FlowLog: Byte-Level Flow Monitoring System in High-Throughput Networks

Long Chen, Mingwei Cui, Qiuheng Yin, Hanglong Lyu, Yisen Hong, Tong Yang, Yangyang Bai, Ziwei Zhao

***Abstract*—Gateways based on the programmable P4 language are becoming a key component in data center traffic management, offering cost-effective solutions for high-throughput environments. However, traditional monitoring techniques like sFlow and NetFlow lack the needed precision to meet the demands of large-scale data centers. In this paper, we present *FlowLog*, the first sketch-based and end-to-end flow monitoring system capable of accurate flow size estimation in 400 Gbps production environments. FlowLog integrates the novel *ByteSketch* algorithm, a transmission subsystem, and a high-speed analysis subsystem, achieving high accuracy even in demanding data center scenarios. Deployed for over six months in ByteDance's data center with peak bandwidths exceeding 400 Gbps, FlowLog outperforms existing solutions such as Bytehunter sFlow and state-of-the-art sketches in both accuracy and efficiency. Additionally, through real-world deployment, we gained valuable insights that guided improvements in system compatibility, integration ease, and traffic detection. These lessons resulted in a more adaptable system, better handling complex traffic patterns and ensuring minimal overhead during monitoring.**

***Index Terms*—Computer networks, data centers, computer network management**

I. Introduction

With the rapid advancement of cloud computing and the exponential growth of data center traffic, high-performance gateways have become the core infrastructure for modern cloud service providers. Data center gateways integrate complex network functions, including congestion control, load balancing, network address translation, and virtualized encapsulation. Byte-level metrics are indispensable for production operations such as bandwidth reservation, billing, fault diagnosis, and Top-K traffic statistics, which cannot be fulfilled by packet-level metrics. Unlike packet-per-second that rarely bottlenecks, byte-per-second directly reflects bandwidth utilization and is the key constrained resource, demanding high-precision byte-level measurement. These result in large-scale, highly dynamic, and skewed traffic patterns that pose stringent requirements for high-precision traffic monitoring [1], [2].

This work was supported in part by the National Key R&D Program Project under Grant 2024YFE0203700, in part by the National Key Research and Development Program of China under Grant 2024YFB2906602, and in part by the National Natural Science Foundation of China (NSFC) under Grant 62372009. *(Corresponding author: Tong Yang).*

L. Chen and M. Cui contributed equally to this research.

L. Chen, M. Cui, Y. Bai, Z. Zhao and Z. Li are with Douyin Vision Co., Ltd., Beijing 100041, China. Q. Yin, H. Lyu, Y. Hong, and T. Yang are with the School of Computer Science, Peking University, Beijing 100871, China. E-mail: longchen.cs@ieee.org; cuimingwei@bytedance.com; yinqiuheng@stu.pku.edu.cn; baiyangyang.1208@bytedance.com; zhaoziwei.05@bytedance.com; lyuhanglong@stu.pku.edu.cn; eason18@pku.edu.cn; yangtong@pku.edu.cn.

Programmable P4 gateways (P4GWs), built on the Protocol Independent Switch Architecture (PISA), have emerged as the dominant solution for modern data center gateways, offering a cost-effective balance of forwarding performance, flexibility, and scalability compared to traditional x86-based software gateways and fixed-function ASICs [3], [4], [5]. However, production-grade traffic monitoring on P4GWs faces fundamental, unresolved challenges. Existing monitoring solutions fall into three broad categories, each with critical limitations in 400+ Gbps high-throughput production environments:

1) *Sampling-based solutions* (e.g., sFlow [6], NetFlow [7]) are widely deployed for their low resource overhead, but suffer from inherent accuracy limitations. For small flows (e.g., a flow with 200 packets), the correct reporting probability is only $\frac{200}{N}$ (where N is the total number of packets processed), and accuracy degrades rapidly as line rates increase.

2) *Sketch-based per-flow measurement solutions* provide provable accuracy guarantees with minimal memory footprint, but are predominantly optimized for packet-level counting. When applied to byte-level measurement (the core requirement for production traffic accounting), these algorithms face severe bucket overflow and estimation bias under the strict SRAM and pipeline stage constraints of PISA architectures [8]. Classic Count-Min Sketch [9] and Conservative-Update Sketch [10] fail to distinguish between large and small flows, leading to diminished accuracy due to hash collisions, while state-of-the-art TowerSketch [11] lacks targeted optimizations for byte-level counting in high-throughput scenarios.

3) *End-to-end sketch-based monitoring systems* (e.g., OmniMon [12], FlowRadar [13], LruMon [14]) integrate sketch counting with flow ID capture, but lack production-grade designs for high-throughput environments. These systems either introduce prohibitive monitoring bandwidth overhead (up to 20% of business traffic in our tests), fail to resolve control/data plane clock asynchrony and synchronization errors [15], or rely on the unrealistic assumption of pre-known flow IDs [9], [11], [10], [12]. Furthermore, they do not address the data skew challenge in real-time stream processing, leading to excessive latency and unstable performance in production deployment.

Through 6 months of production deployment and iterative optimization in ByteDance's 400 Gbps data center networks, we identify *three fundamental but unaddressed challenges* that prevent existing solutions from meeting the requirements of production-grade high-throughput P4GW monitoring: **(Challenge 1)** Memory-constrained byte-level estimation under PISA hard constraints. Existing sketch algorithms cannot simultaneously satisfy the single-stage atomic update requirement, strict SRAM budget, and high-accuracy byte counting,

leading to either severe bucket overflow or excessive resource consumption. **(Challenge 2)** Low-overhead synchronization between flow IDs and sketch statistics. Existing flow ID capture schemes face an inherent trade-off between monitoring bandwidth overhead and synchronization accuracy, with no robust solution to resolve control/data plane clock asynchrony and cycle misalignment in high-throughput environments. **(Challenge 3)** Skew-resilient real-time flow statistics reconstruction. Generic stream processing frameworks cannot efficiently handle the extreme memory asymmetry between large-scale bursty flow IDs and small-sized sketch data, leading to severe data skew, excessive end-to-end latency, and unstable long-term operation.

To address these challenges, we propose *FlowLog*, the first end-to-end sketch-based flow monitoring system that achieves high-precision byte-level flow measurement in 400 Gbps production P4-programmable gateway environments with minimal overhead. FlowLog is built on three tightly coupled, innovatively designed core components: a **device-local ByteSketch** (Section III), a novel byte-counting customized sketch algorithm that eliminates bucket overflow under strict PISA hardware constraints via shift compression and probabilistic compensation, with a provable error bound for flow size estimation and line-rate atomic update support in a single pipeline stage; an **efficient transmission subsystem** (Section IV) based on the KeyWatcher/KeyReceiver framework, which captures flow IDs with ultra-low bandwidth overhead and resolves control/data plane synchronization errors via a tag-based sketch ID mechanism; and a **scalable skew-resilient analysis subsystem** (Section V), which leverages the memory asymmetry between flow IDs and sketch data, adopts broadcast-based job assignment and TTL-based join optimization to eliminate data skew, and supports real-time high-throughput flow processing with bounded end-to-end latency and reliable zero-data-loss operation in long-term production deployment.

We formalize the end-to-end flow monitoring problem as a joint optimization problem of estimation accuracy, resource efficiency, and system reliability, and decompose it into three sub-problems that directly map to the above three components. This formalization provides a rigorous academic anchor for our design, and ensures that each component is justified by a well-defined problem, rather than ad-hoc engineering implementation. The main contributions are as follows:

1) We designed and implemented a comprehensive end-to-end solution called FlowLog that integrates ByteSketch for byte-level flow size estimation, ensuring high accuracy even in high-throughput environments. The system combines a device-local sketch for per-flow byte counting, an efficient transmission subsystem for flow statistics, and a scalable real-time analysis subsystem for scalable monitoring in production environments.
2) We deployed our FlowLog for more than six months in ByteDance's data center, where it handled a peak bandwidth of 400 Gbps. During this time, we compared FlowLog with existing solutions like Bytehunter, a DPDK-based sFlow solution. FlowLog consistently outperformed these systems in both accuracy and efficiency, demonstrating its suitability for high-throughput production environments and highlighting its potential in real-world applications.
3) We distill generalizable, field-validated design principles and lessons learned from large-scale production deployment, covering system integration, parameter tuning, compatibility optimization, and false positive mitigation. These insights provide actionable guidance for the design and deployment of programmable data plane measurement systems in real-world data center environments.

The remainder of this paper is structured as follows: Section II provides an overview of the FlowLog system. Sections III through V present the design details. In Section VI, we evaluate the performance of FlowLog, while Section VII reviews related work. Finally, we discuss the lessons learned in Section VIII and conclude the paper in Section IX.

II. Overview

FlowLog consists of three main components, as shown in Fig. 1, the device-local sketch, the transmission subsystem, and the analysis subsystem. These subsystems are designed to work together seamlessly to estimate the number of packets and the total byte count of *all* or interested flows passing through the P4GW. FlowLog is built for excellent scalability, allowing for easy integration into existing P4 pipelines with minimal impact on the existing business traffic.

Device-local sketch. The device-local sketch is deployed directly on the P4GW and is responsible for collecting flow-level statistics. This component uses the ByteSketch algorithm to process each packet passing through the pipeline of the P4GW. It maintains compact data structures that update packet and byte statistics for interested flows. This subsystem ensures precise flow-level statistics without sampling, enabling efficient flow monitoring.

Transmission subsystem. The transmission subsystem is deployed across both the P4GW control plane and a server running DPDK. It is responsible for preparing and sending flow statistics, along with flow IDs, from the device-local sketch to the analysis subsystem. Specifically, the local controller on the P4GW retrieves the sketch records and periodically sends them to the analysis subsystem. The KeyWatcher component, deployed on the data plane, identifies each newly arrived flow and mirrors a packet associated with that flow for further processing. These mirrored packets, along with the corresponding flow ID, are sent to the KeyReceiver on the DPDK-enabled server, where they are aggregated and batched to reduce transmission load before being forwarded to the analysis subsystem.

Analysis subsystem. The analysis subsystem consists of a distributed storage module and a streaming analysis module. It collects and analyzes flow statistics from the sketches and flow IDs. The distributed storage module temporarily stores the incoming flow data, while the streaming analysis system continuously pulls updates from the storage. It processes the data by matching flow IDs with corresponding statistics, such as the byte and packet count from ByteSketch. The results are then stored in a long-term repository for further analysis and visualization, such as through monitoring dashboards.

