. (3) By offering an optional L7 layer, Ambient reduces the latency of traffic that does not require complex L7 processing. Besides, in Ambient, even traffic requires L7 processing, it only needs to pass through

Table 3: Proportion of users enabling L7 features by region.

	L7	*L7 routing*	*L7 security*
Region1	95%	95%	29%
Region2	93%	93%	33%
Region3	90%	86%	27%
Region4	80%	72%	40%
Region5	88%	80%	53%

the intermediate L7 proxy once. (4) Ambient reduces the number of proxies from $O(pod)$ to $O(node + service)$, lessening control plane overhead. However, Ambient still has open issues as follows.

Most users require L7 processing rules. Ambient believes that reserving a full-featured sidecar for users who do not need L7 functions is overkill [9]. Accordingly, it chooses to retain the commonly used L4 features (*e.g.*, L4 load balancing, zero-trust network) in a per-node proxy and offers L7 functionality as an option. However, based on our observation, the majority of our customers (80%~95%) configure L7 rules in their service mesh. Specifically, the L7 routing policy is the most common configuration (72%~95%), as shown in Table 3. Most customers use L7 routing policies to establish specific packet processing routes based on URLs, HTTP headers, and message content. The operational data reveals that users do want L7 features, but find that the resources consumed by L7 features are high. Given the costs associated with L7, some users choose to stick with L4 only. We believe that the key to the widespread adoption of service mesh lies in reducing the overhead of L7 processing.

Potential risks due to incomplete decoupling. As service mesh liberates apps from infra management, in large enterprises, the apps and the mesh are usually managed by different teams. Due to incomplete decoupling in Ambient, the L4 and L7 proxies (managed by the infra team) are still located in the same cluster of user apps (managed by the service team), and in some cases, even on the same node. The coexistence of proxies and apps may cause resource contention. Besides, since the proxies and apps are in the same cluster without isolation, they share the same controller to issue updates, which incurs the risk of misconfiguration. To avoid misconfiguration, some users even prefer entrusting their app updates to us.

Substantial room for performance improvement. Compared with Istio, Ambient achieves lower end-to-end latency by reducing the L7 proxy processing times from twice to once with an intermediate L7 proxy. However, we believe there is still room for further optimization in the end-to-end processing (as shown in Fig. 10). For example, we can optimize the way traffic is redirected or conduct hardware acceleration for complex L7 processing.

Intrusion remains with insufficient sharing. As the L4 and L7 proxies still reside within the K8s cluster, they continue to consume user resources purchased for their apps. Although Ambient has reduced the overall resource consumption compared to Istio through traffic aggregation at shared proxies, the resource-sharing efficiency is not as high as expected (as shown in Fig. 5). Since Ambient's L7 proxy is shared across different pods belonging to the same service, they inevitably experience synchronized peak and valley workloads, resulting in a reduced peak-shaving effect.

Control plane overhead remains significant. With resource sharing, Ambient reduces the number of proxies from $O(pod)$ to $O(node+service)$. Based on production data, there is approximately a 2:1 ratio between pods and services, and a 15:1 ratio between pods and nodes. Accordingly, we can estimate that Ambient needs to configure approximately 43% fewer proxies compared to Istio. The number of proxies can be further reduced with more radical sharing strategies, *e.g.*, having all services share one proxy.

2.3 Design Goals

Non-intrusive service mesh. The service mesh infra should be as fully decoupled as possible from the user service logic to avoid the risks due to intrusion, such as undesired service disruption.

Minimal performance overhead. As the service mesh operates within the end-to-end path, we should minimize its performance overhead to prevent it from becoming the choke point.

Less resource consumption. Whether the service mesh is injected into the user pod, the user cluster, or hosted by a third party (*e.g.*, a cloud provider), the cost ultimately falls on the user. To lower its infra costs, its resource consumption should be reduced.

Control plane overhead mitigation. As the control plane overhead increases proportionally with the size of a K8s cluster, for K8s production deployment at scale, it becomes crucial to mitigate such non-scalable service mesh orchestration overhead.

3 CANAL MESH ARCHITECTURE

3.1 Design Principles

Remote proxy deployment. Istio and Ambient have more or less intertwined the service mesh proxies with user apps, leading to many issues. To this end, we decide to adopt an aggressive decoupling strategy by deploying these proxies remotely (outside the user's K8s cluster) to achieve complete separation from user apps. Similar to Ambient, by placing the proxies farther away, we can reduce detour routing of traffic to improve performance. What distinguishes us from Ambient is that we move the proxies entirely outside the user cluster. This helps mitigate risks such as controller misconfiguration with physical isolation. It also ensures that the resources purchased by users exclusively serve their apps.

Consolidation of services. Istio allocates one sidecar for each pod in the user cluster. To reduce the control plane overhead of these per-pod sidecars, Ambient allocates L4 proxies for each node and L7 proxies for each service with a reduced proxy number. To further minimize the service mesh orchestration overhead, we aim to consolidate the per-node and per-service proxies into one. Since the consolidated proxy can be shared by all pods, it enables efficient peak shaving to reduce overall resource allocation, eliminating the need to reserve peak resources for individual proxies.

Multi-tenancy. After being deployed remotely, the consolidated service mesh proxy no longer consumes resources within the user's purchased K8s cluster. However, it still needs to be separately acquired by the user or hosted by a third-party cloud provider. As a major cloud provider, we naturally explore the idea of sharing the proxy among multiple tenants, similar to the various cloud products (*e.g.*, SLB [36], NAT [33], cloud gateway [59, 60]) we have previously offered, leveraging economies of scale [64] to reduce tenant expenses. Besides, as we decouple the service mesh from the K8s cluster, we are no longer constrained by the incomplete multi-tenancy capabilities of K8s (which is originally designed for single-tenant usage and provides very limited soft multi-tenancy support with resource partitioning rather than resource sharing [1, 14]). Instead, we can leverage years of experiences in public cloud management to build an efficient multi-tenant service mesh.

3.2 Strawman Architecture and Issues

Based on the above design principles, we propose a strawman architecture for a multi-tenant consolidated service mesh. In this

architecture, each tenant's apps still run on a K8s cluster (either located in the tenant's on-premises IDC or hosted by a cloud provider). Their service mesh proxies are consolidated and deployed remotely in the public cloud, serving apps for all tenants. However, based on our experiences, such a strawman architecture still faces numerous deployment issues (Issue #1 stems from remote deployment, while #2, #3, #4 arise from consolidation and multi-tenancy).

Issue #1: functional equivalence after remote deployment. Service mesh generally offers three major features: traffic control, zero-trust network, and observability. However, not all features can be guaranteed with functional equivalence when deployed remotely. For example, when the proxy is moved remotely into the public cloud, the traffic from the user apps to the proxy will inevitably be exposed to the public cloud environment (which is not trusted by the user) instead of being confined within the user cluster, no longer meeting the requirement of zero-trust network. Another example is that, without the log collected on the user cluster, relying solely on the log collected by the remote proxy will be insufficient to precisely pinpoint faults end-to-end. Even if our cloud-based solution offers superior performance and cost-effectiveness, users won't embrace it if essential features are compromised.

Issue #2: isolation between tenants/services. Consolidation and multi-tenancy lead to a single cloud-based proxy simultaneously serving a large number of tenants and their services. This significantly increases the blast radius in case of proxy failure, affecting all tenants and services. Furthermore, the shared proxy among all tenants and services leads to resource contention (the noisy neighbor problem) and unexpected SLA issues. Moreover, since header address spaces within different tenants' VPCs may overlap in the public cloud, a multi-tenant service mesh proxy requires the ability to differentiate overlapping header addresses across tenants.

Issue #3: stateful resource scaling for concurrent services. Horizontal scaling is a key capability of cloud to handle dynamic workloads. In the public cloud, the scaling of stateless devices (*e.g.*, cloud gateways for VPC routing [60]) only requires spawning new backends (VMs or containers), and redirecting the growing traffic to the new backends. In comparison, the service mesh proxy works at L4/L7 and is not stateless. During scaling, it needs to maintain session states to ensure uninterrupted user services. Specifically, we need to spawn new backends or find existing available backends, and then migrate newly established sessions to the new or available backends via the configuration of LBs. In Istio and Ambient, as the sidecar is allocated per-pod, while the L7 proxy is aggregated at the service level, each sidecar/L7 proxy only handles a single service. This makes their scaling strategy rather simple, only requiring migration of the new sessions for the single service to the newly spawned sidecar/proxy. However, our consolidated proxy needs to handle configurations for tens of thousands of concurrent tenant services, each with different groups of backends for fault isolation. When we observe workload spikes on some backends, a straightforward solution is to iteratively find suitable backends for each service on the overloaded backends to hold the spill of traffic. However, scaling all services blindly is inefficient, leading to lots of unnecessary operations and delayed reduction in proxy load.

Issue #4: CapEx reduction to save infra costs. For elasticity, our consolidated service mesh proxy is built upon backend VMs, which can be elastically created and migrated on physical servers according to the changing workloads. Therefore, our proxy needs LBs to distribute traffic evenly across these backends. Specifically, considering that the backends for different services do not entirely overlap for fault isolation, we need to assign one LB for each service, with each LB carrying specific rules. Furthermore, for service resilience, we deploy services across multiple AZs. To reduce latency, we prefer to deploy LBs locally in each AZ to serve backend VMs in that AZ. However, these LBs incur considerable infra cost.

In addition to LBs, sessions also consume substantial resources [65]. After consolidation, our proxy needs to manage a significant number of sessions, which are distributed across its backend VMs. However, session resources for each VM are sourced from the underlying server, constrained by the limited memory of attached SmartNICs [43]. Once the proxy is using all available sessions, scaling its session capacity requires allocating more VMs, incurring additional infra cost. Additionally, scaling out the proxy for available sessions reduces resource efficiency, as the proxy's backend VMs typically use only 20% of CPU when using 90% of available sessions.

3.3 Refined Architecture

We propose solutions to address the above issues and refine the strawman architecture accordingly.

