Enabling Fast and Stable Service Mesh Communication via Piggyback Layer-7 Traffic Control on Programmable Switches

Gonglong Chen[1], Jiacong Li[3], Yuxin Xu[1,2], Baiyan Ke[4], Zhitao Lan[5], Wenxing Ge[5], Haiying Shen[6], Jiamei Lv[7], Tao Gu[8], Chengzhong Xu[9], and Kejiang Ye[1*]
[1]Shenzhen Institutes of Advanced Technology, CAS, [2]University of CAS, [3]Northeastern University. [4]Foshan University. [5]Independent Researcher. [6]University of Virginia. [7]Zhejiang University. [8]Macquarie University. [9]University of Macau.
Email: {*gl.chen2, yx.xu2 ,kj.ye*}*@siat.ac.cn*, *li.jiaco@northeastern.edu*, {*zhitaolan, vincentge95*}*@gmail.com*, *baiyanke29@gmail.com*, *lvjm@zju.edu.cn*, *hs6ms@virginia.edu*, *czxu@um.edu.mo*, *tao.gu@mq.edu.au*

***Abstract*—Service mesh has become an essential infrastructure for managing cloud-native microservices, widely adopted by major providers to streamline service orchestration and reduce operational overhead. A core component of service mesh architecture is the sidecar proxy, managing policy-based routing, Layer-7 load balancing, and related functions. Traditional per-pod *distributed* sidecar deployments route all inter-pod communication through local proxies, introducing substantial resource consumption and inefficiencies. Recent approaches advocate for *centralized* proxy deployments to offload compute-intensive modules to gateway nodes; however, this design introduces non-trivial traffic detours and exacerbates network congestion.**

To overcome these limitations, we propose PiggyCar, a novel service mesh communication system that *piggybacks* Layer-7 traffic control onto programmable network devices. In PiggyCar, resource-intensive tasks are offloaded to intermediary switches along inter-pod paths, enabling in-network execution of advanced network functions. PiggyCar incorporates a latency-aware offline planner for optimal network policy placement and a stability-oriented online scheduler that dynamically adapts to traffic fluctuations in real time. We prototype PiggyCar on a testbed comprising six servers and four programmable switches. Experimental results show that PiggyCar cuts latency by up to 97.2% compared to Canal, boosts throughput by up to 1.82× under high RPS conditions, and consistently delivers the lowest network jitter.

I. Introduction

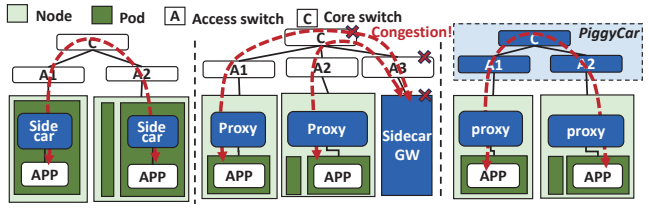
Service mesh is a critical infrastructure for cloud microservices, adopted by providers such as AWS [1], Azure [2], GCP [3], and Alibaba [4] to streamline management and reduce costs. Studies show that up to a 40% boost in development speed and $40,000 in monthly savings [5], [6]. A key component in the service mesh frameworks like Istio [7] and Linkerd [8] is the sidecar proxy, which handles policy-based routing, Layer-7 (L7) load balancing, and A/B testing. However, traditional per-pod *distributed* sidecar deployments force all inter-pod communication through the proxy (as shown in Fig. 1(a)), incurring over 30% CPU and 25% memory overhead [9], [10], [11] and potentially doubling latency.

To mitigate the high resource consumption of traditional per-pod sidecars, recent works propose *centralized* deployments such as Ambient Mesh [12] and Canal Mesh [9]. These approaches offload resource-intensive modules from individual pods. Ambient Mesh [12] uses a Layer-4 (L4) proxy on each node and centralizes L7 processing. Canal Mesh [9] shifts these modules to a public cloud to reduce per-tenant resource usage. However, *centralized* deployments force inter-pod traffic detouring even when pods reside on the same node (as shown in Fig. 1(b)). This detouring causes extra hops and increases bandwidth consumption. Intermediate devices experience higher buffer occupancy. Our experiments in Section II indicate that under DCTCP (a congestion control algorithm widely adopted by major cloud vendors [13], [14]), the congestion control mechanism is triggered earlier, which leads to a nearly 53% drop in throughput.

To address the aforementioned issues, we propose PiggyCar, a *piggyback* service mesh communication system (as shown in Fig. 1(c)). In PiggyCar, resource-intensive and time-consuming modules are extracted and deployed on intermediary network devices along the inter-pod communication links. Traffic between pods is processed by these devices, which execute rich network functions such as L7 load balancing and policy-based routing on the fly. Unlike existing *distributed* approaches (e.g., Istio [7]) that burden user pods with heavy resource consumption or *centralized* schemes (e.g., Canal [9]) that require detouring traffic to remote gateways for policy parsing, PiggyCar integrates policy parsing directly into the communication path. This *piggyback* approach minimizes pod resource overhead and avoids detouring-induced throughput loss, yielding a truly sidecar-transparent experience for applications.

However, achieving the above *piggyback* service mesh faces the following challenges: **Challenge 1: parsing complex L7 policies on programmable switches under zero-trust constraints.** Implementing rich L7 network policies (e.g., load balancing) on programmable switches is challenging. These switches typically parse fixed-length headers (like IP and port) and cannot directly handle variable-length, string-based protocols (such as HTTP) due to the hardware limitations [15]. Moreover, service mesh traffic is encrypted to support zero-trust, so decryption is required before any L7 parsing. To address this, we propose a *decoupled* L7 parsing approach. A rule-based on-node proxy performs rapid, minimal-overhead

*Corresponding author.



(a) Distributed sidecar (e.g., Istio).
(b) Centralized sidecar (e.g., Canal, Ambient).
(c) Piggybacked sidecar (ours).

Fig. 1: The comparison of existing works with PiggyCar. (a) incurs significant resource overhead on each pod, while (b) results in a many-to-one traffic pattern that doubles the likelihood of congestion. In contrast, (c) offloads intensive pod operations to intermediary network devices, thereby enabling direct pod-to-pod communication to reduce latency.

traffic classification and directly assigns a policy ID to the outer VXLAN header. The programmable switch then relies solely on the unencrypted outer header for policy matching and traffic management, eliminating the need for heavy packet inspection. This streamlined design minimizes resource usage and simplifies implementation. On-the-fly matching reduces on-node proxy overhead by over 90% (see Section III-C), and periodic randomization of policy IDs further enhances security.

Challenge 2: optimally deploying network policies on programmable switches. Determining which programmable switch should host a network policy is non-trivial. Our goal is to achieve in-path *piggybacked* policy parsing for any pod pair to minimize end-to-end latency. However, selecting the optimal switch maps to the NP-hard Capacitated Facility Location Problem [16]. We design a heuristic algorithm to efficiently identify switches on critical communication paths (Section III-D). The algorithm leverages topology information to reduce overall latency and avoid unnecessary detours. This approach ensures rich network policies are deployed to provide a minimal pod-to-pod communication latency.

Challenge 3: adapting to runtime traffic variations for stable communications. Runtime traffic fluctuations may cause the number of sessions associated with network policies to exceed the SRAM capacity of programmable switches. When this capacity is exceeded, sessions are forced to use slower lookup paths (i.e., through switch CPU memory), reducing overall throughput. To address this issue, we propose a lightweight, distributed, stability-oriented online scheduler (Section III-E). Programmable switches monitor their resource utilization and exchange minimal state information with neighboring switches. A heuristic algorithm quickly computes optimal migration strategies for both policies and sessions. The algorithm balances the load and prevents resource overflow. This dynamic adaptation minimizes the risk of reduced application throughput and network jitter under varying traffic conditions, thereby achieving stable communications.