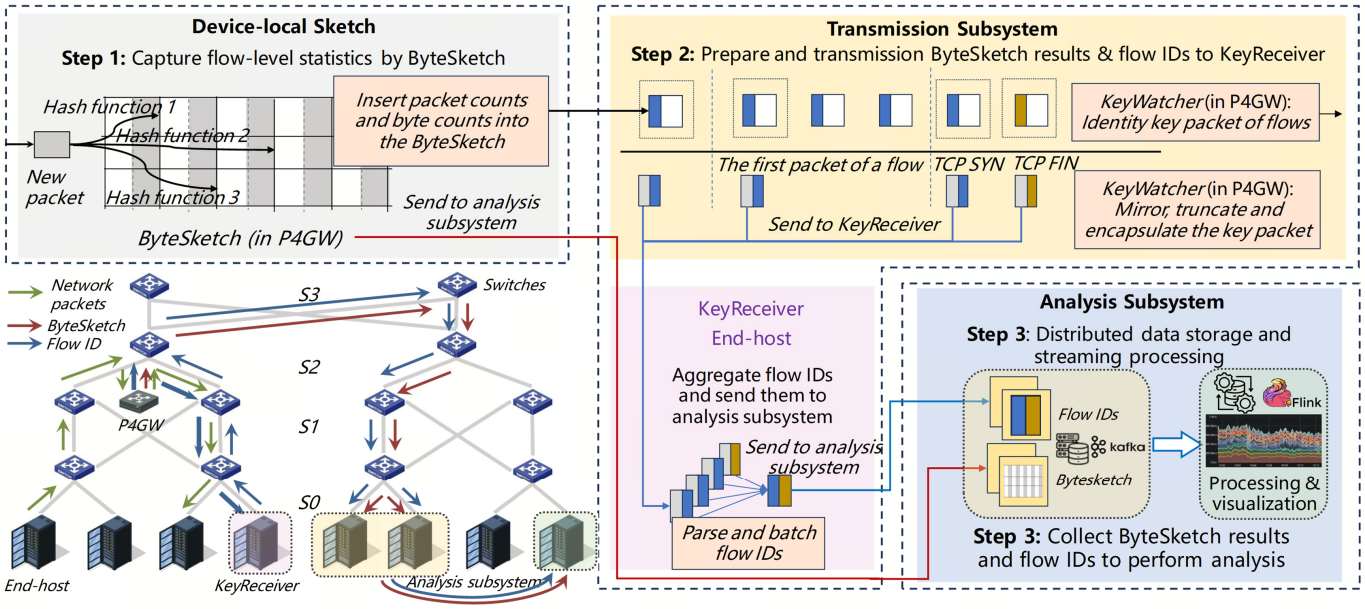


Fig. 1. An overview of the FlowLog system. Three key components: 1) device-local sketch on the P4GW, 2) transmission subsystem, including the KeyWatcher on the P4GW and the KeyReceiver on the DPDK-enabled server, and 3) analysis subsystem, including the Kafka-based distributed data storage module and Flink-based streaming processing module.

Putting everything together. These three key components work in coordination to achieve accurate traffic monitoring. The device-local sketch captures flow-level statistics, which are then passed to the transmission subsystem. The transmission subsystem identifies new flows, records the flow IDs, mirrors a packet for each new flow, and sends the data to the analysis subsystem, where it is stored, processed, and analyzed. The results are then made available for further use, ensuring efficient and scalable traffic monitoring with minimal impact on the existing business traffic.

Problem Formulation. We formulate the byte-level flow monitoring problem as a joint optimization problem of estimation accuracy, resource efficiency, and system reliability. Specifically, the problem aims to minimize per-flow byte size estimation error, with joint consideration of strict P4GW hardware resource constraints, limited monitoring bandwidth budget, and bounded end-to-end processing latency. It simultaneously guarantees line-rate packet processing and non-intrusive integration with production business traffic in each statistical window T.

The decision variables include: 1) ByteSketch configuration parameters: number of arrays d, bucket count per array w_i, and right-shift factor k_i for the i-th array ($\forall i \in [1, d]$); 2) binary synchronization indicator $x_{f,T} \in \{0, 1\}$, where $x_{f,T} = 1$ implies the flow ID of flow f is correctly captured and matched with sketch statistics in window T; 3) binary scheduling indicator $y_{f,T} \in \{0, 1\}$, where $y_{f,T} = 1$ implies flow f is processed by the analysis subsystem within the latency bound. The optimization problem is defined as:

$$\textbf{minimize} \quad \frac{1}{|\mathcal{F}_T|} \sum_{f \in \mathcal{F}_T} \frac{|\hat{a}_{f,T} - a_{f,T}|}{a_{f,T}}, \tag{1}$$

$$\text{s.t.} \quad (2) - (6),$$

$$M(d, \{w_i\}) \le M_{\max}, \quad S(d) \le S_{\max}, \forall i \in [1, d], \tag{2}$$

$$B_{\text{mon}}(T) \le 0.01 \cdot B_{\text{total}}(T), \tag{3}$$

$$D_{f,T} \le D_{\max}, \forall f \in \mathcal{F}_T, \tag{4}$$

$$x_{f,T} \le y_{f,T}, \forall f \in \mathcal{F}_T, \tag{5}$$

$$\sum\nolimits_{T' \in \mathcal{T}} x_{f,T'} \ge 1, \forall f \in \mathcal{F}_T, \tag{6}$$

where $\mathcal{F}_T$ denotes the set of all flows traversing the P4GW in window T, $a_{f,T}$ and $\hat{a}_{f,T}$ are the ground-truth and estimated byte count of flow f, respectively. The constraints are interpreted as follows: (2) defines P4GW hardware constraints: the total SRAM usage $M(\cdot)$ of ByteSketch and pipeline stage occupation $S(\cdot)$ must not exceed the maximum available resources of production gateways. (3) enforces the monitoring bandwidth budget: monitoring traffic must not exceed 1% of total business traffic bandwidth, avoiding competition with production services. (4) specifies the end-to-end latency bound: processing latency of each flow must be within the upper limit $D_{\max}$ to guarantee real-time monitoring. (5) indicates that $x_{f,T}$ can be set to 1 only when flow f is scheduled and processed by the analysis subsystem, i.e., $y_{f,T} = 1$. (6) guarantees the completeness of flow statistics: for long-lived flows spanning multiple statistical windows, their total byte counts are split and aggregated per window, while short-lived flows are guaranteed to be captured in at least one time slice.

To solve the core optimization problem, we decompose it into three tightly coupled sub-problems, each directly mapping to a core component of FlowLog and a dedicated design section in this paper:

Sub-Problem 1: Memory-Constrained Byte-Level Flow Size Estimation (addressed in Section III)

Under the P4GW hardware constraints (2) and line-rate processing requirement, design a sketch data structure for per-flow byte counting, which: (1) eliminates bucket overflow for byte-level updates under limited SRAM memory; (2) provides

a provable error bound for flow size estimation; (3) supports single-stage atomic updates compliant with the PISA pipeline architecture; (4) achieves significantly lower estimation error than state-of-the-art sketches under the same memory budget.

Sub-Problem 2: Low-Overhead Flow ID and Sketch Statistics Synchronization (addressed in Section IV)

Under the peak packet arrival rate λ and bandwidth constraint (3), design a transmission mechanism that: (1) captures flow IDs with minimal packet mirroring overhead; (2) eliminates synchronization errors caused by control-plane/data-plane clock asynchrony; (3) resolves the cycle misalignment between flow ID capture and sketch statistics; (4) guarantees the flow ID-sketch matching accuracy.

Sub-Problem 3: Skew-Resilient Real-Time Flow Statistics Reconstruction (addressed in Section V)

Under the end-to-end latency constraint (4) and millions of flow IDs per second input, design a distributed stream processing system that: (1) performs efficient join between flow IDs and sketch statistics to reconstruct per-flow byte counts; (2) mitigates data skew caused by bursty flow arrival rates; (3) minimizes compute and memory resource consumption; (4) ensures zero data loss and stable long-term operation in production environments.

III. Device-Local Sketch

This section addresses **Sub-Problem 1** defined in Section II. We first analyze the fundamental limitations of state-of-the-art sketches in production byte-level monitoring, then present the design, algorithmic implementation, and theoretical guarantee of our proposed ByteSketch.

Achieving high-precision traffic statistics on programmable switch platforms is challenging, especially in production environments with large-scale, dynamic traffic. Sketch-based solutions offer a memory-efficient way to estimate flow statistics; among them, Count-Min (CM) Sketch is a widely used technique that balances precision and resource usage on switches.

Count-Min Sketch for packet counting. At its core, CM Sketch is implemented as several parallel arrays of buckets (often called a two-dimensional array). Each array, corresponding to a different hash function, contains buckets of a fixed width. When a packet arrives, each hash function maps its flow ID to one bucket per array, and the bucket's value is incremented. The multi-array structure helps CM Sketch efficiently handle skewed flow-size distributions, and simple bitwise updates make it lightweight enough to run at line rate.

Limitations of CM Sketch in production environments. While CM Sketch works well for packet counts, we found serious issues when using it for byte counting in our production deployment, which make it unable to meet the requirements of Sub-Problem 1: *Bucket overflow*: Byte counts grow much larger than packet counts, causing narrow buckets (e.g., the 8-bit bucket) to overflow. In our real traffic, nearly 95% of 8-bit buckets were unusable. *Trade-offs of larger buckets:* Simply increasing bucket size reduces overflow but also reduces the total number of buckets in memory, leading to more hash collisions and degraded precision.

Theoretically, this inefficiency stems from the fundamental limitation of CM Sketch for byte counting: its error bound is proportional to the total sum of encoded values, meaning encoding raw full-size byte values inevitably leads to looser error guarantees and faster bucket saturation compared to compressed smaller values. Recent works have explored quantization and vectorization approaches for finer-grained per-flow measurement: HistSketch [16] enables per-key distribution monitoring via hot-cold separation and histogram shedding, while SketchFeature [17] proposes sketch virtualization for high-quality 3rd-order per-flow feature extraction in the data plane. Both works support aggregated flow byte volume calculation through bin-level counting, but they are not optimized for high-throughput byte-level monitoring in production programmable switches: HistSketch relies on offline server-side decoding and cannot support real-time data-plane queries, while SketchFeature introduces non-negligible memory and pipeline overhead for pure byte count statistics due to its multi-bin virtual sketch design.