Solution #1: functional equivalence via on-node proxy. To ensure functional equivalence after remote proxy deployment, we add a lightweight proxy on each user node responsible for minimal security and observability features. Other features remain deployed remotely. In addition, we use eBPF instead of iptables to improve the traffic redirection performance of the proxy, and we reduce the resource consumption of the proxy by offloading compute-intensive asymmetric crypto for zero-trust network to dedicated hardware.

Solution #2: multi-pronged approach for high availability. To ensure the high availability of the consolidated service mesh proxy, we deploy multiple backends per service within a single AZ. Service unavailability occurs only when all its backends are down. Furthermore, for scenarios where all physical devices within a single AZ may go down due to power outages or other reasons, we also deploy backends across different AZs for each service to further improve the availability of the service mesh proxy.

To avoid the issue of a large blast radius in case of proxy failure, we use shuffle sharding [39] to minimize the overlap of backends for different services. It ensures that when all backends of a service are unavailable, other services still have healthy backends to use.

In worst-case scenarios, healthy backends of a service may be overwhelmed by transient migration of traffic from other failed backends. To tackle this, we build a multi-indicator monitoring system and a set of rapid response mechanisms that can quickly intervene before the service suffers. For normal traffic spikes, we scale out the backends of the service to ease contention, while in abnormal situations (*e.g.*, malicious attacks), we isolate the affected service in a sandbox to prevent impact on other running services.

Solution #3: precise scaling with root cause analysis. To improve inefficient blind resource scaling, we collect utilization data from each proxy backend and analyze it on the controller. This enables real-time monitoring of load fluctuations for the top services on each backend, allowing us to pinpoint the specific services causing the backend load increase. Through precise scaling, we can alleviate backend overload rapidly with minimal scaling operations.

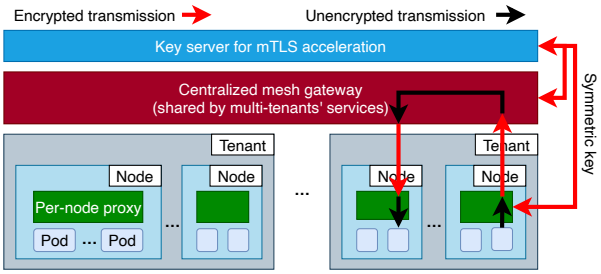


Figure 6: Canal Mesh architecture.

Solution #4: LB disaggregation and session aggregation. To reduce the cost associated with LBs, we break down the LB into its essential functions (*i.e.*, load distribution and session maintenance), and integrate them separately into the existing infra. Such a cloud infra reuse strategy allows us to attain functions that traditionally require deploying dedicated LBs, with almost no additional cost.

To address the imbalance of session and CPU utilization, we propose session aggregation via tunneling. By aggregating a large number of sessions into a few tunnels, the session state maintenance burden on memory-constrained SmartNICs is alleviated.

In summary, the refined Canal Mesh architecture includes a *centralized mesh gateway*, multiple *on-node proxies*, and a *key server*, as shown in Fig. 6. The mesh gateway handles traffic control for all tenants/services. The on-node proxies and mesh gateway work jointly to achieve zero-trust network and end-to-end observability. The multi-tenant shared key server manages asymmetric crypto offloading for both the on-node proxies and the mesh gateway.

4 SYSTEM IMPLEMENTATION

4.1 On-Node Proxy

4.1.1 Functional equivalence analysis. As mentioned, Canal introduces minimal-feature on-node proxies for functional equivalence. So, what features have to be kept in the on-node proxy? We assess the remote deployability of three major service mesh features as follows, by analyzing whether the input and processing logic can remain consistent with the previous sidecar-based architecture.

Traffic control. The traffic control of the service mesh includes route control, load balancing, A/B testing, and canary release [6]. Their inputs are carried by the packets (such as DIP, URL, cookies) and processed by the forwarding tables in service mesh proxies configured by the controller (such as stateless FIB, stateful session table). Since the inputs can be carried by packets to the remote proxy and the forwarding tables can also be configured remotely, traffic control equivalent to a sidecar can be deployed remotely.

Zero-trust network. The zero-trust network of the service mesh includes encryption, authentication, and authorization. Due to distrust in the cloud providers' networks, the data must be encrypted before leaving the user node, preventing remote encryption deployment. Authentication involves issuing certificates to each pod. These certificates, used in the mTLS handshake to verify pod identity, contain sensitive information, thereby making remote deployment unsuitable. However, authorization is similar to traffic control, with input and processing logic being information carried by packets and traffic admission rules, allowing for remote deployment.

Observability. The observability of the service mesh includes metrics collection, logging, and tracing. They depend on instrumentation at critical points in the traffic path for data tagging, recording, and analysis. Deploying observability remotely will impair the end-to-end data collection capabilities. In fact, it is better to deploy observability on all critical nodes.

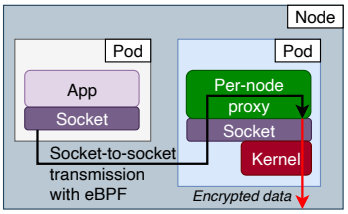


Figure 7: eBPF-based traffic redirection.

In summary, to reduce intrusion into the user cluster while preserving functional equivalence, we offload most of the service mesh functions to the remote mesh gateway and maintain lightweight on-node proxies. Theoretically, the on-node proxies should handle part of zero-trust network (*i.e.*, encryption and authentication) and full observability to supplement the functionality missing due to remote proxy deployment. In practice, similar to other decoupling solutions like Ambient, we downgrade the observability of the on-node proxies to L4, while offering rich L7 observability on the remote mesh gateway. This design choice sacrifices some end-to-end L7 observability but significantly reduces overhead at the user cluster. As our on-node proxy is shared by all pods on the node, the per-user resource occupation and control plane overhead are amortized. Since our on-node proxy handles minimal features without traffic control, it requires infrequent updates from the controller, thereby further reducing user intrusiveness and control plane overhead compared to Istio and Ambient. Finally, we use eBPF and a dedicated key server for its performance acceleration.

4.1.2 eBPF-based traffic redirection. As the traffic originating from the user app still needs to be redirected to the on-node proxy, using iptables-based redirection requires two additional passes through the kernel stack (details in Fig. 21). To reduce the kernel processing times for better performance, we adopt eBPF-based redirection with socket-to-socket transmission (similar to SPRIGHT [61]), as shown in Fig. 7. However, since eBPF bypasses the kernel, it also loses some kernel stack features, such as security checks and TCP optimization. We use a case to show how the lack of kernel stack features affects eBPF processing and how we address this issue.

When forwarding small packets, kernel bypass with eBPF exhibits unexpectedly lower throughput and consumes more resources compared to iptables. Our debugging revealed that eBPF lacks a small packet aggregation mechanism, resulting in increased resource consumption due to a high frequency of context switches when processing small packets (as shown in Fig. 22 in the Appendix). In contrast, the kernel protocol stack defaults to enabling the Nagle Algorithm [57], which aggregates small packets into larger ones, reducing context switch frequency. In response, we implement Nagle with eBPF, aggregating small packets before eBPF redirection.

4.1.3 Dedicated key server for remote mTLS acceleration.

Issues of local mTLS acceleration. mTLS involves two steps: asymmetric crypto and symmetric crypto. Asymmetric crypto occurs only during the negotiation phase of the transmission. After its completion, the computed symmetric key with the private key is used for all subsequent traffic crypto (*i.e.*, symmetric crypto). Although the frequency of asymmetric crypto is lower, its resource

consumption is much higher than symmetric crypto [44]. For performance improvement, Intel has introduced hardware-based (QAT [17]) and software-based (AVX-512 [24]) accelerations for asymmetric crypto. To minimize the intrusiveness of the on-node proxy on user resources, we opted to offload asymmetric crypto to the local CPU. However, we encountered issues in production deployment.

Rigid hardware requirement and high cost. As new hardware features, not all existing Intel CPUs support QAT/AVX-512. In our cloud, only a small number of VM models (<10%) based on new Intel CPUs support acceleration for asymmetric crypto. Furthermore, these VMs are more expensive, increasing user expenses. For example, with all other configurations (*e.g.*, CPU, memory, bandwidth) being equal, the g7 VM model that additionally supports QAT/AVX-512 is around 30% more expensive than the old g6 model [13].

Unexpected performance degradation. We observed that when the number of new sessions is low, local mTLS acceleration may result in increased latency and degraded throughput. Through analysis, we identified the cause as the batch processing architecture adopted by the acceleration solution. When the number of incoming sessions is less than the capacity processed in a single batch, these sessions have to wait, leading to additional latency. Specific test results and analysis are presented in the Appendix (see Fig. 25).

Remote mTLS acceleration. To address the above issues, we offload asymmetric crypto acceleration to a remote key server by changing the local function calls to remote procedure calls. After finishing asymmetric crypto, the key server returns the symmetric key to the requesters (both the on-node proxy and the gateway for mTLS), as shown in Fig. 6. Subsequently, data transfer between the on-node proxy and the gateway uses the symmetric key for local crypto, ensuring that information remains secure from unauthorized access. Still, we remotely offload only asymmetric crypto due to its high computational overhead and low frequency. Symmetric crypto, being frequent and simpler, is kept local to avoid increased latency and limited performance gain from remote acceleration. This ensures that traffic is routed remotely only in the negotiation phase, maximizing benefits with minimal overhead.

Remote mTLS acceleration offers many advantages. First, it does not rely on the specific VM model at the user end, eliminating the need for users to purchase newer and more expensive VMs, thus reducing their expenses. In addition, the shared key server increases resource utilization, further lowering user expenses. Second, as the key server simultaneously serves a massive number of services, the instantaneous arrival of new sessions to the key server is much higher than the processing quota for a single batch, preventing performance degradation from processing bubbles.

Maintaining the security of the key server is critical, as if it were compromised, all the secrets of all the tenants (*i.e.*, the private keys) would be exposed to the attacker. We store private keys in memory rather than on the hard drive, ensuring they are flushed after a restart. This prevents user private keys from being compromised if the server is physically stolen. Additionally, we encrypt the stored private keys and only decrypt them when a verified requester initiates an asymmetric crypto request to the key server. The key server conducts asymmetric crypto and returns the derived symmetric key to the requester on-the-fly, without keeping the intermediate plaintext private key obtained during decryption. Through these measures, we significantly enhance the security of private keys.