We prototype PiggyCar on a testbed of six servers and four programmable switches and simulate it on large-scale clusters. Both experiments demonstrate that PiggyCar achieves low latency and high stability (Section V). We evaluate PiggyCar using a typical serverless scenario for car parking detection and charging [10]. The results show that PiggyCar reduces latency by up to 97.2% compared to Canal [9] and improves throughput by 1.82× under high RPS. In addition, PiggyCar consistently achieves the lowest network jitter and maintains the highest proportion of flows with minimal flow completion times.

This paper makes the following contributions:

- We analyze existing service mesh frameworks and identify key limitations. Some works use per-pod *distributed* sidecars. These burden user pods with heavy resource consumption. Other works use *centralized* sidecars. These force traffic detours to remote gateways for policy parsing. Neither design achieves fast and stable service mesh communications under heavy traffic (Section II).
- We design PiggyCar, a *piggyback* service mesh communication system. In PiggyCar, resource-intensive modules are extracted and deployed on intermediary network devices along inter-pod communication links. These devices process traffic on the fly, efficiently executing functions such as L7 load balancing and policy-based routing (Section III).
- We implement a PiggyCar prototype on a testbed of six servers and four programmable switches. We evaluate it using a typical serverless scenario. The experiments show that PiggyCar significantly reduces latency and improves stability compared to state-of-the-art solutions (Section IV and Section V).

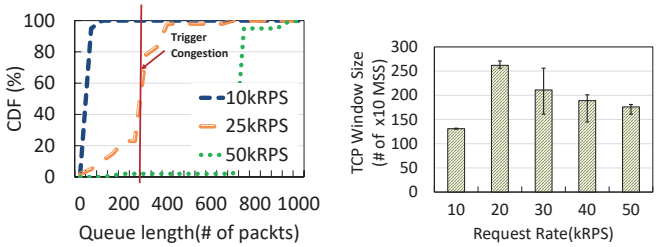
II. Background and Motivations

In this section, we introduce the benefits of the service mesh framework for modern cloud applications and its key component, the sidecar. We then assess the limitations of both the distributed per-pod and centralized sidecar approaches.

A. The Benefits of Service Mesh Framework

Service mesh is an infrastructure layer that orchestrates service-to-service communication in cloud-native microservice architectures [17], [18], [19], [20], [21]. Major cloud providers, including AWS [1], Azure [2], GCP [3], and Alibaba [4], offer service mesh-based products that simplify development and reduce costs. Recent studies report that service mesh frameworks can increase development speed by up to 40% [5] and cut monthly costs by approximately $40,000 [6].

A key component in many service mesh frameworks, such as Istio [7] and Linkerd [8], is the sidecar proxy. The sidecar manages pod network traffic by performing tasks including policy-based routing, load balancing, and rate limiting. For example, to conduct an A/B test in a serverless application, one may configure the routing rules as follows: *Host: reviews.com, route: v1(80%), v2(20%).* This rule directs 80% of traffic from POST requests with the host *reviews.com* to the v1 backend service, and 20% to the v2 service. The sidecar applies these policies automatically, eliminating the need for



(a) Queue length@core switch. (b) TCP window size@Server 1.

Fig. 2: The throughput degradation in a centralized sidecar setup due to the many-to-one traffic pattern. On a 100Gbps server-access link, transmission rates approaching 47.8Gbps (nearly 25kRPS per server) trigger congestion control at the core-access link, reducing average application throughput by almost 53%.

modifications to the application logic and enabling flexible traffic management.

B. The Problems of Traditional Sidecar

Traditional service mesh frameworks use a *distributed* sidecar deployment (e.g., Istio [7]). Each pod is paired with a dedicated sidecar proxy that intercepts all traffic for fine-grained control (see Fig. 1).

Problems. Although this approach decouples networking from application logic, it consumes significant pod resources. Sidecars have been shown to use over 30% of CPU and more than 25% of memory [9]. This resource overhead limits the capacity available for core application functions. Experimental data indicates that when a pod's CPU usage exceeds 45%, inter-pod communication latency doubles, and when it exceeds 75%, latency can increase up to 100-fold [9].

C. The Issues of Recent Works

To address the high resource consumption and performance overhead of per-pod sidecar deployments, recent research has proposed *centralized* sidecar approaches, such as Ambient Mesh [12] and Canal Mesh [9]. These approaches extract time-consuming functions from per-pod sidecars and deploy them in a single gateway, thereby reducing per-pod resource usage.

Ambient Mesh [12] deploys a node-level L4 proxy and a shared L7 proxy (the centralized gateway) to cut pod interference and local resource usage. However, retaining both proxies within the user cluster may lead to resource contention and limited isolation during peak workloads. In contrast, Canal Mesh [9] offloads most sidecar functions from the user cluster to a centralized mesh gateway in the public cloud. This strategy further minimizes pod resource consumption and achieves a balanced offloading approach.

Issues. Nonetheless, both approaches share a key drawback: all pods, even those on the same node, must route traffic through a remote *centralized* gateway (see Fig. 1(b)). This extra hop increases data center traffic and raises the risk of congestion and packet loss, leading to higher latency and lower throughput. For example, compared with the *distributed* sidecar in Fig. 1(a), the *centralized* approach adds 2× transmission hops. This creates a many-to-one traffic pattern that doubles the traffic volume on the link between access switch A3 and core switch C. Existing studies [13], [14] show that this doubling accelerates the triggering of congestion control. Under DCTCP (a congestion control algorithm widely adopted by major cloud vendors [13]), such conditions typically reduce application throughput by nearly half.

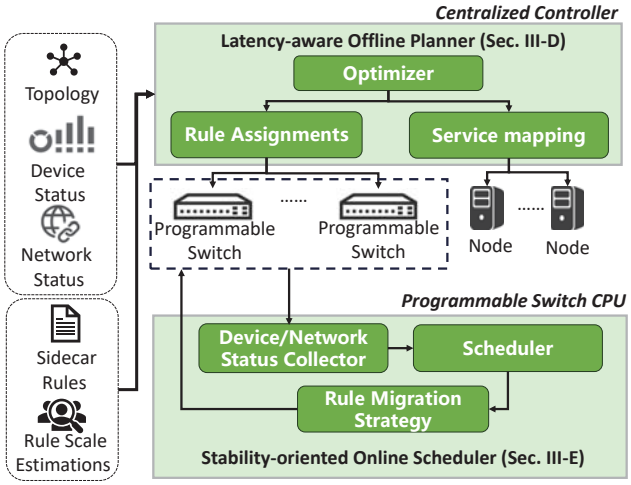


Fig. 3: The overview of PiggyCar.

Furthermore, we assessed the impact of network detours on switch queues and throughput under various request rates. Using the configuration from Section V, the Car Parking detection scenario sends 1.9Mb of image recognition data per request over 100Gbps inter-switch links, with requests evenly distributed across five nodes. The congestion control threshold is set at 359 packets according to DCTCP [13], [14]. Results show that when the single server-access link reaches about 47.8Gbps (25kRPS per server), nearly 100Gbps of traffic is generated at the core switch, quickly saturating the buffers (as shown in Fig. 2). This surge triggers congestion control, halving the TCP window and reducing application throughput by up to 53%.