Design Philosophy. To address the above limitations and solve Sub-Problem 1, we introduce two core design ideas that are fully compliant with PISA pipeline constraints: *Bit shifting for overflow mitigation:* Before updating buckets, we right-shift (compress) each packet's byte length by a configurable number of bits. This reduces the magnitude of values stored in buckets, greatly lowering the overflow rate under the same memory budget. *Probabilistic compensation:* Because bit shifting may introduce low-order bit losses, we correct for this by probabilistically adding back fractional amounts. This compensation is important for restoring the unbiased estimation of flow statistics.

Unlike hierarchical tree-based sketches such as FCM [18] and Count-Less [19] that rely on counter overflow cascading to record traffic statistics, our design decouples overflow mitigation from the underlying counter structure, enabling a PISA pipeline-friendly implementation without inter-stage state dependency. This design strictly satisfies the single-stage atomic update constraint of P4 programmable switches, a mandatory requirement for 400 Gbps line-rate processing.

Data Structure: Our ByteSketch consists of d arrays, i.e., $A_1, A_2, \ldots, A_d$, fully aligned with the notation defined in Section II. Each array A_i contains w_i buckets and is associated with a uniform hash function $h_i : \mathcal{F} \to [1, w_i]$ and a shift number k_i. The capacity of each bucket within array A_i is measured in δ_i bits. Equal memory is allocated to each array to balance hash collision probability across all layers.

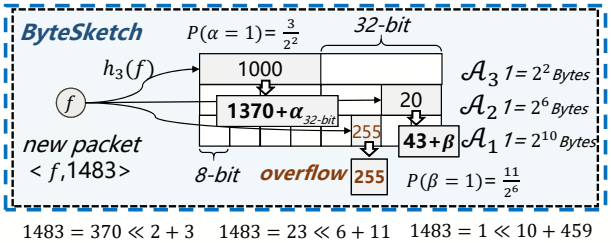


Fig. 2. Data structure of ByteSketch.

Example: As shown in Fig. 2, the bottommost array is equipped with 8 buckets, each possessing 8 bits; conversely, the uppermost array comprises 2 buckets, each with 32 bits.

Collectively, the three arrays utilize an identical memory footprint of 64 bits.

Algorithmic Implementation. We formalize the two core operations of ByteSketch in Algorithm 1 (Insert) and Algorithm 2 (Query), which are fully compliant with the P4 data plane's single-cycle atomic update constraints.

Algorithm 1 ByteSketch Insert Operation

Input: Flow ID f, packet byte count a_j, ByteSketch parameters $\{d, w_i, k_i, \delta_i, h_i(\cdot)\}_{i=1}^{d}$
Output: Updated ByteSketch arrays $A_1, ..., A_d$

for $i = 1$ **to** d **do**
  $idx \leftarrow h_i(f)$
  $n \leftarrow \lfloor a_j / 2^{k_i} \rfloor$, $r \leftarrow a_j \bmod 2^{k_i}$
  $n \leftarrow n+1$ with probability $\frac{r}{2^{k_i}}$ {Probabilistic compensation}
  if $A_i[idx] + n < 2^{\delta_i} - 1$ **then**
    $A_i[idx] \leftarrow A_i[idx] + n$
  else
    $A_i[idx] \leftarrow 2^{\delta_i} - 1$ {Mark overflow}
  end if
end for

Algorithm 2 ByteSketch Query Operation

Input: Flow ID f, ByteSketch parameters $\{d, w_i, k_i, h_i(\cdot)\}_{i=1}^{d}$
Output: Estimated byte count $\hat{a}_f$

Initialize value set $V \leftarrow \emptyset$
for $i = 1$ **to** d **do**
  $idx \leftarrow h_i(f)$
  if $A_i[idx] < 2^{\delta_i} - 1$ **then**
    Add $A_i[idx] \times 2^{k_i}$ to V
  end if
end for
$\hat{a}_f \leftarrow \begin{cases} \min(V) & V \neq \emptyset \\ +\infty & \text{otherwise} \end{cases}$
return $\hat{a}_f$

Insertion Details. We propose a strategy that combines *data compression* with *probabilistic compensation*. Specifically, we use bit shifting to compress the raw byte values before inserting them into the sketch. This compression reduces the likelihood of bucket overflow by limiting the size of the values that are stored. Alongside the compression, we apply a probabilistic compensation mechanism to adjust the recorded counts, ensuring that the estimates more accurately reflect the actual byte counts after decompression.

More specifically, ByteSketch first calculates the value n to be added to the buckets, which is given by $n = \frac{a_j}{2^{k_i}}$, where a_j is the raw byte value. The remainder r is then computed as $r = a_j \mod 2^{k_i}$. The value n is added to the corresponding bucket indexed by the hash function $h_i(\cdot)$. To maintain the unbiased property of the sketch, ByteSketch adds an additional 1 to the bucket with a probability of $\frac{r}{2^{k_i}}$.

If a δ-bit bucket reaches its capacity after the update, it is marked as overflowed by setting its value to $2^{\delta_i} - 1$, which represents the maximum value that a δ -bit bucket can hold. Essentially, the largest value that a δ-bit bucket in line i can store is $2^{\delta_i} - 2$. Once a bucket is overflowed, it is considered to have an infinite value, preventing any further updates.

Example: An example of the insertion process is shown in Fig. 2. ByteSketch first calculates $n = \frac{1483}{4} = 370$ and $r = 1483 \mod 4 = 3$, and then adds the value of 370 to the bucket indexed by the hash function, starting from 1000 and incrementing up to 1370. Finally, with a probability of $\frac{3}{4} = 0.75$, ByteSketch adds 1 to bucket 1370 to compensate for the statistical error introduced during value compression.

Query Details: To determine the byte count for a flow, the query operation retrieves the smallest value among the d hashed buckets. The value from the bucket is then multiplied by 2^{k_i} to obtain the final byte count. Any bucket that has reached its maximum capacity is considered to have a value of $+\infty$.

Theorem 1. *(Error bound) Given a small positive real number ε, the probability that the estimated flow size $\hat{a_j}$ exceeds the actual flow size a_j plus $\varepsilon\|a\|$ is bounded as follows:*

$$Pr(\hat{a_j} > a_j + \varepsilon||a||) \leq \prod_{j=0}^{n} \frac{2^j}{e}, \tag{7}$$

where $\|a\| = \sum_{i=1}^{n} a_i$ and e is the base of the natural logarithm. For more details, please refer to Appendix A.

This theorem provides a rigorous theoretical guarantee for the estimation accuracy of ByteSketch, proving that it meets the error minimization objective of Sub-Problem 1 under the given memory constraints.

IV. Transmission Subsystem

This section addresses **Sub-Problem 2** defined in Section II: low-overhead flow ID and sketch statistics synchronization under the bandwidth constraint (3). We first analyze the fundamental limitations of existing flow ID capture and sketch synchronization schemes in high-throughput production environments, then present the design of our KeyWatcher/KeyReceiver framework, read-write separation mechanism, and tag-based synchronization strategy.

A. Preparation and Transmission of Flow ID

Existing flow ID capture solutions face an inherent trade-off between synchronization accuracy and bandwidth overhead in high-throughput P4GW environments. Full packet mirroring schemes (e.g., Everflow [20]) introduce prohibitive bandwidth consumption (up to 20% of business traffic in our pre-deployment tests), while classic sketch-only systems (e.g., CM Sketch [9], TowerSketch [11]) rely on pre-known flow IDs and cannot reconstruct per-flow statistics in real-world production pipelines. Recent works like FlowRadar [13] and OmniMon [12] integrate flow ID recording with sketch counting, but require end-host collaboration or introduce excessive data plane memory overhead, making them unsuitable for 400 Gbps gateways with strict resource constraints.

We developed the KeyWatcher, which consists of several key components: a Bloom Filter and a combination of two access control list rules, `bf_acl` and `mirror_acl`. The key is to reduce the impact of duplicate packets on existing traffic.

The KeyWatcher begins with `bf_acl`, which serves as a filter to determine which flows should be passed into the Bloom Filter. It handles the forwarding of all UDP traffic and is also used to detect abnormal TCP terminations or suspended

TCP transmissions. These scenarios typically involve a TCP flow where the initiation (SYN) packet has been sent but the termination (FIN) packet has not yet been received, which indicates a potential issue with long-lived connections.

After that, the Bloom Filter checks whether the flow is new by evaluating its state. If the flow has not been encountered before, the Bloom Filter updates its state, effectively marking the flow as a new arrival. After a new flow is identified, the packet is evaluated by `mirror_acl`, which determines whether it should be forwarded for further processing. Specifically, `mirror_acl` allows all UDP packets and TCP packets with SYN or FIN flags to pass through. Only if the packet passes this `mirror_acl` check, a new packet containing the associated flow ID is prepared and sent to the KeyReceiver. The KeyReceiver then handles the flow's state and characteristics for subsequent processing and analysis.

Instead of processing or mirroring every packet within a flow, the KeyWatcher focuses solely on newly arrived flows, minimizing the volume of traffic that needs to be mirrored or analyzed. Specifically, it sends a minimal number of packets, such as the flow ID packet and, when necessary, mirrors key packets like SYN and FIN, which are essential for managing the flow's lifecycle. This targeted approach reduces the pressure on the network and ensures that only the most important packets are processed, preventing unnecessary duplication of traffic and preserving system performance. Please refer to Algorithm 3 for more details.

Algorithm 3 KeyWatcher Flow ID Capture

Input: Incoming packet pkt, flow ID f, ACL rules, Bloom Filter BF, sketch ID sid
Output: Mirrored flow ID packet (if applicable)

```
if pkt passes bf_acl check then
    if f ∉ BF then
        Add f to BF
        if pkt passes mirror_acl check then
            Encapsulate f and sid into mirrored packet
            Send packet to KeyReceiver
        end if
    end if
end if
```

Scalability bounds: The KeyReceiver is implemented on a DPDK-enabled server, with its maximum supported P4GW count bounded by $\frac{\text{DPDK server's peak pps}}{\text{peak pps of statistic packets from all P4GWs}}$. As DPDK's packet processing capacity continues to advance (even enabling DPDK servers to act as P4 gateways [21]), the transmission subsystem's scalability bound is theoretically extremely high and does not constitute a system bottleneck.

B. Preparation and Transmission of Sketch

Existing sketch transmission and synchronization schemes suffer from two core limitations in 400 Gbps production environments. First, data-plane sketch acquisition schemes (e.g., SketchINT [11]) require additional hardware logic and cross-pipeline synchronization, which scales poorly with the number of buckets in high-throughput scenarios. Second, existing control-plane acquisition approaches lack robust mechanisms to resolve read-write conflicts and clock asynchrony between the control and data planes, leading to non-negligible statistical errors in line-rate processing.