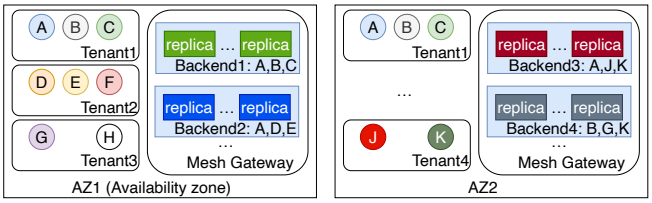


Figure 8: Hierarchical failure recovery.

To prevent the attacker from stealing the symmetric key obtained through asymmetric crypto, communication between the requesters and the remote key server also needs to be encrypted. However, performing a new TLS handshake for each request introduces additional latency overhead. Therefore, we choose to use a pre-established shared channel to encrypt all communication between each requester and the key server.

To reduce the high latency caused by routing to key servers in other AZs, we should deploy local key servers within each AZ to serve local on-node proxies and gateway backends. However, in our cloud, there might be some AZs (less than 5%) that lack QAT/AVX-512 CPUs for asymmetric crypto acceleration. In such cases, we will consider deployment compatibility and fallback to the traditional software-based asymmetric crypto on old CPU models.

4.2 Cloud Infra for Remote Mesh Gateway

Multi-tenant service mesh. In a single-tenant service mesh like Istio, packet header fields (*e.g.*, SIP, DIP, URL) distinguish traffic from different services. However, in a multi-tenant public cloud, where packet header address spaces may overlap across VPCs, using the virtual addresses within VPCs is insufficient for service differentiation across tenants. Generally, VXLAN [56] IDs are used to differentiate tenant traffic. However, since our mesh gateway is deployed based on VMs for elasticity above the vSwitch, the outer VXLAN header is removed at the vSwitch before packets reach the VMs. This makes it impossible for the mesh gateway to differentiate tenants based on VXLAN IDs within VMs. To address this, before striping off the VXLAN header at the vSwitch, we map it to a globally unique service ID and attach it to the inner header, enabling the gateway to distinguish tenant services inside VMs.

Hierarchical failure recovery. To handle the substantial amount of processing workloads from consolidation and multi-tenancy, the mesh gateway is implemented by multiple backends behind a virtual IP. To further enhance the availability of each backend, a backend is composed of multiple replicas, sharing the same set of configurations (as shown in Fig. 8). In our implementation, a replica is a VM while a backend is a group of VMs.

In the event of a replica failure, active sessions on that replica will experience a very short disruption. However, they will be rerouted and reconstructed on other replicas soon, ensuring that the user experience will not be significantly affected.

If all replicas of a backend experience a failure, to ensure service availability, a service's configuration will be installed to multiple backends in the same AZ. In Fig. 8, for service A of tenant1 in AZ1, its configuration is deployed to both Backend1 and Backend2. Only when both of them fail would it lead to a disruption of service A.

To mitigate the risk of a complete failure of all backends within an AZ due to sudden events like power outages, we extend the deployment of a service's configuration to multiple AZs, which

further enhances the service availability. For example, in Fig. 8, Backend3 in AZ2 also carries the configuration for service A.

We have customized the DNS resolution logic to ensure requests are prioritized to be resolved to available backends within the local AZ for optimal latency. Only if all backends in the local AZ are unavailable will the requests be resolved to other AZs.

Inter-service isolation via shuffle sharding. To achieve inter-service failure isolation, *i.e.*, ensuring that the failure of all backends of a service does not lead to a complete disruption of other services, we adopt shuffle sharding [39]. This guarantees that each service has a unique combination of backends. For instance, in Fig. 8, if all backends of service A (Backend1, 2, 3) fail due to a "query of death" [18], service B still has Backend4 alive to handle its requests.

Anomaly detection-triggered rapid intervention. While mechanisms like shuffle sharding ensure that a service always has healthy backends available, traffic migrated from failed backends may temporarily overload healthy ones, causing cascading failures. To handle this, we propose a set of anomaly detection and response mechanisms to intervene promptly before large-scale failures occur.

Backend-level alert. When a backend's water level (*e.g.*, CPU utilization) exceeds a threshold, an alert will be triggered. This is usually due to a traffic increase from some services. To avoid affecting other services, we must keep the backend water level below a safety threshold. While rate limiting can address this, it affects user experience. Instead, our system determines whether the increased usage is due to an expensive query, an increase in overall workload, a DDoS attack, or some undetermined cause. If it is a normal situation, we scale infra capacity on demand to gradually reduce the water level. Otherwise, we migrate anomalous traffic to a sandbox quickly, preventing it from continuously affecting other tenants.

Service-level alert. Some users prefer automatic scaling and they pay based on actual usage when purchasing service mesh resources. To ensure sufficient resources for these users, we continuously monitor resource utilization and performance metrics for each of their services. If any resources are about to be depleted or the performance becomes worse, we automatically scale their resources to prevent user experience degradation. Additionally, we will also check for service-level anomalies, such as unusual frequent scaling operations, and similarly, migrate the affected services to a sandbox.

Tenant-level alert. If the user's K8s cluster is also hosted on our cloud, we monitor its resource usage in real time. When we observe a sudden surge in resource utilization approaching 100%, we will discuss with the user about disabling the service mesh's auto-scaling capability and conducting rate limiting on the mesh gateway to protect the user cluster from being flooded by inbound traffic. Once the user cluster completes its own scaling process, we will remove the throttling on the gateway and restore its auto-scaling.

Traffic pattern monitoring. When traffic patterns among services sharing the same backend exhibit phase synchronization in peaks and valleys, we consider scattering these services onto different backends as much as possible; otherwise, it may lead to a sudden surge in CPU usage, which will affect user SLA. To achieve this, we periodically sample and monitor the traffic patterns of the top services in each backend and employ a transparent traffic migration mechanism to handle the phase synchronization situation.

We explain the workflows of the above rapid intervention and traffic pattern monitoring in more detail in §6.2 and §6.3.

4.3 Precise Cloud Resource Scaling

In most cases, we address user incremental resource demands through scaling out. The aforementioned throttling and migration to the sandbox are reserved only for a few exceptional cases.

Root cause analysis algorithms. To avoid inefficient blind scaling, we use root cause analysis to pinpoint the services with rapid traffic growth and perform precise scaling for these services. Specifically, when the water level of a particular backend exceeds a threshold, we sample the RPS for requests and identify the top services on this backend. Then, we observe whether the traffic trends of those top services align with the backend's water level trends. If the target service is pinpointed, a new backend is scaled out for it.

Furthermore, for high availability, a service usually has multiple backends. Due to load balancing, the traffic growth of a service may lead to a synchronous rise in water levels across multiple backends hosting that service. If such a situation occurs, taking the intersection of services carried by these backends will likely identify the root cause service. However, due to the varying service distributions on each backend, an increase in traffic for one service does not always lead to a simultaneous rise in load across all its backends. Consequently, the algorithm to intersect across multiple backends does not always prove effective. In practice, we will run this algorithm only once at the start to speculate. If it fails to help us quickly identify the root cause, we will revert to the basic algorithm.

Resource scaling strategies. When a service needs backend scaling, we adhere to the following strategies:

(i) Reuse. If there are still backends within the AZ with low water levels (*e.g.*, < 20%), we directly extend the service to these backends.

(ii) New. If all backends within the AZ have high water levels, we deploy a new backend and extend the service to it.

With these strategies, we prioritize utilizing existing backends, enabling faster resource scaling and lower infra costs.

4.4 Infrastructure Cost Reduction

LB disaggregation. For reducing the CapEx of maintaining per-service LBs, we reuse the existing infra to achieve equivalent functions of the original LBs. Generally, an LB has two functions: (i) load distribution to multiple backend VMs (*i.e.*, replicas); (ii) session consistency maintenance. To replace the LBs, we reuse the ECMP [53] ability of the router in front of the original LBs for load distribution. If there is no change in the LBs' replica list, ECMP ensures that the same flow is always hashed to the same replica with session consistency. However, when there is a change in the replica list, the hash base is modified, and session consistency is broken. To address this, we build a system inspired by Beamer [58], using a redirector at each replica to verify the router's hashing decision and conduct chain-based redirection. Specifically, a fixed-size bucket table in the redirector manages flow redirection, ensuring new flows go to new replicas while existing flows continue to their original destinations. This achieves session consistency during replica changes. As it is similar to Beamer, we provide a case study in the Appendix (Fig. 26).

To adapt Beamer to our cloud, we make the following modifications. (i) We increase the length of the replica chain in the bucket table from the original 2 to a larger value to better support multiple scale-out/scale-in events in a short period (*e.g.*, consecutive crashes of multiple replicas due to the query of death). (ii) We maintain per-service bucket table to record bucket-to-replica mapping, indexed by the service ID. (iii) We use eBPF to accelerate the redirector.

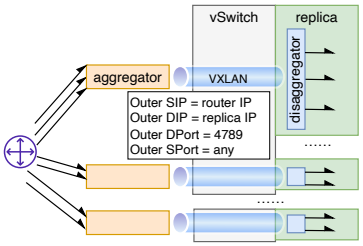


Figure 9: Session aggregation and disaggregation.

LB disaggregation brings many advantages. First, the blast radius is much smaller as one LB is decomposed into multiple redirectors. Second, the processing latency is lower and more stable for two reasons: (i) sometimes traffic will be directed to LBs across AZs (*e.g.*, due to the unavailability of LBs in local AZ), leading to increased latency; (ii) compared to LB disaggregation, dedicated LBs add an extra hop in the overlay, which may correspond to multiple hops in the underlay based on our implementation, thus increasing latency. With LB disaggregation, the end-to-end latency is reduced from 3ms~4.2ms to 1.4ms~2.1ms. Third, the infra cost is significantly reduced as the VMs for dedicated LBs are no longer needed while the redirectors can reuse the VMs for replicas due to their lightweight nature (the redirection frequency is low, and its processing cost is 12x~15x smaller than the L7 processing cost of the replica).