III. Design

A. Key Idea and Overview

Key idea. To address the issues of the high resource consumption and increased latency in *distributed* sidecar deployments (Fig. 1(a)) and the network detouring, queue buildup, and severe throughput loss in *centralized* sidecar deployments (Fig. 1(b)), we propose a *piggybacked* sidecar deployment scheme, PiggyCar (Fig. 1(c)). PiggyCar maintains optimal communication links between pods, avoiding detours that contribute to congestion. It offloads time-consuming tasks, such as L7 fine-grained traffic control, to programmable switches that act as mandatory nodes along the optimal path. This strategy significantly minimizes resource consumption on the pods.

Fig. 3 provides an overview of PiggyCar, which comprises two modules: a latency-aware offline planner integrated with

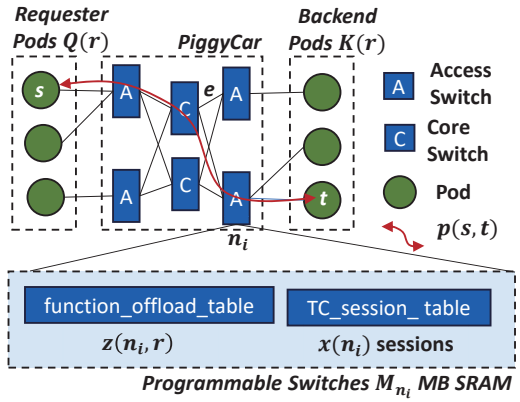


Fig. 4: The system model of service mesh architecture with PiggyCar.

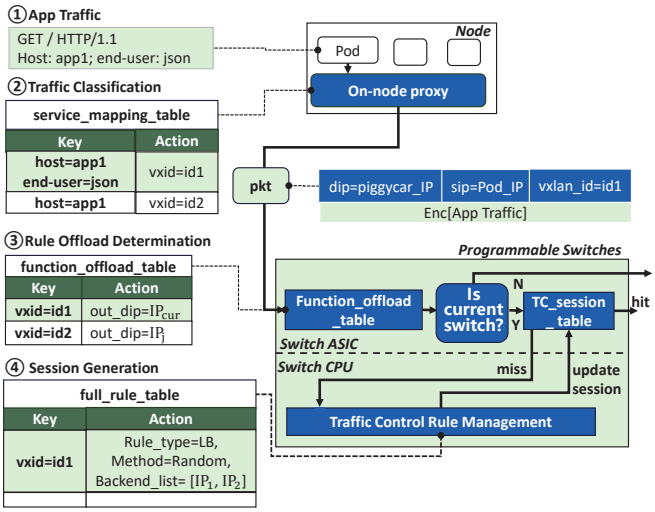


Fig. 5: The decoupled L7 traffic control.

the Istio [7] controller, and a scalability-oriented online scheduler on programmable switches that promptly adapts to traffic changes. Note that in this paper, our primary focus is on efficiently implementing the key traffic control functionality of sidecars on programmable switches. For the other two functions, namely zero-trust security and observability, we adopt an approach similar to Canal [9]: A dedicated key server generates asymmetric keys to reduce pod overhead, and observability is shifted from L7 to L4 to lower resource consumption while preserving essential monitoring.

The **latency-aware offline planner** computes optimal rule assignments for programmable switches. It takes system status (e.g., network topology and etc) and user-provided service mesh information (e.g., the estimated concurrent user count for each rule) as input. When deploying these rules, the planner considers on-chip SRAM limitations and prevents rule placements from forcing network detours between pods. Since this problem reduces to the NP-hard capacitated facility location problem [16], we employ a heuristic algorithm to find a near-optimal solution.

The **stability-oriented online scheduler** dynamically adjusts rule deployments when actual concurrency exceeds estimates. It makes minimal-overhead adjustments to maintain low latency while scaling to more users. This module uses switch session statistics, traffic measurements, and lightweight link data to determine a cost-effective configuration that minimizes delay.

B. System Model and Basic Notations

Fig. 4 shows the system model of service mesh architecture with PiggyCar, which forms a typical CLOS topology [22], [23] denoted by $G = (N, E)$. Where N comprises programmable switches and pods, and E represents the links. Each pod initially connects to an access switch that, in turn, connects to core switches, with no direct links between access switches or among core switches; this design follows standard industry practices [22], [23]. Each programmable switch n_i has a maximum SRAM capacity M_{n_i} and a current usage m_{n_i}. Given a set of rules R with an estimated session scale x_r per rule, each switch is allocated $x(n_i) = \sum_{r \in R} z(n_i, r) x_r$ rule sessions, $z(n_i, r)$ is a binary variable indicating whether rule r is allocated to switch n_i. The function_offload_table then maps the VXLAN_ID *id* to the appropriate switch IP based on this allocation (as we detailed in Section III-C).

For any two nodes (e.g., node s and node t) in the topology, the communication path $p(s,t)$ is derived using the underlay BGP protocol by appending the AS_PATH field to the routing information. Each pair of neighboring nodes establishes an eBGP relationship, so each node receives an AS_PATH ID that is visible at every hop during route exchange. This mechanism enables precise link information between nodes. We assume that when deploying a service mesh application, the requester pod group $Q(r)$ and the backend pod group $K(r)$ for each sidecar rule r can be identified. The communication relationship between these groups is obtained via orchestration logic similar to a service mesh workflow [24].

C. Decoupled L7 Traffic Control

Implementing advanced L7 traffic control (e.g., load balancing) on programmable switches is challenging. These devices are designed for fixed-length headers (e.g., IP/port) and cannot process variable-length, string-based protocols like HTTP due to hardware constraints [15]. Additionally, zero-trust security encrypts service mesh traffic, requiring decryption before any L7 parsing. This decryption process poses implementation challenges (e.g., matrix multiplication) on resource-constrained switches. To address the above challenges, we propose a *decoupled* L7 traffic control approach. The on-node proxy performs lightweight, rule-based L7 traffic classification, while programmable switches handle high-throughput traffic control.

Lightweight on-node proxy. Specifically, as shown in Fig. 5, the on-node proxy implements the table service_mapping_table. This table quickly matches key fields (e.g., *host* and *end-user*) in L7 traffic against user-specified sidecar rules and allocates an associated service ID, id_1 (steps ① and ②). Then, the VXLAN_ID in the outer header is

TABLE I: Input parameters of offline planner.

Variable Name	Symbol
Network topology	$G = \langle N, E \rangle$
Set of devices (pods and programmable switches)	N
Set of Ethernet links	E
Set of core switches	C
Maximum bandwidth capacity per link (Gbps)	$B_{\max}(e)$, where $e \in E$
Maximum SRAM capacity of programmable switches (MB)	M_{n_i}, where $n_i \in N$
Current utilized SRAM of programmable switches (MB)	m_{n_i}, where $n_i \in N$
Communication path set	$P = \{p(s,t)\}$, for $(s,t) \in N$
Individual communication path (set of programmable switches)	$p(s,t) = \{n_i\}$
Load-balancing (LB) rule set	R
Scale (count) of the rule	x_r, for $r \in R$
Traffic volume (Gbps) of the rule	y_r, for $r \in R$
Source pod	s, where $s \in N$
Destination pod	t, where $t \in N$
Requester pod group for a rule	$Q(r)$
Backend pod group for a rule	$K(r)$
Communication pod group pair set	$C(r) = \langle Q(r), K(r) \rangle$
Basic link latency	$T_{\text{base}}(e)$

replaced with id_1, which enables the programmable switch to perform traffic control based solely on this identifier.