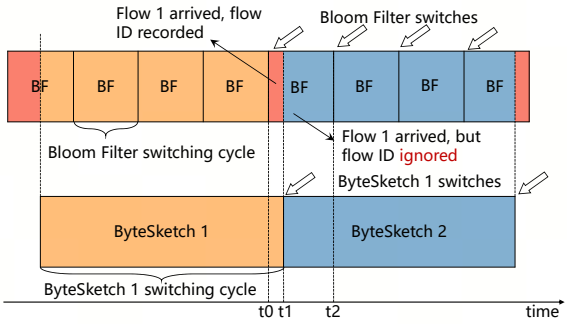


Fig. 3. Synchronization between BF and ByteSketch.

Sketch acquisition in control plane. To transmit the sketch[1] effectively, sketch acquisition can be performed either in the data plane or in the control plane. Solutions relying on data plane acquisition typically require additional hardware and data plane synchronization, scaling with the number of buckets. In contrast, for scalability in production environments, we perform sketch acquisition in the control plane instead.

Read-write separation. In a high-throughput production environment, accurate sketch reading is critical. Even small read errors can lead to significant statistical inaccuracies. To mitigate this, we designed two structures, *active* sketch and *idle* sketch, to enhance read accuracy. A single bit is used to mark whether a sketch is active or idle. The active sketch is used for statistics insertion, while the idle sketch is read and cleared. This separation of read and write operations ensures that the sketch reading process is consistent and free from the issues arising from simultaneous read and write actions.

Accurate matching between flow ID and ByteSketch. In high-throughput environments, accurately matching flow IDs with sketch indices is crucial for precise query results. Achieving this matching requires resolving two primary challenges: *clock synchronization* between the control plane (where the ByteSketch is read) and the data plane (where the flow ID is generated), and the *alignment of switching cycles* between different components.

Flow ID and ByteSketch ID synchronization. In production systems, using timestamps to map flow IDs to sketches introduces significant complexity. Specifically, reading from the sketch relies on the local control plane CPU to periodically trigger actions, so the clock used is based on the CPU's time. However, the flow ID is embedded in packets generated on the switching chip, where the clock is tied to the data plane's crystal oscillator. This inherent asynchrony between the CPU's clock and the data plane's clock increases the difficulty of aligning flow IDs with the sketch's statistical results. Additionally, the data plane clock rolls over every three days, further complicating synchronization efforts between the advancing Linux clock and the rolling data plane clock.

To address the issue of clock asynchrony, we propose a tag-based method where each flow ID is paired with a sketch ID. This sketch ID is included in the mirror packet that carries the flow ID. The sketch ID starts from 0 and increments each time the sketch undergoes an active-to-idle state transition. By

[1]We thereafter use sketch to refer to ByteSketch unless ambiguity arises.

including this sketch ID in the mirrored flow ID, we can ensure that the flow ID and sketch index are accurately synchronized, eliminating discrepancies due to clock mismatches.

Switching cycles synchronization. Another issue arises from the misalignment of switching cycles between the Bloom Filter (BF) and the sketch. For example, as shown in Fig. 3, consider the time period from `t0` to `t1`. During this time, `flow 1` triggers a BF miss, leading its ID to be sent to the KeyReceiver. However, in the subsequent time period, from `t1` to `t2`, since the BF has not yet switched, `flow 1` does not trigger a BF miss, and its ID is not sent to the KeyReceiver. Despite this, the sketch at time `t2` still records `flow 1`'s byte count, resulting in a mismatch between the flow ID and the data recorded by the sketch, causing errors in the statistics. This misalignment between the BF and sketch switching cycles leads to discrepancies in the final results.

To tackle the misalignment of switching cycles, we developed a strategy where the switching cycle of the BF is set to be an integer multiple of the sketch's switching cycle. Specifically, we trigger the sketch's switching on the data plane when the BF's switching reaches a cumulative number of times. Please refer to Algorithm 4 for more details.

Algorithm 4 Sketch Sync & Read-Write Separation

Input: Active sketch S_{act}, idle sketch S_{idle}, BF switching count cnt
Output: Synchronized sketch data for transmission

```
Increment cnt on each BF switching cycle
if cnt reaches integer multiple of sketch cycle then
    Swap S_act and S_idle, increment sketch ID sid
    Read and clear S_idle in control plane
    Transmit S_idle with sid to analysis subsystem
    Reset cnt to 0
end if
```

V. Analysis System

This section addresses **Sub-Problem 3** defined in Section II: skew-resilient real-time flow statistics reconstruction under the end-to-end latency constraint (4). We first analyze the limitations of state-of-the-art stream processing frameworks in sketch-based flow monitoring scenarios, then present our distributed storage design, memory-sensitive job assignment strategy, and TTL-based join optimization.

Existing programmable switch-accelerated stream processing systems, most notably Marple [22] and Sonata [23], have demonstrated the feasibility of fine-grained network traffic analysis via joint switch-stream processor design. However, these works are optimized for general key-value pair traffic queries, and face three core limitations when applied to our sketch-based flow monitoring scenario: (1) they do not account for the extreme memory asymmetry between large-scale flow IDs and small-size sketch data, leading to severe data skew under bursty flow arrival rates; (2) they lack targeted mechanisms to synchronize and match flow IDs with sketch statistics across asynchronous control/data plane outputs; (3) their generic join operators do not optimize for the short lifecycle of flow-sketch pairs, leading to excessive memory occupation and garbage collection overhead in long-term production operation. Our design addresses these limitations with targeted optimizations for the sketch-based flow monitoring pipeline.

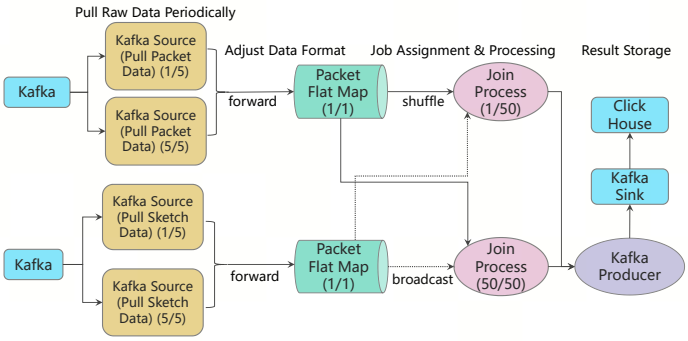


Fig. 4. Streaming analysis subsystem structure.

Distributed data storage. In a production environment, ensuring both the security and performance is essential. Hence, we have created two separate topics within the same Kafka cluster: one to store the flow IDs generated by FlowLog and the other to store the statistical Sketch results. We guarantee data safety by configuring replication, and optimize data read/write speed through partitioning. Additionally, we adopt different acknowledgment strategies for these two topics. Due to the large volume of flow IDs, which are replicated, we use asynchronous subscriptions for the key topic. In this setup, the producer sends data to the cluster and receives confirmation from a broker once the leader has successfully received the message. In contrast, since the sketch results are fewer but more critical, the producer waits for synchronization to complete across all followers before receiving broker confirmation, ensuring the highest level of reliability.

This differentiated storage strategy fully satisfies the reliability requirement of Sub-Problem 3, ensuring zero data loss for critical sketch statistics while maintaining high write throughput for bursty flow ID streams.

Streaming data analysis. Inspired by the performance of Marple [22] and Sonata [23] in stream processing, we adopted Flink as our high-speed analyzer. However, as noted above, their research mainly dealt with data structures consisting of keys and values, whereas our system involves querying and reconstructing data with flow IDs and sketches. The data source we work with for sketch reconstruction differs significantly from those used in Marple and Sonata, which necessitated specific modifications to our design of the stream analysis subsystem.

As shown in Fig. 4, the overall structure of the streaming analysis subsystem is as follows: 1) Pull raw data from Kafka topics. 2) Reformat the packet data using FlatMap. 3) Distribute the reformatted data based on the designed data flow process. 4) Analyze the data using the Join operator. 5) Store the results in a Kafka Sink topic for caching. 6) Offload the data to ClickHouse for subsequent analysis and display. Steps 1, 2, 5, and 6 are similar to those in existing research, but the focus of our work is on steps 3 and 4.

Memory-sensitive job assignment. In step 3, we introduced a new job assignment method based on data characteristics to mitigate data skew issues. Standard Flink Interval Join relies on symmetric hash shuffling for both input datasets,

which works well for balanced key-value streams. However, in our scenario, the sketch structure switches every second to reduce pressure on individual sketches, but the number of corresponding flow IDs can fluctuate significantly from second to second. For example, there could be 6 million flow IDs between seconds 0 and 5, and 12 million flow IDs between seconds 5 and 10. The standard Interval Join process assigns flow IDs from 0 to 5 seconds and from 5 to 10 seconds to different task groups, which often leads to severe data skew, excessive task backpressure, and violation of the end-to-end latency constraint (4).

To address this, we improved the job assignment during the standard Interval Join process, as shown in Fig. 5. When querying sketch results using flow IDs, there are many flow IDs and only a few megabytes of sketches. By leveraging the asymmetric memory distribution characteristic in the Interval Join, we broadcast the smaller, less resource-intensive sketches to all tasks and shuffle the larger, resource-intensive FlowLog keys to all tasks. With minimal data replication, we allow each Join operator to work efficiently in parallel, distributing the processing load evenly and handling the sudden increase in flow IDs effectively. We summarized the core workflow of the memory-sensitive job assignment strategy in Algorithm 5.

Algorithm 5 Memory-Sensitive Job Assignment

Input: Flow ID stream $\mathcal{F}_{stream}$, sketch stream $\mathcal{S}_{stream}$, task set $\mathcal{T}$
Output: Balanced join task assignment

1: **for** each time window T **do**
2: Fetch sketch batch S_T from $\mathcal{S}_{stream}$
3: Broadcast S_T to all tasks $t \in \mathcal{T}$
4: Fetch flow ID batch F_T from $\mathcal{F}_{stream}$
5: Shuffle F_T across tasks $t \in \mathcal{T}$ via uniform hash
6: Execute parallel join on each task $t \in \mathcal{T}$
7: **end for**

TTL-based cache for job join. In step 4, we implemented a TTL-based cache in the Join operator to manage flow IDs and sketches efficiently. Standard Flink join operators retain all input data until the end of the global window, leading to unbounded memory growth for continuous flow-sketch streams. TTL (Time-To-Live) is a mechanism used to specify the valid cache duration before it expires. Given that flow IDs and sketches are continuously generated in large quantities, we utilize a TTL expiration cache to conserve system resources. This prevents memory overuse due to delayed sketches or flow IDs. Due to space limitations, we have moved implementation details to Appendix **??**.