Session aggregation. Fig. 9 shows how session aggregation reduces the session consumption at the underlying server. Specifically, we encapsulate a large number of sessions into a few tunnels at the aggregator based on VXLAN. The outer DIP is set to replica IP and the outer SIP is set to router IP. When the packets reach the replica, we use a disaggregator to remove the outer VXLAN header and let the replica process the original sessions. For implementation, the aggregator can be integrated into the router if the router is implemented with programmable chips [46], and the disaggregator can be implemented on the replica (placed before the redirector).

Since a replica typically occupies multiple CPU cores, it is preferable to balance tunnel loads across these cores. To achieve this, we aggregate sessions into multiple tunnels by setting different outer SPorts. The vSwitch then hashes these tunnels to multiple cores. By setting an appropriate number of tunnels (*e.g.*, 10 times the number of cores), we can distribute the tunnel loads evenly across all cores.

5 EVALUATION

5.1 Experimental Settings

Small-scale testbed. As Ambient is still under rapid iteration, we have not deployed it widely in production. To fairly compare Istio, Ambient, and Canal, we set up a small-scale testbed to observe how Canal addresses the issues in §2 (the versions of Istio and Ambient in evaluation are [15] and [7], respectively). The testbed has an Intel Xeon 8269CY CPU with 8 cores 16 threads, and 64GB memory. The K8s resources include two worker nodes (for apps) and one master node (for the controller). Each worker node has 15 pods. The number of hosted services is 3, thus there are 3 L7 proxies for Ambient. The results in §5.2, §5.3 and §5.4 are from the testbed.

Production deployment. Canal has been deployed in Alibaba Cloud since 2023 and is made available to beta users. The results in §5.5 and §5.6 are collected from production cloud regions. The mesh gateway in these cloud regions is made up of thousands of VMs, handling an average of millions of RPS.

Figure 10: Latency under light workloads.

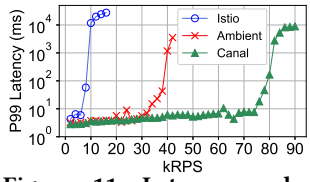


Figure 11: Latency under changing workloads.

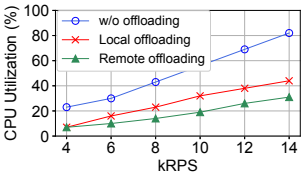


Figure 12: CPU usage saving with crypto offloading.

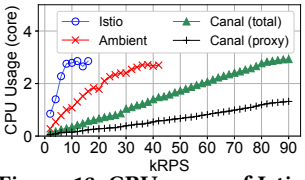


Figure 13: CPU usage of Istio, Ambient and Canal.

5.2 Performance

5.2.1 Performance improvement with key server and eBPF. With crypto offloading to the key server, the throughput is improved by 1.6x~1.8x (Fig. 27), while the latency is reduced by 53%~60% (Fig. 28). With eBPF-based redirection, the throughput is improved by 1.3x~2.3x (Fig. 29), while the latency is reduced by 55%~66% (Fig. 30). Detailed experimental results are shown in the Appendix.

5.2.2 Comparison with Istio and Ambient. We measure the latency of three service mesh solutions under light workloads, then adjust the workloads to evaluate their throughput with acceptable latency.

Latency under light workloads. Fig. 10 shows the end-to-end latency of different solutions under light workloads. Specifically, we use 1 thread and 1 connection to send 1 request per second and repeat this 100 times. "No service mesh" is the latency when the client and server are directly connected without L7 processing as the baseline. The latency of Canal is the closest to the "No service mesh" baseline, and is 1.7x and 1.3x lower than that of Istio and Ambient, demonstrating that Canal can provide L7 functions to users with minimal overhead. Istio has the highest latency because it requires traversal through two L7 processing nodes, while Ambient and Canal only require traversal through one L7 node.

Latency under changing workloads. Fig. 11 shows the P99 latency under different workloads. In this experiment, we use 1 thread and 100 connections to send requests at varying RPS. When the workloads are within the processing capability, the latency remains stable. Once the CPUs are saturated, the latency increases sharply. Canal lowers the CPU usage and thus increases the throughput (*i.e.*, the maximum RPS before latency spikes), which is 12.3x and 2.3x higher than that of Istio and Ambient, respectively.

5.3 Resource Consumption

Resource usage saving with key server. Fig. 12 shows the reduction in CPU utilization of the on-node proxy achieved through local CPU offloading and remote key server offloading, both using Intel AVX-512 for crypto acceleration. The local and remote offloading can reduce CPU utilization by 43%~70% and 62%~70%, respectively, leaving more resources to the user apps.

Resource usage comparison with Istio and Ambient. Fig. 13 shows the CPU core usage (4 cores in total) under different workloads. Canal (total) and Canal (proxy) represent CPU core usage for Canal with and without the mesh gateway, respectively. For Ambient, we allocate 2 cores for the L4 proxies and 2 cores for the L7

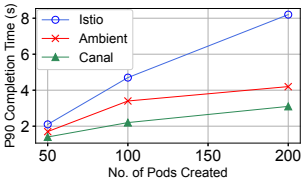


Figure 14: Time for creating multiple pods.

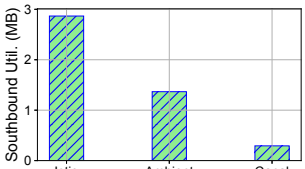
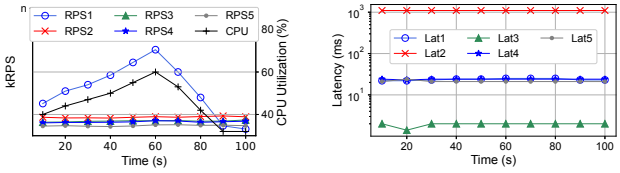


Figure 15: The southbound bandwidth overhead.



(a) RPS (b) Latency

Figure 16: Noisy neighbor isolation in a multi-tenant backend (2023/12/28 20:30:00 - 2023/12/28 20:31:30).

proxies. For Canal, we allocate 2 cores for the on-node proxies and 2 cores for the gateway. As shown in Fig. 13, Canal consumes 12x~19x and 4.6x~7.2x less user CPU resources than Istio and Ambient.

5.4 Control Plane Overhead

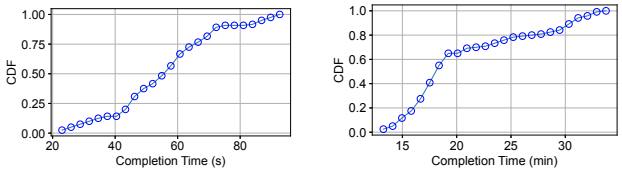
Configuration completion time. Fig. 14 shows the completion time for creating multiple pods in a K8s cluster through an API call. By recording the time taken before successful pings to the pods, we obtain the P90 completion time. The configuration time of Canal is 1.5x~2.1x and 1.2~1.5x lower than Istio and Ambient. For Canal, the majority of configurations only need to be pushed to the centralized mesh gateway. In contrast, Istio requires to configure per-pod sidecars, while Ambient requires to configure L4/L7 proxies.

Southbound bandwidth overhead. Fig. 15 shows the southbound bandwidth occupation during an update of routing policies. Since Canal only needs to configure the mesh gateway, its bandwidth usage is the lowest. Ambient, however, requires the configuration of multiple L7 and L4 proxies, resulting in a higher bandwidth consumption (4.6x). Since Istio needs to configure per-pod sidecars, its bandwidth overhead is the highest, achieving 9.8x that of Canal.

5.5 Performance on Cloud Infra

Noisy neighbor isolation. To demonstrate Canal's service isolation capability, we retrieve a historical case illustrating a typical scenario where a sudden surge in traffic for a service led to an elevated backend load beyond the safety threshold. Subsequently, we monitored the changes over time in RPS and latency for the top services hosted on the same backend, as well as the backend CPU utilization, as shown in Fig. 16. At the 50s, a backend-level CPU utilization alert was triggered. After that, through precise resource scaling (employing *Reuse* for responsiveness), the CPU utilization rapidly decreased below the safety threshold. During the entire process, neither the RPS (Fig. 16(a)) nor the latency (Fig. 16(b)) of other services suffered any degradation. Furthermore, the HTTP error codes for all services continuously remained at 0, indicating that we successfully isolated the "noisy neighbor" from other services.

Resource scaling. Fig. 17 and Table 4 show the time from executing the scaling operation to the reduction of utilization below the safety threshold for both scaling strategies (Reuse and New) discussed in §4.3. In our experiments, the P50 time of Reuse and New is about 55s and 17min. The New takes much longer time due to initialization



(a) Reuse (b) New

Figure 17: CDF of completion time of Reuse and New.

Table 4: Examples of reusing existing backend (Reuse) and creating new backend (New) in a cloud region.

	Reuse	*New*
Traffic increase	2024-01-01 10:00:10	2023-12-20 19:01:45
Exceed threshold	2024-01-01 10:05:24	2023-12-20 19:19:20
Execute Reuse/New	2024-01-01 10:06:48	2023-12-20 19:20:49
Finish Reuse/New	2024-01-01 10:07:11	2023-12-20 19:38:19
Below threshold	2024-01-01 10:08:02	2023-12-20 19:39:21

tasks, such as VM creation, image loading, network setup, and resource registration with Canal's resource pool.

Fig. 18 depicts the daily occurrences of scaling through the Reuse and New operations over a month. Although the New operation takes longer, it is invoked far less frequently than the Reuse. Moreover, to reduce the waiting time of applying New, we often execute it in advance. This is because, it usually takes several minutes to tens of minutes to consume all backend resources, providing sufficient time to invoke New in advance. Additionally, users typically communicate with us before launching large-scale resource scaling.

Shuffle sharding for failure isolation. Fig. 19 shows the effect of shuffle sharding on the backends for top services. The figure illustrates that there is no complete overlap among the backend combinations of services, ensuring that a failure in one service does not lead to a complete outage of others.

High availability for a single service. Furthermore, Fig. 19 also shows each service has multiple backends, which enhances the service-level high availability.

Daily operational data. Fig. 20 shows the trend of RPS and HTTP error codes during Canal's daily operations, including service migration, Canal version update, and resource scaling (*i.e.*, Reuse and New). In Fig. 20, the error codes generally follow the same trend as RPS, and the above operations have not caused any spikes in error codes, suggesting that no faults have occurred during these operations. To minimize the impact of any potential faults, the version update is scheduled during the night. The version update takes about 4 hours as it involves rolling upgrades of machines. The time for other operations is short, ranging from seconds to minutes.