Compared with traditional Envoy [25] (serving as the data plane of Istio [7]) that performs complete traffic control on the pod, our lightweight on-node proxy offers two major enhancements. First, it employs a concurrent parsing and matching strategy instead of waiting for the complete HTTP tree to be constructed, which reduces node-side processing latency by approximately 10.5%. Second, it offloads bandwidth-intensive tasks, such as load balancing, to the programmable switch, thereby further reducing node-side processing delays by approximately 77.2% (Section V-A).

Scalable programmable switch boosted traffic control. As shown in Fig. 5, there are two tables, function_offload_table and full_rule_table, allocated in the data plane and the control plane of the programmable switches. The function_offload_table is used to determine which switch should process the received packet and updates the outer destination IP with the matched IP (step ③). If the matched IP is local, TC_session_table (traffic control session table) is enabled to match. Otherwise, the packet is forwarded to the designated switch. When a session match is found, the tunnel's destination IP is updated to the backend service pod's IP. If no session match is found, the packet is sent to the switch CPU where the full_rule_table is used to identify the proper traffic control rule (step ④). Based on this rule, an appropriate backend service pod's IP is selected, and the session table in the switch's ASIC data plane is updated accordingly.

D. Latency-aware Offline Planer

In this subsection, we provide a detailed description of how to model the aforementioned fast service mesh communications based on the system model introduced earlier.

Algorithm 1: Latency-aware offline planner.

Input : Parameters in Table I.
Output: $z(n_i, r)$.

/*Define high-priority set S_{high} as the core and aggregation switches directly connected to all pods in $Q(r)$ and $K(r)$ */;
$S_{low} \leftarrow N \setminus S_{high}$;
foreach $r \in R$ **do**
  $S_{cand} \leftarrow S_{high} \neq \emptyset ? S_{high} : S_{low}$;
  foreach $n_i \in S_{cand}$ **do**
    $T_{n_i} \leftarrow$estimate_latency($Q(r)$, $K(r)$, n_i);
    T_{all}.append(T_{n_i});
  Find the best n_i^{best} with minimal latency from T_{all};
  $z(n_i^{best}, r) \leftarrow 1$;
  Update S_{high} or S_{low} where switches exceed M_{n_i};
Perform simulated annealing to avoid local optima;
procedure estimate_latency($Q(r)$, $K(r)$, n_i)
  Find the maximum latency T_{lb}^m from $Q(r)$ to n_i;
  Find the maximum latency T_{svc}^m from n_i to $K(r)$;
  return $T_{lb}^m + T_{svc}^m$;

1) Metrics Modeling: The optimization goal is to determine the optimal rule assignments $z(n_i, r)$ that minimize the maximum communication latency between any pod pairs C in the service mesh clusters. Let T denote the maximum latency among all pod group pairs C, defined as $T = \max_{r \in R, s \in Q(r), t \in K(r)} T(r,s,t)$, where $T(r,s,t)$ is the latency between the requesting pod s and the backend service pod t when the traffic control rule r is applied.

$$\min \quad T \tag{1}$$

The communication latency between any two pods can be decomposed into two parts: the latency $T_{LB}(r,s,t)$ from the requesting pod s to the switch that processes rule r, and the latency $T_{SVC}(r,s,t)$ from that switch to the backend service pod t. Then the latency can be computed as follows:

$$T(r,s,t) = T_{LB}(r,s,t) + T_{SVC}(r,s,t) \tag{2}$$

Given the forwarding delay $F(e,r)$ for the link e and the associated r, we can infer that $T_{LB}(r,s,t) = \sum_{e \in p(s,n_i)} F(e,r)$,and $T_{SVC}(r,s,t) = \sum_{e \in p(n_i,t)} F(e,r)$.

$$F(e,r) = \frac{T_{base}(e)}{1 - \rho(e,r)} \tag{3}$$

Where $T_{base}(e)$ denotes the basic RTT latency for the link e under rule r when there is no congestion. It can be obtained offline and remains relatively stable. The parameter $\rho(e,r)$ represents the bandwidth utilization ratio for the link e under rule r, and the latency of link e is inferred as $\rho(e,r) = \sum_{r \in R(e)} \frac{y_r}{B_{max}(e)}$ based on the M/M/1 queuing model [26]. Here, $R(e)$ denotes the set of rules that traverse link e, which is expressed as $R(e) = \{r \mid e \in E(r)\}$. The set $E(r)$ denotes the edges traversed by rule r and is derived as $E(r) = \bigcup_{s \in Q(r), t \in K(r)} \{p(s,n_i) \cup p(n_i,t)\}$, for all n_i such that $z(n_i, r) = 1$.

2) Solving the Problem: The above modeled rule assignment problem that involves in considering the communication latency (i.e., the cost of transmission) of pod pair and the SRAM constraints of each switch (i.e., the capacity of facility) on the road, can be reduced to the typical NP-hard problem Capacitated Facility Location Problem [16].

Therefore, we propose a heuristic algorithm to solve it. As shown in Algorithm 1, the overall algorithm comprises three steps. **1) Switch prioritization**. We prioritize the set of candidate switches for deployment. Switches that are more likely to reduce the inter-pod communication latency are given higher priority. If these high-priority switches cannot accommodate all the rules, the remaining switches are considered. The high-priority set includes the core switches and the aggregation switches that are directly connected to every pod in the pod group (lines 1 to 2). **2) Maximum latency calculation simplification.** For both high-priority and low-priority switch sets, the latency between each pair of pods is simplified to the sum of the maximum delays from two segments: the delay between the requester pod s and the deployed switch n_i, and the delay from that switch n_i to the backend service pod t (lines 12 to 15). This approximation reduces the computational complexity from $O(kn^2)$ to $O(2kn)$, where k represents the number of candidate switches, and n denotes the number of initiating and backend pods (assumed to be of similar scale). **3) Simulated annealing.** To avoid local optima, we employ a simulated annealing algorithm that randomly exchanges rules among the switches until either the desired latency reduction is achieved or a predetermined number of iterations is reached (line 11).

E. Stability-oriented Online Scheduler

Although the offline planner produces near-optimal rule assignments, the actual session count may deviate from estimates. Indeed, application providers struggle to accurately predict both the number of users and the generated traffic volume [27].

Therefore, we propose a stability-oriented online adaptation approach that runs on each switch. This method designs a lightweight rule migration scheme by accounting for deviations between observed values and the estimated session scale. It prevents input estimation errors from exceeding the switch's resource capacity, which would otherwise lead to unstable transmissions.

Trigger conditions. When the current programmable switch n_{cur} detects that the difference between observed values and the estimated session scale exceeds the predefined threshold ΔX, or that the utilized SRAM $m_{n_{cur}}$ is approaching the threshold T_{SRAM} (with T_{SRAM} set slightly below the maximum SRAM $M_{n_{cur}}$ to preempt overflow), the rule migration optimization algorithm is triggered. The trigger conditions are shown below:

$$\delta x(n_{cur}) > \Delta X \text{ or } (m_{n_{cur}} > T_{SRAM}). \tag{4}$$

$\delta x(n_{cur})$ denotes the sum of differences between the observed session amount and the estimated session scale for all rules assigned to switch n_{cur}. It is defined as $\delta x(n_{cur}) = \sum_{r\in R} z(n_{cur},r)[x_r^{ob} - x_r]$. Here, x_r^{ob} is the observed session count for rule r.