Besides, the processor's query algorithm employs thread-local memory pools to facilitate object reuse, reducing the need to generate new sketches and minimizing the frequency of garbage collection. The core execution flow of our flow-sketch join mechanism is summarized in Algorithm 6.

Algorithm 6 TTL-Based Flow-Sketch Join

Input: Flow ID f with sketch ID sid_f, sketch S with sketch ID sid_S, TTL duration T_{TTL}
Output: Estimated byte count $\hat{a}_f$ for flow f

1: Insert f into flow cache with expiration time $now + T_{TTL}$
2: Insert S into sketch cache with expiration time $now + T_{TTL}$
3: **if** $\exists S' \in$ sketch cache with $sid_{S'} = sid_f$ **then**
4: Query $\hat{a}_f$ from S' using f
5: Remove f from flow cache, output $\hat{a}_f$
6: **end if**
7: **if** $\exists f' \in$ flow cache with $sid_{f'} = sid_S$ **then**
8: Query $\hat{a}_{f'}$ from S using f'
9: Remove f' from flow cache, output $\hat{a}_{f'}$
10: **end if**
11: Evict all expired entries from flow/sketch cache

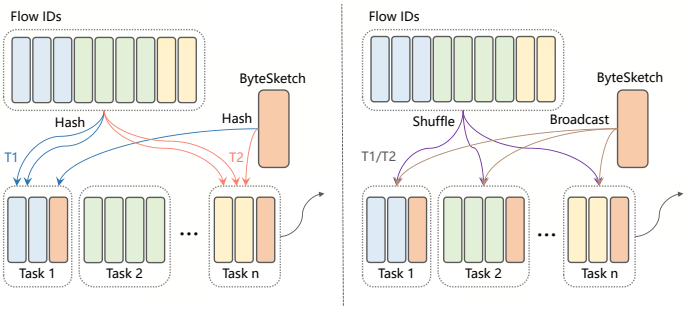


Fig. 5. Memory-sensitive job assignment policy (right).

VI. Evaluation and Analysis

In this section, we present the evaluation results for FlowLog, aiming to answer the following questions: 1) How does ByteSketch compare to existing algorithms? 2) How effectively does FlowLog detect flows under varying loads compared to existing solutions? 3) What is the overhead in both the data plane and the control plane?

Production environment setup. The testbed configuration is shown in Fig. 6. At the physical layer, we captured a 1:1 ratio of outbound traffic from the ByteDance data center using 8 splitters. This traffic was then fed into a H3C 64-port 100Gbps Ethernet switch, which forwarded the traffic through 8 of its 100Gbps ports to the deployed Bytehunter sFlow monitoring system. Bytehunter is a traffic monitoring system based on sFlow, widely used in ByteDance's data centers. Additionally, mirrored traffic was sent to our FlowLog system. Please refer to Appendix **??** for more implementation details.

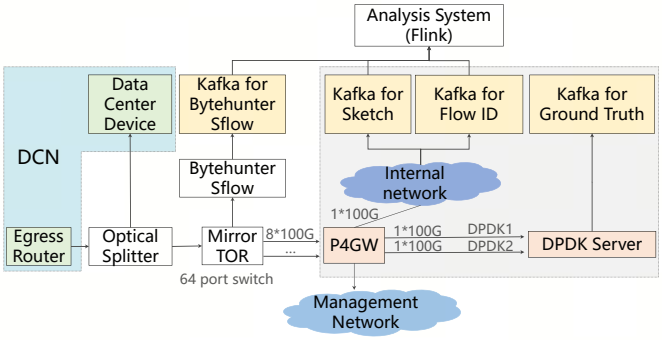


Fig. 6. Production environment setup.

Specifically, the mirrored 4*100Gbps traffic was directed into a Wedge100BF-65X switch equipped with the FlowLog system. This switch connects to both the management network and the internal network via a management port and its front panel ports, respectively. Using static routes, the ByteSketch measurement results sent to the Kafka Broker were directed to the interface connected to the internal network. The gateway

device used two 100Gbps ports to connect to a server with two DPDK-enabled cards, one for GroundTruth analysis and the other for FlowLog Keys reception. The results collected by KeyReceiver were then sent to the corresponding Kafka instance for storage through the server port connected to the internal network.

This testbed was deployed in a production environment for 6 months, with careful attention to ensuring that the configurations aligned with the operational demands of a real-world production network. Security checks, Kafka location backups, and production consumption permissions were configured to ensure seamless integration and reliability within the existing production infrastructure. However, these aspects are not the focus of this work and are not discussed in detail here. We also integrated FlowLog with existing visualization system, as shown in Appendix **??**.

We used the control plane of the P4GW to dynamically configure the data plane, controlling the randomness of the Ground Truth's hash function seeds and the table entries for the hash values selected as Ground Truth. Additionally, we varied the ACL rules for port admission to manage traffic inputs of different bandwidths, allowing us to evaluate performance under varying traffic conditions.

Parameter settings. The FlowLog system relies on three key parameters: the Bloom Filter update period, the ByteSketch switching period, and the scaling parameters for each layer of the ByteSketch. Given that stability is prioritized over performance in production networks, we accounted for the maximum input and added an appropriate margin to ensure system stability. Therefore, the ByteSketch scaling parameters were set to 13, 8, and 2, ensuring that the largest flows do not overflow the 32-bit buckets.

We set the Bloom Filter update period to 700 milliseconds, aligning with the current second-level granularity of our large-scale monitoring system. The ByteSketch switching period was configured to 4.9 seconds. While a faster rotation period would theoretically allow each Sketch to handle less traffic and improve performance, we took into account the need for adaptation to various gateways, each of which may have different data plane read/write requirements.

Compared solutions. We selected three representative measurement methods for experimental evaluation: 1) sFlow [6] is a simple, globally traffic estimation method based on uniform sampling. It works by using the switch mirroring function to send sampled packets to the switch CPU, which are then uploaded for remote analysis. Due to its straightforward and user-friendly approach, sFlow is widely deployed in the industry. 2) CMSketch [9] is a traffic estimation method based on probabilistic data structures. By using multiple hash functions for insertion and minimum value queries, it achieves high space efficiency and insertion rates. Its simple operations make it easily implementable in hardware and are considered one of the classic sketch-based methods. 3) TowerSketch [11] leverages flow distribution skewness by using different-sized counters in different arrays while maintaining the same bit-width. TowerSketch automatically assigns larger flows to larger counters and smaller flows to smaller counters, resulting in high accuracy in statistical estimation.

These methods emphasize different aspects: sFlow, widely adopted in industry, enables us to evaluate how our approach improves upon existing system-level solutions. CMSketch, a classic sketch-based method, and TowerSketch, a state-of-the-art, high-accuracy solution, serve as standard benchmarks, allowing us to compare our method against leading solutions.

A. Performance of ByteSketch

To accurately evaluate FlowLog in a production environment, we need true flow sizes, but per-packet measurement via a DPDK-enabled cluster is impractical. Instead, we apply hash-based sampling on each five-tuple (source IP, destination IP, source port, destination port, protocol number), extracting 1/128 of the tuples and sending them to a DPDK-enabled server for ground-truth statistics. A 32-bit random seed is inserted into the hash key each run to preserve traffic distribution, and all metrics are computed on this sampled subset.

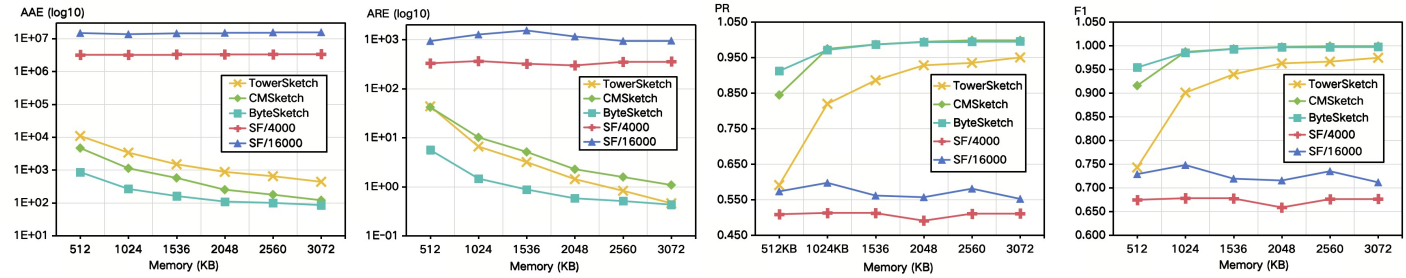
Fig. 7 demonstrates that our solution outperforms classic CMSketch, enhanced TowerSketch, and general sFlow-based deployment approaches in terms of Average Absolute Error (AAE) and Average Relative Error (ARE) across various available memory spaces. Additionally, it performs well in terms of identifying heavy hitters, with Precision Rate (PR) and F1 thresholds set at 10,000. Notably, our solution shows a significant advantage over other methods, especially in the case of small memory with unchanged input. In these scenarios, our system is approximately 10 times better in terms of ARE and several times better in terms of AAE. As memory becomes more abundant, TowerSketch gradually improves in performance; however, it remains challenging for P4GW chips to provide sufficient memory.

Fig. 8 illustrates that our solution exhibits minimal performance degradation across different inputs. It's important to note that the AAE and ARE curves are logarithmic, so while the degradation appears similar across the three sketches, our solution experiences significantly less degradation compared to the other two approaches. This highlights that our system demonstrates strong tolerance for large input sizes.

B. Performance of FlowLog

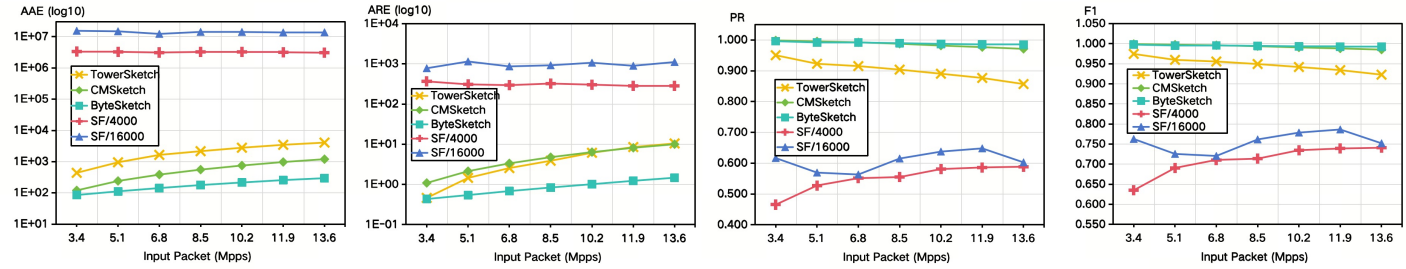
Flow detection performance. From Fig. 9(a), our system consistently maintains a stable detection rate of around 90%, which significantly outperforms sFlow in scenarios where flow ID is processed in the data plane. As shown in Fig. 9(b), sFlow exhibits a better detection rate for large flows compared to small ones. Our system excels in large flow detection via two separate Bloom Filters dedicated to small and large flows, respectively. However, since flow detection in our system relies on Bloom Filters, it performs better in scenarios with smaller input traffic compared to large traffic volumes. This difference highlights the trade-off between flow size and the effectiveness of flow detection in our system.