Based on our experiences, most error codes originate from the user side. For example, a user's service might generate a large number of error codes (*e.g.*, traffic exceeding their quota), but if their service is not significantly affected, they may choose to ignore them (*e.g.*, unwilling to pay for a scale-out). In another case, a user intentionally designs programs to return error codes by default. Therefore, as long as there is no sharp increase in error codes, it suggests that our operations have not made any negative impact.

5.6 Deployment Costs

Cost reduction through embedded redirectors. By embedding the redirectors into replicas, we no longer need to maintain separate LBs, reducing the cost of acquiring dedicated cloud resources (*e.g.*, VMs as LB instances) by 32%~48%, according to our measurement of 4 cloud regions deploying redirectors (as shown in Table 5).

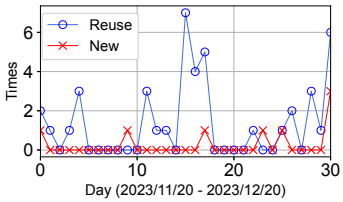


Figure 18: Occurrences of Reuse and New in a cloud region.

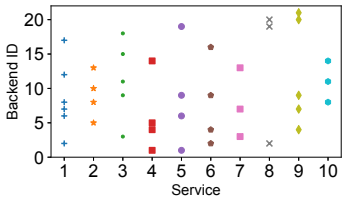


Figure 19: Backend combinations from shuffle sharding.

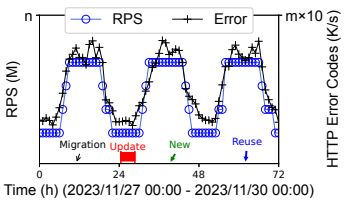


Figure 20: Daily operational data in a cloud region.

Table 5: Cost reduction by redirector and tunneling.

	Redirector	*Tunneling*	*Redirector&Tunneling*
Region1	47.5%	32.2%	64.4%
Region2	45.1%	45.3%	69.9%
Region3	32.1%	33.6%	54.9%
Region4	36.7%	36.5%	59.9%

Cost reduction through tunneling. By aggregating sessions into tunnels after deploying redirectors, cloud resource saving can further reach 55%~70% (see Table 5). Note that although the number of sessions can decrease from hundreds of thousands to just a few, this does not mean a proportional reduction in the number of required VMs. This is because, even when sessions are no longer a bottleneck, we still need to provide an adequate number of VMs to meet other resource requirements, such as CPU and memory.

6 EXPERIENCES

6.1 Excessive Health Check Handling

Health check probes outnumber user app traffic. To monitor the availability of user apps in the K8s cluster, each service mesh proxy needs to perform periodical health checks with probe packets. After deploying Canal in production, we received user complaints about their apps receiving excessive health checks from our mesh gateway. As shown in Table 6, the health check traffic far exceeds the app traffic, reaching up to 515x!

Table 6: Excessive health checks *vs* app traffic.

	Case1	*Case2*	*Case3*	*Case4*	*Case5*
App traffic (RPS)	21	4221	385	496	9224
Health checks (RPS)	10817	52122	12960	22107	19014

After investigation, we found that the centralized gateway causes redundant health checks. Specifically, due to a user's service being configured on multiple backends of the gateway, with each backend containing multiple replicas and each replica having multiple CPU cores, conducting health checks from each core for the user apps associated with the service generates redundant traffic. Besides, in a cluster, apps in a pod may belong to different services, causing health checks for these services to converge on the same pod.

Health check proxy with multi-level aggregation. To reduce the redundancy, at the service level, when the controller detects overlapping apps associated with different services configured on the same gateway backend, we aggregate their health checks. For example, if service A and B are configured on a backend and the apps associated with them are (1, 2, 3) and (3, 4), we apply the aggregated health checks to (1, 2, 3, 4) on that backend. If A and B are configured on different backends, we do not aggregate, as synchronizing health check results between backends incurs communication overhead.

At the CPU core level, we elect one core on behalf of others to send health checks to apps, while the remaining cores inquire about the health check results from this core.

At the replica level, we take a similar approach. However, as the number of replicas for a service can be substantial due to scale-out, to avoid large overhead due to numerous replicas querying the health check results from a single replica, we spawn a dedicated health check proxy to handle health checks at the replica level.

Table 7: Health check reduction by aggregation.

	Base	*Service-*	*Core-*	*Replica-*	*Reduction*
Case1	10817	9344	584	18	99.83%
Case2	52122	46592	3328	104	99.8%
Case3	12960	12960	1620	50	99.61%
Case4	22107	13464	1122	62	99.72%
Case5	19014	18351	1624	49	99.74%

Less health checks than per-pod sidecar. Table 7 shows the step-by-step health check reduction by multi-level aggregation with a minimal decrease of 99.6%. In Istio, each user app receives health checks initiated by all sidecars, whereas in Canal, the health check proxy aggregates the health checks from $O(sidecar)$ to $O(1)$.

6.2 Exception Handling in Production

Exception handling mechanisms. We use migration and throttling to address exceptional cases, such as DDoS attacks.

Migration (for mesh gateway protection). Migration is leveraged to prevent a user's abnormal service growth, which could exhaust resources in our mesh gateway and impact other tenants. As migration is to protect our infra, the specific strategy, *e.g.*, either lossless or lossy migration, is decided entirely by the cloud provider based on the rate of user traffic and the water level of gateway clusters. Lossy migration resets all sessions and reconstructs them in a sandbox within seconds, while lossless migration migrates new sessions to a sandbox, allowing existing sessions to continue serving users seamlessly. Completion of lossless migration depends on the timeout of existing flows, with a median time of approximately 20min.

Throttling (for user app protection). Throttling is leveraged to protect the user apps by limiting the traffic that goes through the gateway. To reduce ineffective CPU usage, we prioritize early rate limiting, dropping packets that exceed the quota when they reach the redirector, rather than waiting until they reach the application layer. Since throttling user traffic may go against the SLA commitments we've made, and throttling brings a persistent service impact (unlike the short service disruption of lossy migration), the intensity and duration of throttling require early communication with users.

Case #1: Lossy migration. When abnormal traffic raises our gateway cluster's water level rapidly, a lossy sandbox migration is necessary. In the past, there have been cases where user traffic suddenly saturated 80% of the backend sessions, triggering a backend-level alert. Subsequent analysis revealed signs of an attack — #TCP sessions surged without a corresponding increase in RPS. In response, we conducted a lossy migration. After communicating with our customer, it was confirmed that they were indeed under attack.

Case #2: Lossless migration. When detecting abnormal traffic but the backend remains stable, a lossless migration is recommended. For instance, there have been cases where user traffic slowly increased over hours. As they had purchased auto-scaling services, we kept scaling resources in the gateway. However, such unusual

scaling, differing from their historical usage pattern, prompted us to confirm with the user whether the traffic rise was due to normal service. Upon user self-check, an attack was discovered.

Case #3: Traffic throttling. Throttling often occurs with social media customers due to unexpected surges in traffic with hotspot events. The rapid influx of traffic in a short time can easily overload a customer's K8s cluster. When requests accumulate, it may lead to a query of death of their resources and a global service outage. Although the customer's K8s cluster supports auto-scaling, the elasticity is limited by the resource creation and configuration speed, which cannot keep up with the sudden growth of traffic. To address this, we protect customer apps by throttling at the mesh gateway after confirming with them. Subsequently, we gradually relax the throttling based on the capacity of customer resource scaling. Without throttling, if the processing capacity of customer apps is less than the traffic, a service outage will inevitably occur.

Besides, we have observed an interesting phenomenon where outages caused by hotspot events can spread across platforms in a short time. When social media end users cannot access content on one platform, they are likely to migrate to others, potentially causing eventual unavailability of all platforms (another query of death). Through throttling, we ensure platform access for a portion of users first, thereby reducing cross-platform traffic and reserving valuable time for resource scaling on other platforms.

6.3 Traffic Migration for In-Phase Services

When we discover in-phase services within a backend based on traffic pattern monitoring, we need to scatter these in-phase services onto other backends with out-of-phase traffic patterns. To achieve this, we need to (i) select the services to migrate and (ii) select the backends for landing these services.

Which services to migrate? To select the appropriate services for migration, we adhere to the following two principles: (i) prioritize services with higher RPS for migration; (ii) prioritize services with fewer long-lasting sessions for migration. The first principle aims to minimize churn caused by reconfiguration and redeployment of services. Selecting services with higher RPS allows us to migrate fewer services overall. The second principle ensures that the selected services can transition to the new backends more quickly, reducing the overhead of maintaining states and configurations across both new and original backends. Note that when counting the RPS, HTTPS sessions should be weighted three times higher than HTTP sessions due to our observation showing that HTTPS requests consume approximately three times more resources.

Which backends to land the services? We select target backends based on backend locations and traffic patterns. The basic principle is to choose backends within the same AZ that have complementary traffic patterns to the selected services. Choosing the same AZ ensures consistent performance before and after migration. Choosing complementary patterns is to minimize the risk of exceeding the target backends' safety threshold after service migration and to achieve a better balance in resource utilization in our cloud.

Specifically, we first obtain the half-width at half-maximum (HWHM) time period of the selected service within 24 hours. Next, we conduct ten samplings at fixed intervals during the HWHM period to obtain sampling points. Then, we sample other backends in the same AZ at the same sampling points, and the set of sampling values is recorded as G. Subsequently, we identify the five backends with the lowest sum of sampling values in G. We then sample the RPS of the five selected backends over the past 24 hours, and the set of sampling values is recorded as G'. Finally, we select the backends with the lowest sum of sampling values in G' as the target backends.

6.4 Proof for Absence of Failure

As a cloud provider, we receive complaints from tenants when issues arise with their hosted services. However, troubleshooting these service-level issues is complex as they are closely related to the availability of the underlay network, the overlay network, the mesh gateway, and the hosted services. To ensure the absence of failure in our cloud infra, we deploy diverse app instances (like WebSocket, HTTP, HTTPS, gRPC, *etc.*) across all AZs, periodically sending full-mesh probing traffic. This strategy enables us to troubleshoot and prove our innocence by assessing the connectivity and performance between different service types, particularly focusing on L7 service anomalies. This distinguishes our approach from other telemetry solutions focusing primarily on network connectivity [52, 67].