1) Application Level Metrics: The **goal** of finding the best migration strategy is defined as follows:

$$\begin{aligned} \min \quad & I = \alpha \cdot E_{mig} + \beta \cdot E_{reroute} \\ \text{subject to} \quad & \delta x^{new}(n_i) \le \Delta X \\ & m_{n_i} \le T_{SRAM} \end{aligned} \tag{5}$$

Where E_{mig} denotes the migration overhead and consists of two parts, the session migration latency and the rule migration latency. $E_{reroute}$ denotes the potential reroute overhead when applying a new rule assignment to the switches. $\delta x^{new}(n_i)$ denotes the updated differences of the observed session scale and the expectations for all switches of PiggyCar, after applying the above new rule assignment strategy $z^{new}(n_i,r)$. α and β are two adjustable parameters that balance the importance of the two overhead (In our experiments, we set $\alpha = 0.5$ and $\beta = 0.5$).

2) Metrics Modeling: **Constraints estimation.** For the constraint $\delta x^{new}(n_i)$ of all switches in PiggyCar, we derive using the formula $\delta x^{new}(n_i) = \delta x(n_i) + \sum_{r\in R} \Delta z(n_i,r)[x_r^{ob} - x_r]$, where $\delta x(n_i)$ denotes the difference (actual minus estimated) for all rules on switch n_i prior to migration, and $\Delta z(n_i,r) = z^{new}(n_i,r) - z(n_i,r)$ captures the change in rule assignment. For example, if rule r migrates from n_j to n_{j+1}, then $\Delta z(n_j,r) = -1$ and $\Delta z(n_{j+1},r) = +1$; for any other switch n_i, $\Delta z(n_i,r) = 0$. The session difference $[x_r^{ob} - x_r]$ for a migrating rule r remains unchanged and can be obtained from n_{cur}.

Thus, each switch in PiggyCar holds two types of rules: those selected by n_{cur} for migration and those that are not. For non-migrating rules, the assignment remains unchanged (i.e., $\Delta z(n_i,r) = 0$), so no additional session difference value is required. For migrating rules, $\Delta z(n_i,r)$ is provided by the online scheduler algorithm, while the session difference is observed at n_{cur}. Consequently, to infer $\delta x^{new}(n_i)$, each switch only needs to send a single data value representing the session difference for all its rules before performing the rule migration, i.e., $\delta x(n_i)$. While the current SRAM utilization m_{n_i} is also directly collected from other switches on n_{cur}.

Migration overhead E_{mig}. The migration overhead is determined by the slower of two operations, i.e., the session migration latency and the rule migration latency, expressed as:

$$E_{mig} = \max\{w_u, w_s\} \tag{6}$$

The rule migration is given by $w_u = \max_{n_d\in N} \ \delta_{n_d}(D_u)$, and the session migration latency is $w_s = \max_{n_d\in N^e} \ \delta_{n_d}(D_s)$, with $\delta_{n_d}(D) = \sum_{e\in P(n_{cur},n_d)} \frac{D}{B(e)}$. $B(e)$ denotes the rest of the bandwidth for the link e.

The data sizes are defined as $D_u = u \cdot \text{len}(R_\Delta^{full})$ and $D_s = \sum_{r\in R_\Delta^e} s \cdot x_r^{ob}$. Here, u is the number of bytes for each rule in the table function_offload_table, and s is the number of bytes for each session in the table TC_session_table. $R_\Delta^{full} = \{r \in R \mid \delta x(n_{cur},r) > 0\}$ denotes the set of rules whose session counts exceed the expectation, $\delta x(n_{cur},r)$ represents the difference between the observed and expected session scales. Note that

all observation data is obtained solely from the current switch n_{cur}, eliminating the need for additional data collection from other switches.

Reroute overhead $E_{reroute}$. We infer the maximum communication latency after rule migration as

$$E_{reroute} = \max_{r\in R, s\in Q(r), t\in K(r)} T^{new}(r,s,t) \tag{7}$$

Where each rule's new latency is given by $T^{new}(r,s,t) = T(r,s,t) + \Delta T(r,s,t)$. Here, $T(r,s,t)$ is the initial latency distributed during offline planning, and $\Delta T(r,s,t)$ represents the additional delay incurred after migration. In particular, $\Delta T(r,s,t)$ is decomposed as $\Delta T(r,s,t) = \Delta T_{LB}(r,s,t) + \Delta T_{SVC}(r,s,t)$, with the key update based on the difference between the actual delay observed on each switch edge and the estimated delay, denoted by $\Delta F(e,r)$. Specifically, for each rule r, the bandwidth utilization ratio change on link e is calculated as $\rho'(e,r) = \frac{\sum_{r\in R'(e)}\left(y_r^{ob} - y_r\right)}{B_{max}(e)}$, where $R'(e)$ denotes the set of rules that traverse link e after rule migration.

Broadcasting variation metrics for all rules incurs high overhead. To optimize, we focus on reroute latency E_{reroute}. Since prior work [28] indicates only 26% of applications involve large transfers (>1GB) that dominate delay, we limit synchronization to the top-k rules (R_k, e.g., k=20%). The latency for the remaining rules is updated using a constant ΔT^c, significantly reducing network traffic.

3) Solving the Problem Online: When the trigger for online rule scheduling is met, we propose the following heuristic algorithm to improve the solving speed. **1) Candidate switch selection:** select the top-U switches S_{candi} with the highest remaining SRAM, ensuring that their combined SRAM exceeds the migration requirement. **2) Rule deployment decision:** as shown in Algorithm 2, the core idea is as follows: for rules outside R_k, the detour overhead E_{reroute} is negligible, so we only consider the migration overhead E_{mig} (lines 7 to 8). Moreover, if the candidate switch has a different role than the current one, a one-hop transmission makes the migration overhead negligible (lines 3 to 4). For rules in R_k, if the candidate's role differs, deployment is allowed provided the detour delay does not increase (lines 5 to 6); otherwise, the candidate with the smallest overall metric I is chosen (lines 9 to 12).

Algorithm 2: Stability-oriented online planner.

Input : m_{n_i}, $\Delta F^k(e,r)$.
Output: $z^{new}(n_i,r)$.

```
1  foreach r ∈ R_Δ^full do
2      foreach n_i ∈ S_cand do
3          if r ∉ R_k and Role_diff(n_cur, n_i) then
4              z^new(n_i, r) ← 1; break;
5          else if r ∈ R_k and Role_diff(n_cur, n_i) and
               T^new(r,s,t) ≤ T^(r,s,t)) then
6              z^new(n_i, r) ← 1; break;
7          else if r ∉ R_k and not Role_diff(n_cur, n_i) then
8              I_m.append(E_mig(n_i, m_{n_i}));
9          else
10             I_a.append(I(n_i, m_{n_i}, ΔF^k(e,r)));
11     if z^new(,r) not set then
12         Find n_i with minimal I_m or I_a, z^new(n_i, r) ← 1;
```

IV. Implementation

We prototyped PiggyCar using a centralized scheduler and agents on both servers and switches to coordinate service mesh rule assignments and enforce online rule adaptation. Our implementation comprises over 4.5K lines of Python code for the switch control plane, more than 350 lines of P4 code for the programmable switch data plane, 3.4K+ lines of Python code running on the controller server, and over 2K lines of C++ code on the servers.

Central scheduler. It is implemented as a Python application that periodically aggregates static topology data (e.g., server configurations and switch capacities) and dynamic network metrics. When deploying new serverless applications, the scheduler optimizes and disseminates computed rule assignments and service mappings to programmable switch agents via high-speed gRPC.