Flow size estimation performance. To ensure a fair comparison in the experiment, the measurement results presented here are calculated based on traffic sizes of 50KB or above, as sFlow estimates traffic by multiplying the packet length by the sampling rate. From Fig. 10(a) and (b), we observe



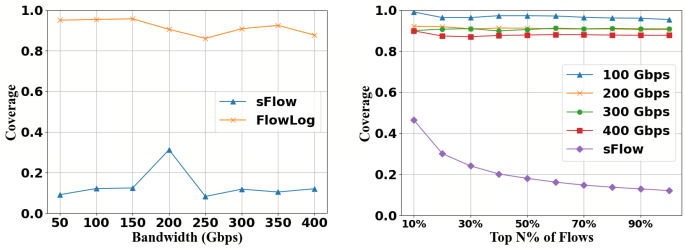
(a) Flow Size Estimaiton (AAE) in different Memory (b) Flow Size Estimaiton (ARE) in different Memory (c) Precsion Rate (PR) in different Memory (d) F1 in different Memory

Fig. 7. Basic performance of ByteSketch.



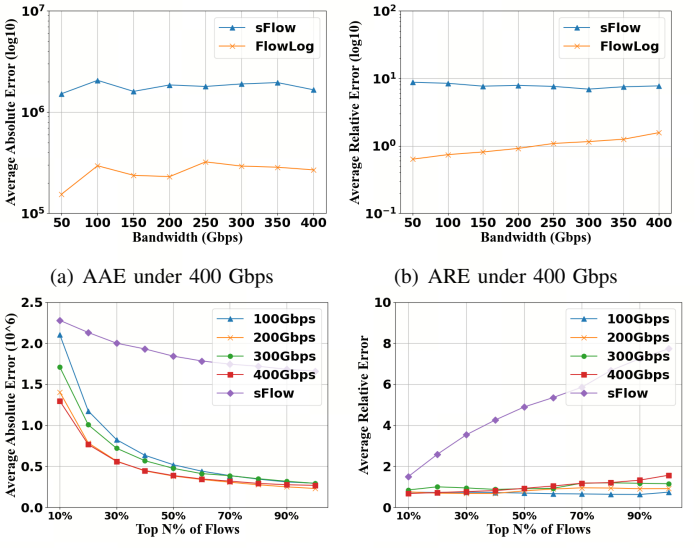
(a) Flow Size Estimaiton (AAE) in different Memory (b) Flow Size Estimaiton (ARE) in different Memory (c) Precsion Rate (PR) in different Memory (d) F1 in different Memory

Fig. 8. Basic performance of ByteSketch with different inputs.



(a) Coverage v.s. bandwidth (b) Coverage v.s. traffic distributions

Fig. 9. Flow detection performance.



(a) AAE under 400 Gbps (b) ARE under 400 Gbps

(c) AAE v.s. traffic distributions (d) ARE v.s. different traffic distributions

Fig. 10. Flow size estimation performance.

that both the AAE and ARE in our system are impacted by increasing traffic size, but the effect is minimal. As shown in Fig. 10(c), regardless of the solution or input traffic, the top 10% of AAE values remain significantly large. However, our system consistently outperforms sFlow in terms of accurately capturing small and medium-sized flows. Furthermore, as Fig. 10(d) illustrates, the top 10% of large flows have very low ARE across all solutions. This indicates that for large flows, both FlowLog and sFlow perform similarly well. However, for small and medium-sized flows, FlowLog consistently provides superior performance compared to sFlow.

C. Generalizability and Parameter Sensitivity

Beyond the comprehensive validation in our production data center environment, we further evaluate the cross-scenario generalizability of ByteSketch on three widely adopted public network traffic datasets, which cover three distinct real-world deployment contexts: enterprise security monitoring (CIC-IDS 2017), continental wide-area backbone network (CAIDA 2019), and international WAN link (MAWI). The experimental results consistently demonstrate that ByteSketch retains its significant performance advantage over state-of-the-art baseline schemes across all three testbeds: it achieves orders-of-magnitude lower average relative error for flow size estimation, and consistently outperforms competing solutions in heavy hitter detection precision and F1 score under identical memory constraints, regardless of variations in traffic distribution or network scale. Details of the performance metrics and results are provided in Appendix **??**.

To complement the detailed design description of ByteSketch's core parameters and transmission architecture, we

further conduct a systematic parameter sensitivity analysis to quantify the design trade-offs between ByteSketch's key configuration knobs and its measurement performance under heterogeneous flow size distributions. We focus on the scaling factors of the 8-bit and 16-bit counter arrays, the core tuning parameters of ByteSketch, where a scaling factor of 1 corresponds to a 1-bit arithmetic right shift applied to all counters in the target array (applicable to both 8-bit and 16-bit arrays).

Our analysis yields two clear, scenario-specific tuning insights: for traffic dominated by small flows, a higher 8-bit array scaling factor delivers notable performance gains by expanding the array's capacity to accommodate small flows and reducing cross-flow interference; for traffic with prominent heavy hitters, appropriately increasing the 16-bit array scaling factor effectively extends the dynamic range for large flow counting, delivering consistent improvements in heavy hitter estimation accuracy. The full parameter sweep results, in-depth trade-off discussion, and scenario-based tuning guidelines are elaborated in Appendix **??**.

D. System Overhead

Overhead in the Dataplane. As shown in Table I, our system does not rely on the most power-hungry or expensive TCAM resources. We allocate half of the available SRAM resources to each stage and make efficient use of Hash resources. Additionally, resources that occupy the Packet Header Vector, such as IPv6 addresses, can be shared when embedding other business logic. Our system introduces an additional 16 bytes of bridge metadata for transmission, which is acceptable. We utilize all 12 stages, but the monitoring and business pipelines run in parallel, so no additional serial logic is introduced.

TABLE I
DATA PLANE RESOURCE UTILIZATION

TCAM%	SRAM%	Hash%	HashDist Unit%	Bridged Md	Serial Stage
3.10	44.30%	12.10	44.40	16 Bytes	12

For CPU and memory usage, these are primarily dependent on the configuration. With our current setup, under a 400 Gbps input, the system consumes approximately 4.1 CPU cores and 1.184GB of memory. On a 4-core, 8-thread Intel Pentium D1517 @ 1.60GHz CPU, this performance is acceptable, and P4GW deployment does not impact normal forwarding. This demonstrates good compatibility, allowing the system to adapt to older P4 switches by reducing detection frequency. When deployed with more powerful CPUs, a faster Sketch update frequency can be configured for improved accuracy.

Overhead of the Analysis Subsystem. As shown in Table II, under a 400 Gbps input, our FlowLog system exerts the highest pressure on the analysis system, utilizing 154 vCPUs and outputting 1.3Gbps of flow IDs. As the input traffic decreases, both the number of CPUs used and the flow IDs output also decrease proportionally.

Given that handling high-throughput traffic is our primary consideration, the system's memory usage remains relatively stable at around 390GB, regardless of input size. After allocating a significant amount of memory, the largest garbage collection during the six-month deployment was 76.4ms, with the largest observed lag being 55ms, demonstrating FlowLog's consistent performance under varying conditions.

VII. RELATED WORK

In this section, we provide an overview of sampling-based and sketching solutions for measuring network traffic. As indicated in Table III, our system stands out by offering both byte and packet counting capabilities, compatibility with current network traffic analysis applications, and the distinction of being the first system capable of adapting to bandwidths as high as 400 Gbps.

Sampling-based solutions. Many measurement systems [24], [25], [26], [27], [28], [6], [7], [20], [29] have been developed through packet sampling techniques. These systems estimate the behavior of the entire network by sampling a portion of the data. Typical sampling-based systems in this context include NetFlow [7], sFlow [6], Csamp [30], OpenSample [24], and Everflow [20]. Sampling techniques reduce bandwidth usage and lower the demands on storage. However, this efficiency gain comes with a cost: decreased accuracy and the potential to miss critical events, especially when packet loss occurs and important data isn't sampled.

Sketch-based solutions. The academic community has conducted extensive research on sketching solutions, with typical examples including Count–min (CM) [9], Conservative-Update (CU) [10], Pyramid [31], Tower [11], Elastic [8], and more [32], [33], [34]. Initial methods such as CM and CU, however, falter in distinguishing between large and small flows, resulting in diminished accuracy due to hash collisions. In response to the challenges posed by skewed traffic distributions, more recent sketches have integrated specialized mechanisms to address these issues. For instance, PyramidSketch [31] and TowerSketch [11] use large counters to more accurately account for large flows. Specifically, PyramidSketch arranges counters within a hierarchical structure with flag bits. TowerSketch comprises multiple arrays of counters with varying sizes, recording small flow information in smaller counters. SketchLearn [35] and DeltaINT [36] further optimize sketch designs from the perspectives of automated statistical inference for measurement configuration and low-overhead in-band network telemetry, respectively.