7 RELATED WORK

The open-source community has proposed diverse implementations of service mesh, such as Istio [30], Linkerd [32], Cilium [23], Cilium with Envoy [12], Ambient [38], *etc*, which can be categorized into per-pod sidecar and sidecar-less solutions. To address per-pod sidecar's issues, Cilium and Ambient adopt a sidecar-less architecture. Specifically, Ambient achieves this by sharing sidecar functions at the node level and service level [38], while Cilium uses eBPF to embed part of the sidecar functions into the kernel [23] and deploys complex L7 functions optionally with an on-node proxy [12]. However, these sidecar-less solutions still retain heavy proxies on user nodes, leading to numerous open issues as discussed in §2.2.

Most research studies focus on harnessing service mesh for various tasks [41, 42, 48, 49, 54, 66]. Only a few focus on service mesh implementation. Muppet [50] leverages synthesis to address multi-party configuration with service mesh. SPRIGHT [61] adopts eBPF and shared memory to improve service mesh performance. However, eBPF lacks support for flexible L7 function implementation. Additionally, the requirement for pods within the same service function chain to be deployed on the same node using shared memory restricts the elasticity and scalability of its deployment. ServiceRouter [63] introduces Meta's experiences with service mesh. However, 99% of its traffic is still processed using a library-based solution, with only 1% of the traffic handled with service mesh.

8 CONCLUSION

This paper discusses issues in large-scale service mesh deployment from a cloud provider's perspective. To alleviate the intrusion of deploying sidecar proxies within user resources and achieve more efficient resource utilization, we propose Canal Mesh, a sidecar-free multi-tenant service mesh architecture. The centralized architecture reduces management overhead while maintaining security and observability capabilities. Hosted on the public cloud, Canal maximizes resource utilization, and transparently addresses the high availability and elasticity needs in service mesh deployment.

Ethics. *This work does not raise any ethical issues.*

Acknowledgements. The authors would like to thank the shepherd David A. Maltz and the anonymous reviewers for their constructive comments. This work was partially supported by the Key R&D Program of Zhejiang Province, China (2023R5202).

REFERENCES

[1] 2018. Istio Soft Multi-Tenancy Support. https://istio.io/v1.10/blog/2018/soft-multitenancy. (2018).

[2] 2021. Benchmarking Linkerd and Istio. https://linkerd.io/2021/05/27/linkerd-vs-istio-benchmarks/#latency-at-20-rps. (2021).

[3] 2021. How eBPF Streamlines the Service Mesh. https://thenewstack.io/how-ebpf-streamlines-the-service-mesh. (2021).

[4] 2021. How eBPF will solve Service Mesh - Goodbye Sidecars. https://isovalent.com/blog/post/2021-12-08-ebpf-servicemesh/#sidecar-vs-per-node-proxy. (2021).

[5] 2021. Sidecar Concept In 3 Minutes. https://medium.com/code-factory-berlin/sidecar-concept-in-2-minutes-a9f834cffe6f. (2021).

[6] 2022. A Comprehensive Guide to Canary Releases. https://www.getambassador.io/blog/comprehensive-guide-to-canary-releases. (2022).

[7] 2022. Ambient version. https://gcsweb.istio.io/gcs/istio-build/dev/0.0.0-ambient.191fe680b52c1754ee72a06b3e0d3f9d116f2e82. (2022).

[8] 2022. CNCF 2022 Annual Survey. https://www.cncf.io/reports/cncf-annual-survey-2022. (2022).

[9] 2022. Introducing Ambient Mesh. https://istio.io/latest/blog/2022/introducing-ambient-mesh. (2022).

[10] 2022. Traffic types and iptables rules in Istio sidecar explained. https://tetrate.io/traffic-types-and-iptables-rules-in-istio-sidecar-explained. (2022).

[11] 2023. Accelerate Microservice Networking Performance with 4th Gen Intel Xeon Scalable Processor. https://networkbuilders.intel.com/solutionslibrary/microservices-solution-optimizations-with-intel-xeon-scalable-processor-solution-brief. (2023).

[12] 2023. Cilium Service Mesh - Everything You Need to Know. https://isovalent.com/blog/post/cilium-service-mesh. (2023).

[13] 2023. Elastic Compute Service price in Ali cloud. https://www.alibabacloud.com/en/product. (2023).

[14] 2023. Getting Started with Multi-tenancy and Routing Delegation in Gloo Platform. https://www.solo.io/blog/multi-tenancy-routing-gloo-gateway. (2023).

[15] 2023. Istio 1.17.0. https://github.com/istio/istio/releases/tag/1.17.0. (2023).

[16] 2023. Istio Ambient Waypoint Proxy Made Simple. https://istio.io/latest/blog/2023/waypoint-proxy-made-simple. (2023).

[17] 2024. Accelerating OpenSSL Using Intel QuickAssist Technology. https://www.intel.com/content/dam/www/public/us/en/documents/solution-briefs/accelerating-openssl-brief.pdf. (2024).

[18] 2024. Addressing Cascading Failures. https://sre.google/sre-book/addressing-cascading-failures/. (2024).

[19] 2024. Alibaba Cloud Service Mesh. https://www.alibabacloud.com/product/servicemesh. (2024).

[20] 2024. AWS App Mesh. https://aws.amazon.com/app-mesh. (2024).

[21] 2024. Azure Service Fabric. https://azure.microsoft.com/en-us/products/service-fabric. (2024).

[22] 2024. Calico CNI plugin. https://github.com/projectcalico/calico. (2024).

[23] 2024. Cilium Service Mesh. https://cilium.io/use-cases/service-mesh. (2024).

[24] 2024. Crypto Accelerations in Istio and Envoy with Intel Xeon Scalable Processors. https://networkbuilders.intel.com/solutionslibrary/service-mesh-crypto-accelerations-istio-envoy-intel-xeon-sp-user-guide. (2024).

[25] 2024. eBPF. https://ebpf.io. (2024).

[26] 2024. Envoy is an open source edge and service proxy, designed for cloud-native applications. https://www.envoyproxy.io. (2024).

[27] 2024. Flannel CNI plugin. https://github.com/flannel-io/flannel. (2024).

[28] 2024. Google Cloud Service Mesh. https://cloud.google.com/products/service-mesh. (2024).

[29] 2024. Iptables Redirection. https://release-v1-2.docs.openservicemesh.io/docs/guides/traffic_management/iptables_redirection. (2024).

[30] 2024. Istio: simplify observability, traffic management, security, and policy with the leading service mesh. https://istio.io. (2024).

[31] 2024. Kubernetes. https://kubernetes.io. (2024).

[32] 2024. Linkerd: the world's most advanced service mesh. https://linkerd.io. (2024).

[33] 2024. NAT Gateway. https://www.alibabacloud.com/product/nat. (2024).

[34] 2024. Netperf. https://github.com/HewlettPackard/netperf. (2024).

[35] 2024. Overview of ENIs. https://www.alibabacloud.com/help/en/ecs/user-guide/overview-48. (2024).

[36] 2024. Server Load Balancer. https://www.alibabacloud.com/product/server-load-balancer. (2024).

[37] 2024. Sidecar Containers in Kubernetes Pods. https://www.baeldung.com/linux/kubernetes-pods-sidecar-containers. (2024).

[38] 2024. What is Istio Ambient Mode? https://www.solo.io/topics/istio/ambient-mode. (2024).

[39] 2024. Workload isolation using shuffle-sharding. https://aws.amazon.com/builders-library/workload-isolation-using-shuffle-sharding. (2024).

[40] 2024. wrk - a HTTP benchmarking tool. https://github.com/wg/wrk. (2024).

[41] Gianni Antichi and Gábor Rétvári. 2020. Full-stack SDN: The next big challenge?. In *Proceedings of the Symposium on SDN Research*. 48–54.

[42] Sachin Ashok, P Brighten Godfrey, and Radhika Mittal. 2021. Leveraging service meshes as a new network layer. In *Proceedings of the 20th ACM Workshop on Hot Topics in Networks*. 229–236.

[43] Deepak Bansal, Gerald DeGrace, Rishabh Tewari, Michal Zygmunt, James Grantham, Silvano Gai, Mario Baldi, Krishna Doddapaneni, Arun Selvarajan, Arunkumar Arumugam, et al. 2023. Disaggregating stateful network functions. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. 1469–1487.

[44] Rajdeep Bhanot and Rahul Hans. 2015. A review and comparative analysis of various encryption algorithms. *International Journal of Security and Its Applications* 9, 4 (2015), 289–306.

[45] Karthikeyan Bhargavan, Ioana Boureanu, Pierre-Alain Fouque, Cristina Onete, and Benjamin Richard. 2017. Content delivery over TLS: a cryptographic analysis of keyless SSL. In *2017 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, 1–16.

[46] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, et al. 2014. P4: Programming protocol-independent packet processors. *ACM SIGCOMM Computer Communication Review* 44, 3 (2014), 87–95.

[47] Lee Calcote and Zack Butcher. 2019. *Istio: Up and running: Using a service mesh to connect, secure, control, and observe*. O'Reilly Media.

[48] Lianjie Cao and Puneet Sharma. 2021. Co-locating containerized workload using service mesh telemetry. In *Proceedings of the 17th International Conference on emerging Networking EXperiments and Technologies*. 168–174.

[49] Boutheina Dab, Ilhem Fajjari, Mathieu Rohon, Cyril Auboin, and Arnaud Diquélou. 2020. Cloud-native service function chaining for 5G based on network service mesh. In *ICC 2020-2020 IEEE International Conference On Communications (ICC)*. IEEE, 1–7.

[50] Kevin Dackow, Andrew Wagner, Tim Nelson, Shriram Krishnamurthi, and Theophilus A Benson. 2020. Solver-Aided Multi-Party Configuration. In *Proceedings of the 19th ACM Workshop on Hot Topics in Networks*. 103–109.