From user-specified Istio rule configuration files, we extract only the matching field information. This extracted data is distributed to all on-node proxies to update their service_mapping_table entries, reducing the transmission volume by nearly 91%. Meanwhile, the complete configuration is sent solely to a limited number of programmable switches, significantly lowering the deployment overhead.

On-node proxy. It performs lightweight traffic matching. Based on the classification results, it leverages eBPF to encapsulate packets within a VXLAN tunnel and updates the VXLAN_ID to the service identifier corresponding to the targeted service mesh service.

Programmable switch. *1) Data plane*: It contains two core tables, i.e., fuction_offload_table and session_table. They are all implemented as exact-match table. We leverage the switch's built-in aging mechanism to automatically purge stale entries. This approach prevents long-inactive flows from occupying valuable SRAM resources, thereby ensuring that memory is effectively allocated to active traffic and maintaining overall system efficiency. *2) Control plane.* In addition to generating sessions to the switch data plane according to the stored service mesh rules, the control plane retrieves real-time fast-path session capacity data from the switch data plane to facilitate runtime rule migration and optimization strategies.

V. Evaluation

We evaluate PiggyCar using a combination of small-scale testbed experiments and large-scale simulations, and we compare its performance against two of the most relevant existing solutions, namely, Istio [7] and Canal [9].

Testbed deployment. As illustrated in Fig. 6, the testbed includes six servers and four programmable switches equipped with a Tofino ASIC [29]. One switch acts as the Core, while the other three serve as Access switches. Two Access switches each connect two servers via two 100Gbps links, the third

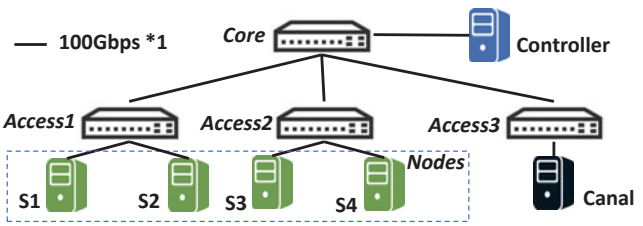


Fig. 6: Testbed.

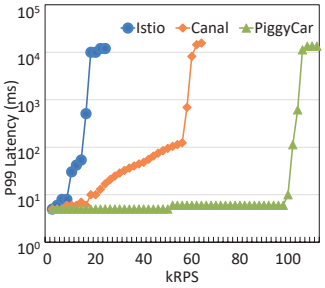


Fig. 7: [Testbed] P99 latency.

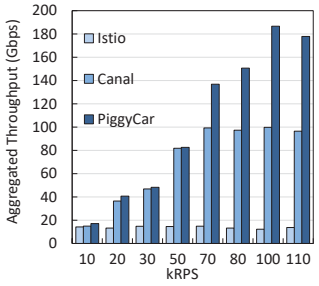


Fig. 8: [Testbed] Aggregated throughput.

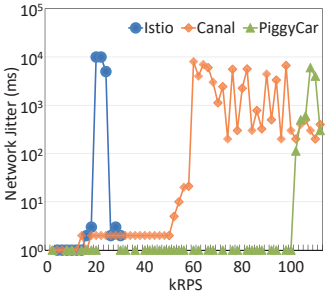


Fig. 9: [Testbed] Jitter.

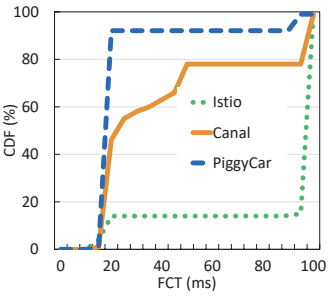


Fig. 10: [Testbed] Flow completion time.

one connects to Canal gateway [9]. Among the servers (each with a 96-core AMD CPU, a 100Gbps NIC, and 500GB of memory, 95Gbps+ AES encryption/decryption capability), four host service mesh nodes, one serves as the Canal gateway [9], and one runs the central scheduler as the controller.

Workload setup. We deploy a typical serverless scenario, i.e., car parking detection and charging [10], to examine the improvements of PiggyCar. This scenario uses parking spot snapshots to detect occupancy, extract and store vehicle license plate metadata if needed, and charge parking fees accordingly [10]. We use the CNRPark+EXT image dataset [30], every 240 seconds, 164 snapshots (350KB each) are sent to the function chain, resulting in a nearly 1.91Mbps data rate for each request. We vary the request from 10k request per seconds (RPS) to examine the performance benefits obtained by PiggyCar. The workload is generated evenly from all of the above five service mesh nodes. We retrieved sample traffic control rules from the Istio project [7] to generate 1,000 distinct rules for the car parking application and allocated concurrent accesses in a 2:8 ratio based on the RPS settings, thereby simulating a data center scenario where a small subset of flows consumes most of the traffic [28].

Simulation settings. We perform simulations on a physical server equipped with an Intel Core i9-9900K processor and an NVIDIA GeForce RTX 2080 Ti GPU. We utilize a high fielity network simulator NS-3 [31] to perform the large scale simulation. We simulate ten thousand programmable switches connecting ninety thousand servers and deployed one hundred thousand rules to validate PiggyCar's performance in large-scale scenarios.

Results reveal that:

- PiggyCar achieves the lowest latency: it reduces latency by up to 98.7% and 97.2% relative to Istio [7] and Canal [9] under high RPS, respectively.
- PiggyCar achieves highest throughput, it significantly improves the throughput by 15.1× and 1.82× compared to [7] and Canal [9].
- PiggyCar consistently produces stale network transmissions exhibiting the lowest network jitter and maintains the highest proportion of flows with minimal flow completion times.

A. Testbed Experiments

Latency. Fig. 7 compares latency performance at various kRPS in the testbed environment. At nearly 90kRPS, PiggyCar reduces latency by up to 98.7% and 97.2% relative to Istio [7] and Canal [9], respectively. This improvement is due to PiggyCar's efficient piggyback execution of service mesh policies, which prevents detour transmission among pods. We also conduct experiments to demonstrate that reducing HTTP tree analysis and eliminating complex load balancing can lower the average on-node latency from 764μs to 94μs, an 87.1% reduction.

In contrast, Canal [9] suffers from centralized gateway issues. At nearly 60k RPS, core-to-access3 traffic nears 100Gbps, even though access-to-server links average 50Gbps (50% utilization). This heavy load pushes core buffers to their limit, triggering ECN marking under DCTCP and causing the application layer to reduce its sending rate, which in turn increases request latency. Furthermore, Istio [7] shows increased CPU utilization at around 10kRPS due to its per-pod implementation, leading to higher per-packet processing delays.

Throughput. Fig. 8 presents aggregated throughput performance at varying request rates. Throughput was calculated by aggregating the processed requests across all pods. Experimental results show that at nearly 100kRPS, PiggyCar achieves improvements of up to 15.1× and 1.82× compared to the Istio [7] and Canal [9], respectively. This enhancement is attributed to PiggyCar's ability to avoid traffic detours and reduce redundant network transmissions, thereby lowering the probability of network congestion.