The two most closely related works, FCM-Sketch [18] and Count-Less [19], propose hierarchical heterogeneous counter architectures for PISA programmable switches, tailored to real network traffic's skewed Zipf distribution. FCM-Sketch uses a tree-based feed-forward design with overflow-triggered cascaded updates, while Count-Less employs a cross-layer minimum update strategy for enhanced traffic distribution robustness. Both are optimized for packet-level network measurement, yet lack targeted optimizations for byte-level monitoring in 400+ Gbps high-throughput environments, where small counters suffer severe overflow and estimation bias. In contrast, our ByteSketch is purpose-built for byte-level flow measurement with a tightly coupled shift-compression and probabilistic compensation framework. This design fundamentally resolves small counter overflow in byte counting, ensures unbiased flow size estimation with rigorous theoretical error

TABLE II
OVERHEAD OF ANALYSIS SUBSYSTEM

Input		Key		Analysis resource		Local resource	
PPS (Mpps)	BPS (Mbps)	PPS (Mpps)	BPS (Mbps)	vCPU	Memory (GB)	CPU	Memory (GB)
60.17	404484	3.96	1360.38	154	388	4.18	1.184
52.68	354482	3.38	1122.18	122	385	4.12	1.056
45.18	304127	2.89	1046.74	105	386	4.14	1.073
37.65	253389	2.36	811.38	85.4	386	4.16	1.035
30.25	204166	1.84	661.38	66.6	384	4.1	1.022
22.69	153076	1.37	494.57	48.2	386	4.09	1.002
14.96	103350	0.85	303.96	33.0	387	4.12	1.056
7.45	53108	0.44	154.78	16.8	388	4.08	0.992

TABLE III
COMPARISON OF DIFFERENT SOLUTIONS. (RMT REPRESENTS THE RECONFIGURABLE MATCH TABLES)

Systems	Obtain flow IDs	Bytes counting	Per packet counting	RMT	End-to-end system implementation	Experiment environment
CM [9]	×	×	✓	✓	×	Theoretical Analysis
SKetchINT [11]	×	×	✓	✓	×	Simulation
ElasticSketch [8]	✓	×	✓	✓	×	Simulation
sFlow [6]	✓	✓	×	✓	✓	Industry
OmniMon [12]	✓	✓	✓	✓	✓	Simulation
everflow [20]	✓	✓	×	×	✓	Industry
flowradar [13]	✓	×	✓	×	✓	Simulation
CU [10]	×	×	✓	×	×	Simulation
LruMon [14]	✓	✓	✓	✓	×	Simulation
FlowLog	✓	✓	✓	✓	✓	Industry

bounds, and better conforms to P4 switch pipelines' single-step atomic update constraints.

Recent works adopt quantization and vectorization for finer-grained per-flow measurement. HistSketch [16] uses a hot-cold separated architecture with histogram shedding and equation-based decoding to mitigate counter amplification in per-key distribution monitoring. SketchFeature [17] leverages sketch virtualization and Bloom filter membership testing to achieve high-resolution full-range 3rd-order per-flow feature extraction for in-network security detection. While both support aggregated flow byte volume measurement via bin-level counting, they fail to meet core demands of production-grade byte-level monitoring: HistSketch's offline decoding cannot support real-time in-data-plane queries, and SketchFeature's multi-bin virtual sketch incurs excessive memory and pipeline overhead for pure byte count statistics. By contrast, our ByteSketch addresses CM's inherent overflow limitation in byte counting via lightweight shift-compression and probabilistic compensation, achieving line-rate byte-level measurement with minimal hardware overhead in 400+ Gbps production environments.

Additionally, certain measurement systems deploy sketches tailored to their specific needs, including FlowRadar [13], OpenSketch [37], BeauCoup [38], OmniMon [12], Marple [22], Sonata [23], and more [39], [40], [11]. Specifically, OmniMon uniquely manages flow information by storing only the flow keys and associated per-flow metrics at the end-hosts, while aggregating flow metrics in shared memory slots within the switches. This approach optimizes memory utilization while ensuring precise tracking for each flow. The most closely related work to our system is P4LRU [14], which offers both flow ID popping and byte & packet counting capabilities. P4LRU is defined as an approach to implement the classical Least Recently Used (LRU) cache on the data plane of programmable switches. It introduces a pipelined variant of the LRU implementation, which enhances packet processing efficiency. LruMon [14], a network measurement system, works in conjunction with TowerSketch and the P4LRU mechanism. It excels at filtering out small flows and accurately aggregating and measuring large flows. However, when compared to our system, LruMon exhibits several limitations: 1) has not been evaluated in high-bandwidth environments. 2) lacks a comprehensive analysis system. 3) never deployed in real-world production environments.

VIII. LESSONS LEARNED AND LIMITATIONS

This section distills generalizable, field-validated design principles for programmable data plane network measurement systems, drawn from our end-to-end production deployment experience with FlowLog.

Maintaining isolated metadata structures is critical for non-intrusive integration into existing programmable data planes. Reusing shared packet header vector (PHV) or global metadata for new logic introduces high risks of compile conflicts, resource allocation failures and runtime interference. Isolating purpose-built metadata for new modules eliminates cross-module contention, minimizes modifications to the original pipeline, and guarantees forwarding stability.

During FlowLog's deployment, we encountered compile issues when integrating new logic into our existing P4 gateway. To implement this principle, we introduced separate

local metadata as local variables for FlowLog exclusively, using `HashAlgorithm_t.IDENTITY` for assignment. This approach avoids reusing PHV for multiple packet lengths, ensuring FlowLog's operations do not interfere with gateway's existing logic, and minimizes changes to the original PHV allocation for smoother integration.

Co-locating intermediate state production and consumption in the same pipeline stage maximizes bandwidth efficiency for stateful in-network processing. Transmitting large intermediate variables across pipeline stages occupies precious PHV bandwidth, increases latency, and limits system throughput, a universal constraint for commercial programmable switches. Co-locating the generation and consumption of intermediate state within the same stage eliminates unnecessary transmission overhead and improves overall bandwidth efficiency.

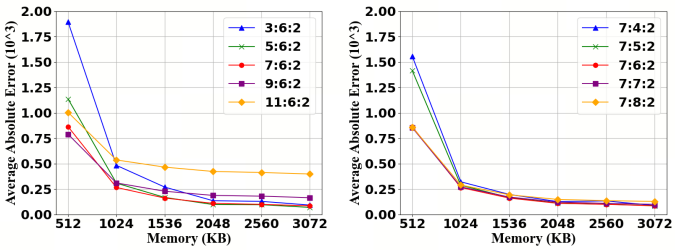
To optimize system bandwidth, we repositioned registers across different layers of the sketch structure so that the production and consumption of large intermediate variables, such as hash indices, occur within the same pipeline stage. This reduces transmission overhead and increases supportable bandwidth. We embedded necessary sketches directly into the incoming pipeline rather than transmitting additional fields, and the negligible extra overhead from this optimization does not impact most switches, especially those where input packets are generally longer than output packets.

Modular hierarchical encapsulation enables flexible deployment across heterogeneous pipeline resource constraints. Production-grade programmable gateways have highly variable available pipeline resources and memory across deployment scenarios. A monolithic design either fails to integrate into resource-constrained pipelines or wastes available resources in high-end devices. Modular encapsulation allows users to selectively enable functions and scale parameters according to available resources, balancing functionality, resource usage and integration complexity.

FlowLog supports both basic and advanced deployment modes. In basic mode, all functional modules are centralized in a single pipeline for quick integration into business gateways. In advanced mode, we encapsulated FlowLog sub-functions (preprocessor, mirror, sketch, etc.) into independent control modules, improving the clarity of P4 source code management. This allows users to integrate required functionalities by adjusting sketch sizes according to available pipeline resources.

Tiered state refresh and adaptive classification mitigate BF false positives under dynamic traffic. Static filter parameters and fixed classification thresholds for sketch-based flow detection inevitably lead to severe false positives under highly dynamic, non-stationary production traffic, which often exhibits bursty behavior, rapid flow churn and fluctuating flow size distributions that cannot be captured by static configurations. A general applicable two-tier optimization strategy is required: tiered state refresh cycles for complementary sketch structures, and adaptive traffic classification tailored to the stability of the target application scenario.

During deployment, we encountered false positive challenges with the BF in flow detection. To mitigate this, we first implemented a faster refresh cycle for the BF compared to the ByteSketch. We also incorporated a lightweight online



(a) 8-bit parameter (b) 16-bit parameter

Fig. 11. Effects of parameters on performance.

traffic query module using sketch-derived statistics to prioritize large flows, but found fixed thresholds failed to fully capture the dynamic nature of traffic. For stable scenarios with minimal new business integration, training a transformer model on historical data allowed us to dynamically adjust classifications based on learned patterns. For scenarios with a rapid influx of new business traffic, we adopted online learning algorithms (reinforcement learning, anomaly detection, etc.) to quickly identify traffic pattern changes and enable real-time adjustments to thresholds and flow classifications.

Differentiated scaling factor tuning balances overflow risk and accuracy for multi-bit-width sketches. For multi-bit-width probabilistic sketch structures with hierarchical bucket designs, the sensitivity of measurement accuracy to scaling factor adjustments follows two general rules: 1) scaling factor changes have a greater impact on accuracy in memory-constrained scenarios; 2) under the same memory budget, shorter bit-width buckets have higher sensitivity to scaling factor adjustments than longer bit-width buckets. The core tuning principle is: prioritize overflow prevention for the longest bit-width buckets to set the upper bound of the measurement range, and optimize empirical accuracy for shorter bit-width buckets.

Let X, Y, and Z represent the number of bits to right shift when updating values in 8-bit, 16-bit, and 32-bit buckets, respectively, given by the format $X : Y : Z$. As X, Y, and Z increase, the scaling factor of the records in the buckets becomes larger. As shown in Fig. 11, we validate the above rules with the following observations:

Firstly, scaling factor adjustments have a greater effect on accuracy in small memory scenarios. In small memory scenarios where multiple flows share a bucket, smaller scaling factors increase the likelihood of bucket overflow and reduce accuracy. In large memory scenarios with more buckets and fewer flows per bucket, smaller scaling factors introduce less estimation error from probabilistic compensation, resulting in higher accuracy. The trends for 32-bit parameters are consistent with those for 8-bit and 16-bit parameters.

Secondly, under the same memory conditions, scaling factor adjustments are more sensitive for shorter bit-width buckets. For 8-bit buckets, adjustments have a greater effect on accuracy due to fewer flows recorded per bucket, which amplifies errors from probabilistic compensation. In contrast, long bit-width buckets accumulate more packets from more flows, reducing the side-effect of probabilistic errors.

Thirdly, the measurement accuracy demonstrates high robustness to slight deviations in the scaling factors. In both

small and large memory scenarios, when any scaling parameter deviates from its optimal configuration by a shift of 1, the resulting increase in Average Absolute Error (AAE) remains below 5%. Therefore, in production environments, achieving the exact optimal configuration is not strictly required. The system allows for slight parameter fluctuations, making the tuning process relatively straightforward.

Based on these observations, we establish a generalizable tuning workflow: For buckets with the longest bit width, the scaling factor should be set to ensure the maximum statistical peak can accommodate the largest input, preventing overflow across all buckets. For shorter bit-width buckets, traffic replay tools can be used to test different scaling factor combinations for optimal tuning results. Alternatively, an empirical approach for rapid deployment is to control the overflow rate of byte statistics buckets between 1% and 5%, which can still provide acceptable accuracy.