[51] João Tiago Duarte Maia and Filipe Figueiredo Correia. 2022. Service mesh patterns. In *Proceedings of the 27th European Conference on Pattern Languages of Programs*. 1–12.

[52] Chuanxiong Guo, Lihua Yuan, Dong Xiang, Yingnong Dang, Ray Huang, Dave Maltz, Zhaoyi Liu, Vin Wang, Bin Pang, Hua Chen, et al. 2015. Pingmesh: A large-scale system for data center network latency measurement and analysis. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*. 139–152.

[53] C Hopps. 2000. RFC 2992: Analysis of an Equal-Cost Multi-Path Algorithm. (2000).

[54] Haan Johng, Anup K Kalia, Jin Xiao, Maja Vuković, and Lawrence Chung. 2019. Harmonia: A continuous service monitoring framework using devops and service mesh in a complementary manner. In *Service-Oriented Computing: 17th International Conference, ICSOC 2019, Toulouse, France, October 28–31, 2019, Proceedings 17*. Springer, 151–168.

[55] Matt Klein. 2017. Lyft's Envoy: Experiences Operating a Large Service Mesh. USENIX Association, San Francisco, CA.

[56] M Mahalingam, D Dutt, K Duda, P Agarwal, L Kreeger, T Sridhar, M Bursell, and C Wright. 2014. RFC 7348: Virtual eXtensible Local Area Network (VXLAN): A Framework for Overlaying Virtualized Layer 2 Networks over Layer 3 Networks. (2014).

[57] John Nagle. 1984. RFC 896: Congestion Control in IP/TCP Internetworks. (1984).

[58] Vladimir Olteanu, Alexandru Agache, Andrei Voinescu, and Costin Raiciu. 2018. Stateless datacenter load-balancing with beamer. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*. 125–139.

[59] Tian Pan, Kun Liu, Xionglie Wei, Yisong Qiao, Jun Hu, Zhiguo Li, Jun Liang, Tiesheng Cheng, Wenqiang Su, Jie Lu, et al. 2024. {LuoShen}: A {Hyper-Converged} Programmable Gateway for {Multi-Tenant}{Multi-Service} Edge Clouds. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*. 877–892.

[60] Tian Pan, Nianbing Yu, Chenhao Jia, Jianwen Pi, Liang Xu, Yisong Qiao, Zhiguo Li, Kun Liu, Jie Lu, Jianyuan Lu, et al. 2021. Sailfish: Accelerating cloud-scale multi-tenant multi-service gateways with programmable switches. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*. 194–206.

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

[62] Eric Rescorla. 2018. RFC 8446: The Transport Layer Security (TLS) Protocol Version 1.3. (2018).

[63] Harshit Saokar, Soteris Demetriou, Nick Magerko, Max Kontorovich, Josh Kirstein, Margot Leibold, Dimitrios Skarlatos, Hitesh Khandelwal, and Chunqiang Tang. 2023. {ServiceRouter}: Hyperscale and Minimal Cost Service Mesh at Meta. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*. 969–985.

[64] George J Stigler. 1958. The economies of scale. *The Journal of Law and Economics* 1 (1958), 54–71.

[65] Chengkun Wei, Xing Li, Ye Yang, Xiaochong Jiang, Tianyu Xu, Bowen Yang, Taotao Wu, Chao Xu, Yilong Lv, Haifeng Gao, et al. 2023. Achelous: Enabling Programmability, Elasticity, and Reliability in Hyperscale Cloud Networks. In *Proceedings of the ACM SIGCOMM 2023 Conference*. 769–782.
[66] Łukasz Wojciechowski, Krzysztof Opasiak, Jakub Latusek, Maciej Wereski, Victor Morales, Taewan Kim, and Moonki Hong. 2021. Netmarks: Network metrics-aware kubernetes scheduler powered by service mesh. In *IEEE INFOCOM 2021-IEEE Conference on Computer Communications*. IEEE, 1–9.
[67] Shunmin Zhu, Jianyuan Lu, Biao Lyu, Tian Pan, Chenhao Jia, Xin Cheng, Daxiang Kang, Yilong Lv, Fukun Yang, Xiaobo Xue, et al. 2022. Zoonet: a proactive telemetry system for large-scale cloud networks. In *Proceedings of the 18th International Conference on emerging Networking EXperiments and Technologies*. 321–336.
[68] Xiangfeng Zhu, Weixin Deng, Banruo Liu, Jingrong Chen, Yongji Wu, Thomas Anderson, Arvind Krishnamurthy, Ratul Mahajan, and Danyang Zhuo. 2023. Application Defined Networks. In *Proceedings of the 22nd ACM Workshop on Hot Topics in Networks*. 87–94.
[69] Xiangfeng Zhu, Guozhen She, Bowen Xue, Yu Zhang, Yongsu Zhang, Xuan Kelvin Zou, XiongChun Duan, Peng He, Arvind Krishnamurthy, Matthew Lentz, et al. 2023. Dissecting overheads of service mesh sidecars. In *Proceedings of the 2023 ACM Symposium on Cloud Computing*. 142–157.

APPENDICES

Appendices are supporting material that has not been peer-reviewed.

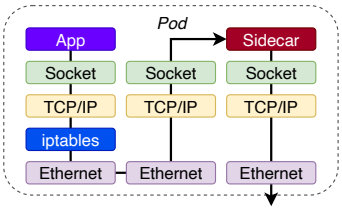


Figure 21: Traffic redirection with iptables to sidecar.

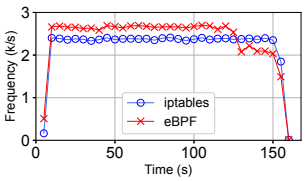


Figure 22: Higher context switch frequency of eBPF (16B packets, 4kRPS).

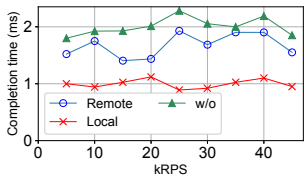


Figure 23: Completion time of crypto with remote/local/no offloading.

A OVERHEAD ANALYSIS OF CANAL MESH

Per-node *vs* per-pod. Compared to sidecars running in each pod, our on-node proxy ensures observability and security capabilities with almost no compromise, but it does require some additional work. For example, to track the inbound and outbound traffic for each pod, a per-pod sidecar can directly perform statistics without additionally labeling the traffic. However, our on-node proxy needs to differentiate traffic from different pods by introducing additional labels for fine-grained statistics at the pod level.

Key server. Compared to local CPU offloading, offloading asymmetric crypto remotely to a key server will increase the blast radius. To address this, in the event of a failure in the remote key server in the local AZ, we will fallback to using the local CPU as a backup for asymmetric crypto offloading.

Compared to local CPU offloading, using a remote key server also introduces additional transmission latency within an AZ. To evaluate this, we measure the request completion time for local offloading, remote offloading and no offloading (*i.e.*, software-based asymmetric crypto on old CPU models). As shown in Fig. 23, the

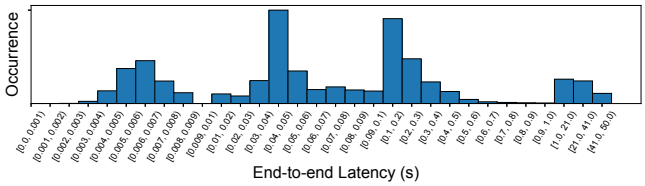


Figure 24: Distribution of end-to-end latency in a production K8s cluster.

completion time for remote offloading (including the RTT between the requester and the key server) remains relatively stable at around 1.7ms, regardless of workload changes. The completion times for local offloading and no offloading are 1ms and 2ms, respectively. It shows that remote offloading is faster than no offloading, indicating that the added RTT is outweighed by the time saved through asymmetric crypto offloading. Compared to local offloading, remote offloading introduces only 0.7ms additional latency.

To further analyze the latency impact introduced by remote offloading with the key server, we measure the distribution of end-to-end latency within a production K8s cluster (with mesh gateway), as shown in Fig. 24. It can be observed that the majority of latencies fall within the range of 40~50ms and 100~200ms. This indicates that, in most cases, the 0.7ms latency introduced by the key server is negligible compared to the processing latencies of user apps.

Hairpin routing to mesh gateway. In Canal Mesh, we enforce all traffic to detour through the remote centralized mesh gateway (akin to hairpin routing), even for communication between two neighboring pods within the same node. Despite sounding costly, this approach does not impose significant extra overhead on the service chain processing of user apps. The reason is that the RTT within the same AZ (*e.g.*, less than 1ms) is much lower than the average request processing time at the application layer within the user pod (*e.g.*, 40-200ms) as shown in Fig. 24. Therefore, the additional transmission delay caused by hairpin routing does not noticeably increase the request processing time.

LB disaggregation. When using redirectors to replace an LB, longer replica chains in the bucket table (*e.g.*, greater than 2) may result in multiple redirections and potential latency overhead. However, our analysis indicates that this latency overhead is not significant for several reasons. Firstly, based on our production data, multiple consecutive scale-out/scale-in events within a short period occur infrequently. Secondly, even in cases where multiple redirections occur, they are typically short-lived. Our observation shows that most traffic consists of transient flows, which establish new sessions with higher-priority backends after a timeout, thereby reducing the need for continued redirections. Lastly, introducing an additional hop does not substantially increase end-to-end latency, given that RTTs within an AZ are generally less than 1ms.

Session aggregation. Addressing session resource shortages via session aggregation involves several overheads, including packet encapsulation/decapsulation, increased blast radius, additional operational complexity, and the risk of exceeding the MTU limit. Specifically, encapsulation can be achieved at line rate with Tofino, resulting in minimal overhead. However, decapsulation needs to be performed internally in the VM, consuming additional CPU cycles. Based on our measurement, we find that whether or not to perform additional decapsulation has an insignificant impact on CPU

utilization, which we consider acceptable. When multiple sessions are aggregated into one, any exception affecting the aggregated session will impact all user traffic within that session, unavoidably increasing the blast radius. Before encapsulation into the tunnel, identifying the root cause of a faulty session is straightforward. However, after encapsulation, it becomes necessary to decapsulate and enter the tunnel to pinpoint the source of the fault, thus increasing the operational complexity. Furthermore, appending an additional VXLAN header to packets may cause them to exceed the MTU. To mitigate this issue, we adjusted the device's MTU limit.