Jitter. Fig. 9 presents the average network jitter performance under various request rates, where jitter is defined as the absolute difference between the per-flow delay measurements and the average delay. The experimental results indicate that PiggyCar reduces network jitter by 98.9% and 97.9% compared to Istio [7] and Canal [9] at nearly 100kRPS, respec-

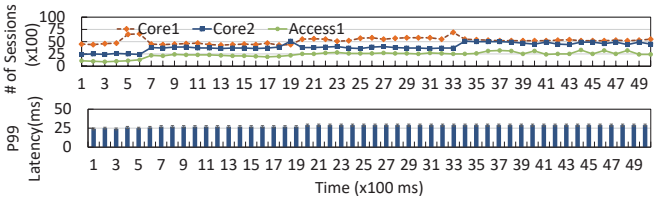


Fig. 11: [Simulation] The session scale and p99 latency when running online adaptation (40kRPS).

tively. When the RPS reaches around 10k, Istio [7] experiences significant jitter due to pod CPU exhaustion, which prevents packet forwarding. As for Canal [9], approaching 60kRPS triggers network congestion control, causing persistent and large fluctuations through continuous adjustments of the TCP transmission window. In contrast, PiggyCar significantly minimizes redundant network traffic and maintains stable performance even at high RPS.

Flow completion time. Fig. 10 presents the results for flow completion time (FCT). We tested three approaches, PiggyCar, Istio [7], and Canal [9], across 65 scenarios with request rates ranging from 1k to 110k RPS, with each scenario running for one minute. The FCT from all scenarios was aggregated into a distribution plot. Results indicate that PiggyCar increases the probability of achieving lower flow completion times by 6× and 1.9× compared to Istio [7] and Canal [9], respectively. Moreover, PiggyCar maintained flow completion times around 22ms in over 90% of cases, whereas Istio and Canal achieved such low delays in only approximately 15% and 46.1% of cases, respectively.

B. Simulation Experiments

Fig. 11 illustrates the changes in fast-path session specifications across several key switches, as well as the network-wide p99 latency, in a simulated environment running the runtime adaptive session migration algorithm. Results show that our proposed algorithm can promptly detect deviations from expected session scales and quickly adjust session migrations to maintain stable p99 latency. Specifically, at approximately 500ms, a surge in sessions occurred on the Core1 switch, persisting for 200ms. This anomaly was detected at 700ms, at which point the optimal migration plan was computed and executed: rules contributing to high p99 latency were migrated to the Core2 switch (which had greater remaining SRAM capacity and shorter reroute latency), while a few low-latency rules were transferred to the Access1 switch. As a result, the overall network p99 latency increased by only 1ms. Subsequently, similar session surges at 2000ms and 3400ms on the Core1 and Core2 switches were also promptly mitigated through timely session and rule migrations, ensuring that p99 latency remained consistently low.

VI. Related Work

Performance optimizations for service mesh frameworks. Existing efforts optimize service mesh by offloading functions to gateways [9], leveraging eBPF and shared memory [10], or enhancing scheduling and routing strategies [32], [33], [34], [35]. Others focus on hardware acceleration or architectural simplification [36], [37], [38], [39]. However, most approaches still rely on external gateways, per-pod proxies, or specialized hardware. In contrast, PiggyCar integrates sidecar functionality directly onto intermediary switches using piggyback parsing. This design eliminates the need for extra gateways or proxies, significantly conserving user pod resources.

Traffic control on programmable switches. Prior works utilize programmable switches for L7 [15], [40] or L4 [41], [42], [43], [44] load balancing. These solutions typically adopt a centralized switch cluster design, which overlooks the potential of switches as direct interconnects, leading to traffic detours and congestion. PiggyCar addresses this by employing a piggyback-style traffic control mechanism that maintains optimal communication paths between pod pairs, thereby reducing latency and avoiding redundant traffic.

VII. Conclusion and Future Works

This paper introduces PiggyCar, a *piggyback* service mesh communication system. In PiggyCar, resource-intensive and time-consuming modules are offloaded to intermediary network devices along inter-pod links, enabling on-the-fly execution of rich network functions such as Layer-7 load balancing. PiggyCar employs a latency-aware offline planner for network policy placement and a stability-oriented online scheduler to promptly adapt to runtime traffic fluctuations. We prototype PiggyCar on a testbed of six servers and four programmable switches. Experimental results show that PiggyCar reduces latency by up to 97.2% compared to the state-of-the-art Canal and improves throughput by 1.82× under high RPS (requests per second), while consistently achieving the lowest network jitter and the highest proportion of flows with minimal completion times.

In the future, there are several avenues to explore. First, we plan to enhance L7 observability on programmable switches to improve debugging and operational transparency. Second, we will implement encryption and decryption functions on programmable switches for deep payload parsing, thereby providing richer application-layer insights.

Acknowledgment

This work is supported by the National Key R&D Program of China (No. 2025YFE0204100), Science and Technology Development Fund of Macao S.A.R (FDCT) under number 0074/2025/AMJ, the National Natural Science Foundation of China (No. 92267105), Guangdong Basic and Applied Basic Research Foundation (No. 2023B1515130002), Key Research and Development and Technology Transfer Program of Inner Mongolia Autonomous Region (2025YFHH0110), Shenzhen Basic Research Program (No. JCYJ20250604183035046, No. JCYJ20220818101610023, No. KJZD20230923113800001), Ningbo Yongjiang Talent Project, U.S. NSF grants NSF-2421782, NSF-2350425, NSF-2319988, NSF-2206522, Microsoft Research Faculty Fellowship 8300751, Amazon research award.

REFERENCES

[1] AWS, "Aws app mesh," 2024. [Online]. Available: https://aws.amazon.com/app-mesh/

[2] Azure, "Azure service fabric," 2024. [Online]. Available: https://azure.microsoft.com/en-us/products/service-fabric

[3] Google, "Google cloud service mesh," 2024. [Online]. Available: https://cloud.google.com/products/service-mesh

[4] Alibaba, "Alibaba cloud service mesh," 2024. [Online]. Available: https://www.alibabacloud.com/en/product/servicemesh

[5] R. Duke, "How cloud-native architectures enable faster product releases in high-change environments," 2025. [Online]. Available: https://medium.com/@richarddukeusa/how-cloud-native-architectures-enable-faster-product-releases-in-high-change-environments-dff0f3229e51

[6] Flynn, "How service mesh can reduce the cost of running modern applications," 2022. [Online]. Available: https://www.spiceworks.com/tech/networking/guest-article/how-service-mesh-can-reduce-the-cost-of-running-modern-applications/

[7] Istio, "Istio: simplify observability, traffic management, security, and policy with the leading service mesh," 2025. [Online]. Available: https://istio.io

[8] Linkerd, "Linkerd: the world's most advanced service mesh," 2025. [Online]. Available: https://linkerd.io/

[9] E. Song, Y. Song, C. Lu, T. Pan, S. Zhang, J. Lu, J. Zhao, X. Wang, X. Wu, M. Gao, Z. Li, Z. Fang, B. Lyu, P. Zhang, R. Wen, L. Yi, Z. Zong, and S. Zhu, "Canal mesh: A cloud-scale sidecar-free multi-tenant service mesh architecture," in *Proceedings of ACM SIGCOMM*, 2024.

[10] S. Qi, L. Monis, Z. Zeng, I.-c. Wang, and K. K. Ramakrishnan, "Spright: extracting the server from serverless computing! high-performance ebpf-based event-driven, shared-memory processing," in *Proceedings of ACM SIGCOMM*, 2022.

[11] Q. Chen, J. Qian, Y. Che, Z. Lin, J. Wang, J. Zhou, L. Song, Y. Liang, J. Wu, W. Zheng, W. Liu, L. Li, F. Liu, and K. Tan, "Yuanrong: A production general-purpose serverless system for distributed applications in the cloud," in *Proceedings of ACM SIGCOMM*, 2024.