B SPECIAL DEPLOYMENT SCENARIOS

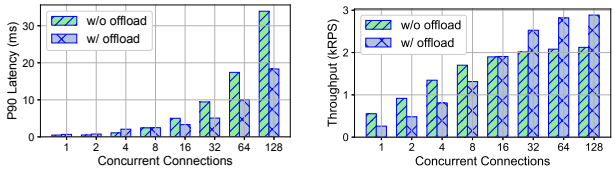
Cloud-based keyless service mesh. Some customers have extremely high security requirements (such as financial customers) and do not wish to entrust their private keys (used in asymmetric crypto) to third parties [45]. However, when deploying a service mesh based on Istio or Ambient, it is necessary to provide the private keys to the mesh admin to enable mTLS crypto based on sidecars/proxies. If the service mesh based on Istio or Ambient is deployed on the public cloud, the private keys have to be given to the cloud providers. However, this is not acceptable to those customers with high security concerns.

By contrast, our remote key server architecture supports the keyless TLS mode [45] due to security functionality decoupling from sidecars/proxies. This allows users to enjoy the convenience of cloud-based service mesh without having to entrust their private keys to cloud providers. Specifically, users can choose not to use our multi-tenant shared key server to store their private keys. Instead, they can opt for deploying a local key server in their own on-premises data centers to store the private keys and handle asymmetric crypto requests from the cloud. Such a keyless service mesh eliminates the risk of exposing private keys.

Cloud-based proxyless service mesh. We have limited the intrusion into user nodes by using the remote centralized mesh gateway and remote asymmetric crypto offloading. However, even such limited intrusion is not acceptable for some customers whose resource access is entirely blocked to third parties due to security concerns. To address this issue, we propose a cloud-based proxyless service mesh that completely eliminates the on-node proxy.

With the proxyless mode, we need an alternative method to redirect traffic from the user cluster to our mesh gateway. In most cases, as the cloud provider, we can access the user's DNS servers. With the user's permission, the cloud provider can configure the DNS servers to redirect service traffic to the remote gateway. However, because there is no on-node proxy to collaborate with the mesh gateway on zero-trust network capabilities and data collection for observability, these functions become partially usable.

With the proxyless mode, we recommend to do the authentication through the virtual network interfaces (*i.e.*, ENIs [35]) attached to the containers. Most virtual network interfaces provided by cloud providers have the authentication and verification mechanisms embedded to ensure the traffic goes through the network interfaces cannot be forged or tampered with. However, this authentication scheme has two issues. First, we need to create a virtual network interface for every container. Because each interface requires memory resources on the node and IP addresses allocated from the network, as the number of containers grows, the maximum limit of interfaces



(a) Latency (b) Throughput

Figure 25: Performance under different #newly established concurrent connections with AVX-512.

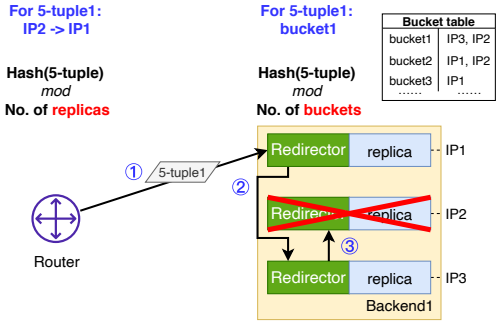


Figure 26: Session consistency maintenance.

is easily hit. Second, we need a protection mechanism to ensure that the virtual network interfaces are only accessed by the attached containers, a feature not well supported by popular open-source solutions like Flannel [27] and Calico [22] in K8s.

With the proxyless mode, encryption has to switch from a fully managed mode to a semi-managed mode. If users manage their own certificates, the encryption can be done equivalently as the the on-node proxy mode. Users can also rely on the TLS encryption services of the mesh gateway, if the cloud providers are trusted.

With the proxyless mode, traffic collection, labeling and sampling cannot be done on the user nodes. But those actions are still taken at the mesh gateway. As a result, we only provide the partial observability capabilities on the traffic at the mesh gateway.

C ADDITIONAL FIGURES

Performance degradation in mTLS acceleration. After offloading asymmetric crypto locally using AVX-512, we had an intriguing discovery: in certain cases, the throughput and latency become unexpectedly worse than without offloading. We conducted experiments to evaluate the performance under different numbers of newly established concurrent connections, as depicted in Fig. 25. As shown in the figure, there is a significant performance degradation when the number of new concurrent connections is below 8.

After careful analysis, we found that the performance degradation is due to the batch processing nature of AVX-512. When the buffer of AVX-512 is not fully occupied, it causes a wait time until a timeout occurs. The wait time is configurable with a minimum threshold of 1ms. The buffer size in AVX-512 is 512 bits, allowing for processing of 8 crypto operations in a batch. Therefore, when the number of concurrent connections is less than 8, the performance of AVX-512 will degrade.

Session consistency maintenance with redirector. We use a case to show, when a replica is about to go offline, how the redirector

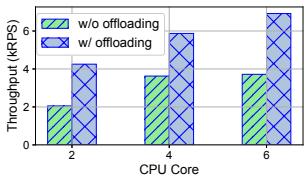


Figure 27: Throughput improvement with offloading.

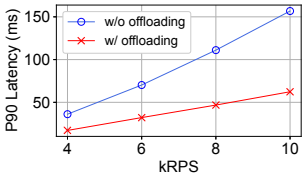


Figure 28: Latency improvement with offloading.

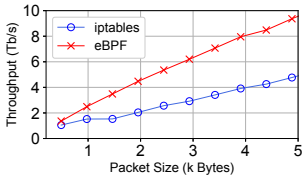


Figure 29: Throughput improvement with eBPF.

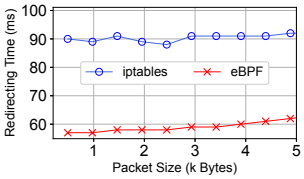


Figure 30: Latency improvement with eBPF.

ensures that the replica no longer processes new sessions while continues to serve old sessions.

As shown in Fig. 26, before replica IP2 is about to go offline, the flow 5-tuple1 is hashed by the router in the front through ECMP to replica IP2. After IP2 goes offline, the router will statelessly hash the traffic to the remaining replicas (by hashing 5-tuple mod #replicas). However, this may cause the traffic that previously went to IP2 to be hashed to different replicas, which disrupts the session consistency of the traffic with existing flow records on IP2.

To solve this problem, we follow Beamer's idea [58] by adding a redirector to each replica, which stores a bucket table with a fixed number of entries. The bucket tables on different replicas store the same entries, updated by a centralized controller. A fixed number of entries in each bucket table ensures that packets with the same 5-tuple (*i.e.*, belonging to the same flow) will be hashed to the same bucket entry (by hashing 5-tuple mod #buckets). Each bucket entry contains a replica chain, sorted by priority. When replica IP2 is about to go offline, we will add a higher priority available replica in front of IP2 in all bucket entries containing IP2 in the bucket table (such as IP3 in bucket1 and IP1 in bucket2).

When any packet (including the first packet of a new flow), reaches a replica based on the router's hashing result, it first queries the replica chain in the redirector of that replica, rather than being processed directly at the replica. For example, if 5-tuple1 is initially hashed to bucket1, the replica chain in bucket1 is then checked. If IP3 has the highest priority at that moment, the packet needs to be redirected to IP3. When this packet reaches IP3, because the kernel stack lookup does not find the flow stored at IP3 for 5-tuple1, it continues to search in the replica chain for the next priority replica, which is IP2. Therefore, the existing flow will ultimately be redirected to IP2, and finding its flow record in IP2's kernel stack.

If this packet is a SYN packet (*i.e.*, the first packet of a new flow), it will directly choose to insert at IP3. In other words, the new flow will be placed on the replica with highest priority (*i.e.*, the newly available replica) in its hash bucket (*i.e.*, IP3 in bucket1). In the future, subsequent packets of this new flow will also be hashed to bucket1 and then redirected to IP3. However, because these packets are not SYN packets and there is already a flow record existing in the kernel stack, they will be processed locally at IP3. When the flows in replica IP2 have all aged, IP2 can be safely taken offline.

This approach ensures that replicas preparing to go offline will continue to handle previously established sessions correctly but will not handle the establishment of new sessions.

Performance improvement with key server. To evaluate the impact of crypto offloading on latency and throughput, we conduct experiments using wrk [40] to measure the throughput and P90 latency of HTTPs short flows [62].

Throughput. Fig. 27 illustrates the improvement in throughput through offloading when providing different CPU cores to the node proxy. As shown in the figure, the throughput with crypto offloading is approximately 1.6x~1.8x times than that of without offloading.

Latency. Fig. 28 illustrates the decrease in latency with crypto offloading. As shown in the figure, as the RPS increases, the rate of latency reduction becomes higher, ranging from 53% to 60%. The reason behind this is that as the RPS increases, the resources of the on-node proxy become increasingly exhausted, resulting in a sharp increase in processing latency.

Performance improvement with eBPF. As mentioned in §4.1.2, the packet size may have an impact on the throughput and latency when using eBPF-based redirection. To validate the feasibility of our proposed approach, we leverage Netperf [34] to evaluate the performance of redirecting the traffic to the per-node proxy with eBPF and iptables, under different packet sizes.

Throughput. Fig. 29 shows the throughput improvement with eBPF-based redirection. Both iptables-based and eBPF-based redirection enable the Nagle algorithm for aggregating small packets. As shown in the figure, the throughput with eBPF increases significantly for both small and large packets, compared to iptables-based redirection. The throughput increases by approximately 2 times for packet sizes larger than 1500 bytes. For smaller packets (*e.g.*, 500 bytes), the throughput with eBPF is approximately 1.3 times higher. This indicates that the throughput improvement is more significant for larger packets, as there is no need for packet aggregation.

Latency. As shown in Fig. 30, it is evident that eBPF also introduces a significant improvement in latency. The latency of iptables-based redirection is approximately 1.5x~1.8x compared to that of eBPF-based. In contrast to throughput, the latency of both approaches show less sensitivity to the changes in packet size.