[12] Istio, "What is istio ambient mode," 2025. [Online]. Available: https://www.solo.io/topics/ambient/ambient-mode

[13] A. Dhamija, B. Madhavan, H. Li, J. Meng, S. Khare, M. Rao, L. Brakmo, N. Spring, P. Kannan, S. Sundaresan, and S. Ghorbani, "A large-scale deployment of dctcp: operational systems track," in *Proceedings of USENIX NSDI*, 2024.

[14] M. Alizadeh, A. Greenberg, D. A. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan, "Data center tcp (dctcp)," in *Proceedings of the ACM SIGCOMM*, 2010.

[15] X. Shi, L. He, J. Zhou, Y. Yang, and Y. Liu, "Miresga: Accelerating layer-7 load balancing with programmable switches," in *Proceedings of ACM WWW*, 2025.

[16] P. B. Mirchandani and R. L. Francis, *Discrete Location Theory*. Wiley, 1990. [Online]. Available: https://search.worldcat.org/title/19810449?oclcNum=19810449

[17] G. Antichi and G. Rétvári, "Full-stack sdn: The next big challenge?" in *Proceedings of ACM SOSR*, 2020.

[18] S. Ashok, P. B. Godfrey, and R. Mittal, "Leveraging service meshes as a new network layer," in *Proceedings of ACM HotNets*, 2021.

[19] J. a. T. Duarte Maia and F. Figueiredo Correia, "Service mesh patterns," in *Proceedings of EuroPLop*, 2023.

[20] M. Klein, "Lyft's envoy: Experiences operating a large service mesh," 2017. [Online]. Available: https://www.usenix.org/conference/srecon17americas/program/presentation/klein

[21] X. Zhu, W. Deng, B. Liu, J. Chen, Y. Wu, T. Anderson, A. Krishnamurthy, R. Mahajan, and D. Zhuo, "Application defined networks," in *Proceedings of ACM HotNets*, 2023.

[22] K. Qian, Y. Xi, J. Cao, J. Gao, Y. Xu, Y. Guan, B. Fu, X. Shi, F. Zhu, R. Miao, C. Wang, P. Wang, P. Zhang, X. Zeng, E. Ruan, Z. Yao, E. Zhai, and D. Cai, "Alibaba hpn: A data center network for large language model training," in *Proceedings of the ACM SIGCOMM*, 2024.

[23] A. Gangidi, R. Miao, S. Zheng, S. J. Bondu, G. Goes, H. Morsy, R. Puri, M. Riftadi, A. J. Shetty, J. Yang, S. Zhang, M. J. Fernandez, S. Gandham, and H. Zeng, "Rdma over ethernet for distributed training at meta scale," in *Proceedings of the ACM SIGCOMM*, 2024.

[24] Z. Li, Y. Liu, L. Guo, Q. Chen, J. Cheng, W. Zheng, and M. Guo, "Faasflow: enable efficient workflow execution for function-as-a-service," in *Proceedings of ACM ASPLOS*, 2022.

[25] Envoy, "Envoy is an open source edge and service proxy, designed for cloud-native applications," 2025. [Online]. Available: https://www.envoyproxy.io/

[26] A. Lazar, "The throughput time delay function of an m/m/1 queue (corresp.)," *IEEE Transactions on Information Theory*, vol. 29, no. 6, p. 914–918, 2006.

[27] S. Di, D. Kondo, and W. Cirne, "Host load prediction in a google compute cloud with a bayesian model," in *Proceedings of SC*, 2012.

[28] S. Eismann, J. Scheuner, E. v. Eyk, M. Schwinger, J. Grohmann, N. Herbst, C. L. Abad, and A. Iosup, "The state of serverless applications: Collection, characterization, and community consensus," *IEEE Transactions on Software Engineering*, vol. 48, no. 10, pp. 4152–4166, 2022.

[29] Intel, "Intel intelligent fabric processors," Online: https://www.intel.com/content/www/us/en/products/details/network-io/intelligent-fabric-processors.html, 2024.

[30] F. F. C. G. Giuseppe Amato, Fabio Carrara and C. Vairo, "Car parking occupancy detection using smart camera networks and deep learning," in *Proceedings of IEEE ISCC*, 2016.

[31] NS-3, "A discrete-event network simulator for internet systems, ns-3," 2025. [Online]. Available: https://www.nsnam.org/

[32] L. Wojciechowski, K. Opasiak, J. Latusek, M. Wereski, V. Morales, T. Kim, and M. Hong, "Netmarks: Network metrics-aware kubernetes scheduler powered by service mesh," in *Proceedings of IEEE INFOCOM*, 2021.

[33] H. C. J. H. M. H. C. C. K. P. Yi Hu, Haonan Ding, "Collaborative orchestration with probabilistic routing for dynamic service mesh in clouds," in *Proceedings of IEEE INFOCOM*, 2025.

[34] J. Park, J. Park, Y. Jung, H. Lim, H. Yeo, and D. Han, "Topfull: An adaptive top-down overload control for slo-oriented microservices," in *Proceedings of ACM SIGCOMM*, 2024.

[35] N. Zheng, T. Qiao, X. Liu, and X. Jin, "MeshTest: End-to-End testing for service mesh traffic management," in *Proceedings of USENIX NSDI*, 2025.

[36] F. Lu, X. Wei, Z. Huang, R. Chen, M. Wu, and H. Chen, "Serialization/deserialization-free state transfer in serverless workflows," in *Proceedings of ACM EuroSys*, 2024.

[37] N. Lazarev, S. Xiang, N. Adit, Z. Zhang, and C. Delimitrou, "Dagger: efficient and fast rpcs in cloud microservices with near-memory reconfigurable nics," in *Proceedings of ACM ASPLOS*, 2021.

[38] D. Saxena, W. Zhang, S. Pailoor, I. Dillig, and A. Akella, "Copper and wire: Bridging expressiveness and performance for service mesh policies," in *Proceedings of ACM ASPLOS*, 2025.

[39] S. Ashok, V. Harsh, B. Godfrey, R. Mittal, S. Parthasarathy, and L. Shwartz, "Traceweaver: Distributed request tracing for microservices without application modification," in *Proceedings of ACM SIGCOMM*, 2024.

[40] M. Scazzariello, T. Caiazzi, H. Ghasemirahni, T. Barbette, D. Kostić, and M. Chiesa, "A High-Speed stateful packet processing approach for tbps programmable switches," in *Proceedings of USENIX NSDI*, 2023.

[41] R. Miao, H. Zeng, C. Kim, J. Lee, and M. Yu, "Silkroad: Making stateful layer-4 load balancing fast and cheap using switching asics," in *Proceedings of ACM SIGCOMM*, 2017.

[42] D. Kim, J. Nelson, D. R. K. Ports, V. Sekar, and S. Seshan, "Redplane: Enabling fault-tolerant stateful in-switch applications," in *Proceedings of the ACM SIGCOMM*, 2021.

[43] C. Zeng, L. Luo, T. Zhang, Z. Wang, L. Li, W. Han, N. Chen, L. Wan, L. Liu, Z. Ding, X. Geng, T. Feng, F. Ning, K. Chen, and C. Guo, "Tiara: A scalable and efficient hardware acceleration architecture for stateful layer-4 load balancing," in *Proc. of USENIX NSDI*, 2022.

[44] Y. Feng, Z. Chen, H. Song, Y. Zhang, H. Zhou, R. Sun, W. Dong, P. Lu, S. Liu, C. Zhang, Y. Xu, and B. Liu, "Empower programmable pipeline for advanced stateful packet processing," in *Proceedings of USENIX NSDI*, 2024.