Limitations. First, due to the need to reserve optical ports for emergency debugging, we couldn't fully utilize all 16 fiber connections. As a result, the traffic rate was limited to 400 Gbps on the existing 8 fiber connections, considering the stability of the production network. Second, our system's performance consumes local CPU resources. With older devices equipped with 4 cores and 8 threads, we couldn't increase table read speeds further. If more CPU resources become available, we can accelerate register read rates. Third, the KeyReceiver, implemented on a DPDK-enabled server, efficiently collects keys from a large number of gateways. However, it may not be ideal for small-scale scenarios, where a specialized software parsing solution may be required.

IX. Conclusion

We present FlowLog, a traffic monitoring system that achieves high accuracy, speed, and adaptability. FlowLog introduces ByteSketch, a novel algorithm enabling precise per-flow byte counting in high-throughput environments. Throughout its design and implementation, we have addressed the real-world challenges encountered in deployment and developed effective solutions to minimize overhead while ensuring accuracy. After running continuously for over six months in ByteDance's data center with 400 Gbps peak bandwidth, FlowLog demonstrated superior performance over existing solutions and state-of-the-art algorithms. The deployment provided valuable lessons learned, which guided improvements in system compatibility, integration, and traffic detection, further enhancing FlowLog's adaptability and efficiency in complex production environments.

References

[1] H. Wang, A. Abhashkumar, C. Lin, T. Zhang *et al.*, "{NetAssistant}: Dialogue based network diagnosis in data center networks," in *Proc. USENIX NSDI*, 2024, pp. 2011–2024.

[2] S. McClure, Z. Medley, D. Bansal, K. Jayaraman, A. Narayanan *et al.*, "Invisinets: Removing networking from cloud networks," in *Proc. USENIX NSDI*, 2023, pp. 479–496.

[3] T. Pan, N. Yu, C. Jia, J. Pi, L. Xu, Y. Qiao, Z. Li, K. Liu, J. Lu, J. Lu *et al.*, "Sailfish: Accelerating cloud-scale multi-tenant multi-service gateways with programmable switches," in *Proc. ACM SIGCOMM*, 2021, pp. 194–206.

[4] T. Pan, K. Liu, X. Wei, Y. Qiao, J. Hu, Z. Li, J. Liang *et al.*, "{LuoShen}: A {Hyper-Converged} programmable gateway for {Multi-Tenant}{Multi-Service} edge clouds," in *Proc. USENIX NSDI*, 2024, pp. 877–892.

[5] J. Gao, J. Cao, Y. Li, M. Liu, M. Tang, D. Cai, and E. Zhai, "Sirius: Composing network function chains into {P4-Capable} edge gateways," in *Proc. USENIX NSDI*, 2024, pp. 477–490.

[6] P. Phaal, S. Panchen, and N. McKee, "Rfc3176: Inmon corporation's sflow: A method for monitoring traffic in switched and routed networks," USA, 2001.

[7] B. Claise, "RFC 3954: Cisco Systems NetFlow Services Export Version 9," USA, 2004.

[8] T. Yang, J. Jiang, P. Liu, Q. Huang, J. Gong, Y. Zhou, R. Miao, X. Li, and S. Uhlig, "Elastic sketch: Adaptive and fast network-wide measurements," in *Proc. ACM SIGCOMM*, 2018, pp. 561–575.

[9] G. Cormode and S. Muthukrishnan, "An improved data stream summary: the count-min sketch and its applications," *Journal of Algorithms*, vol. 55, no. 1, pp. 58–75, 2005.

[10] C. Estan and G. Varghese, "New directions in traffic measurement and accounting: Focusing on the elephants, ignoring the mice," *ACM Transactions on Computer Systems*, vol. 21, no. 3, pp. 270–313, 2003.

[11] K. Yang, S. Long, Q. Shi, Y. Li, Z. Liu, Y. Wu, T. Yang, and Z. Jia, "Sketchint: Empowering int with towersketch for per-flow per-switch measurement," *IEEE Transactions on Parallel and Distributed Systems*, 2023.

[12] Q. Huang, H. Sun, P. P. Lee, W. Bai, F. Zhu, and Y. Bao, "Omnimon: Re-architecting network telemetry with resource efficiency and full accuracy," in *Proc. ACM SIGCOMM*, 2020, pp. 404–421.

[13] Y. Li, R. Miao, C. Kim, and M. Yu, "{FlowRadar}: A better {NetFlow} for data centers," in *Proc. USENIX NSDI*, 2016, pp. 311–324.

[14] Y. Zhao, W. Liu, F. Dong, T. Yang, Y. Li, K. Yang, Z. Liu, Z. Jia, and Y. Yang, "P4lru: Towards an lru cache entirely in programmable data plane," in *Proc. ACM SIGCOMM*, 2023, pp. 967–980.

[15] J. Han, K. Xue, W. Wang *et al.*, "Ratemp: Optimizing bandwidth utilization with high burst tolerance in data center networks," in *Proc. IEEE INFOCOM*, 2024, pp. 1361–1370.

[16] J. He, J. Zhu, and Q. Huang, "Histsketch: A compact data structure for accurate per-key distribution monitoring," in *Proc IEEE ICDE*, 2023, pp. 2008–2020.

[17] S. Kim, S. M. M. Mirnajafizadeh, B. Kim, R. Jang, and D. Nyang, "Sketchfeature: High-quality per-flow feature extractor towards security-aware data plane," in *Proc. ISOC Network and Distributed System Security Symposium (NDSS)*, 2025, pp. 1–17.

[18] C. H. Song, P. G. Kannan, B. K. H. Low, and M. C. Chan, "Fcm-sketch: generic network measurements with data plane support," in *Proc. ACM CoNEXT*, 2020, pp. 78–92.

[19] S. Kim, C. Jung, R. Jang, D. Mohaisen, and D. Nyang, "A robust counting sketch for data plane intrusion detection," in *Network and Distributed System Security Symposium (NDSS)*, 2023, pp. 1–17.

[20] Y. Zhu, N. Kang, J. Cao, A. Greenberg, G. Lu, R. Mahajan, D. Maltz, L. Yuan, M. Zhang, B. Y. Zhao *et al.*, "Packet-level telemetry in large datacenter networks," in *Proc. ACM SIGCOMM*, 2015, pp. 479–491.

[21] J. Lu, S. Zhu, J. Liang, Y. Lin *et al.*, "Albatross: A containerized cloud gateway platform with fpga-accelerated packet-level load balancing," in *Proc. ACM SIGCOMM*, 2025, pp. 71–84.

[22] S. Narayana, A. Sivaraman, V. Nathan, P. Goyal *et al.*, "Language-directed hardware design for network performance monitoring," in *Proc. ACM SIGCOMM*, 2017, pp. 85–98.

[23] A. Gupta, R. Harrison, M. Canini, N. Feamster, J. Rexford, and W. Willinger, "Sonata: Query-driven streaming network telemetry," in *Proc. ACM SIGCOMM*, 2018, pp. 357–371.

[24] J. Suh, T. T. Kwon, C. Dixon, W. Felter, and J. Carter, "Opensample: A low-latency, sampling-based measurement platform for commodity sdn," in *Proc. IEEE INFOCOM*, 2014, pp. 228–237.

[25] M. Yu, L. Jose, and R. Miao, "Software {Defined}{Traffic} measurement with {OpenSketch}," in *Proc. USENIX NSDI*, 2013, pp. 29–42.

[26] F. Li, A. A. Niaki, D. Choffnes, P. Gill, and A. Mislove, "A large-scale analysis of deployed traffic differentiation practices," in *Proc. ACM SIGCOMM*, 2019, pp. 130–144.

[27] P. Nikolopoulos, C. Pappas, K. Argyraki, and A. Perrig, "Retroactive packet sampling for traffic receipts," *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, vol. 3, no. 1, pp. 1–39, 2019.

[28] D. Yu, Y. Zhu, B. Arzani, R. Fonseca *et al.*, "{dShark}: A general, easy to program and scalable framework for analyzing in-network packet traces," in *Proc. USENIX NSDI*, 2019, pp. 207–220.

[29] A. Roy, H. Zeng, J. Bagga, G. Porter, and A. C. Snoeren, "Inside the social network's (datacenter) network," in *Proc. ACM SIGCOMM*, 2015, pp. 123–137.

[30] V. Sekar, M. K. Reiter, W. Willinger, H. Zhang, R. R. Kompella, and D. G. Andersen, "Csamp: a system for network-wide flow monitoring," in *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation*, ser. NSDI'08. USA: USENIX Association, 2008, p. 233–246.

[31] T. Yang, Y. Zhou, H. Jin, S. Chen, and X. Li, "Pyramid sketch: A sketch framework for frequency estimation of data streams," *Proceedings of the VLDB Endowment*, vol. 10, no. 11, pp. 1442–1453, 2017.

[32] J. Li, Z. Li, Y. Xu, S. Jiang, T. Yang, B. Cui, Y. Dai, and G. Zhang, "Wavingsketch: An unbiased and generic sketch for finding top-k items in data streams," in *Proc. ACM SIGKDD*, 2020, pp. 1574–1584.

[33] Q. Huang, S. Sheng, X. Chen, Y. Bao, R. Zhang, Y. Xu, and G. Zhang, "Toward {Nearly-Zero-Error} sketching via compressive sensing," in *Proc. USENIX NSDI*, 2021, pp. 1027–1044.

[34] H. Li, Q. Chen, Y. Zhang, T. Yang, and B. Cui, "Stingy sketch: a sketch framework for accurate and fast frequency estimation," *Proceedings of the VLDB Endowment*, vol. 15, no. 7, pp. 1426–1438, 2022.

[35] Q. Huang, P. P. Lee, and Y. Bao, "Sketchlearn: Relieving user burdens in approximate measurement with automated statistical inference," in *Proceedings of SIGCOMM*, 2018, pp. 576–590.

[36] S. Sheng, Q. Huang, and P. P. Lee, "Deltaint: Toward general in-band network telemetry with extremely low bandwidth overhead," in *Proc. IEEE ICNP*, 2021, pp. 1–11.

[37] Y. Gryaditskaya, M. Sypesteyn, J. W. Hoftijzer, S. C. Pont, F. Durand, and A. Bousseau, "Opensketch: a richly-annotated dataset of product design sketches." *ACM Trans. Graph.*, vol. 38, no. 6, pp. 232–1, 2019.

[38] X. Chen, S. Landau-Feibish, M. Braverman, and J. Rexford, "Beaucoup: Answering many network traffic queries, one memory update at a time," in *Proc. ACM SIGCOMM*, 2020, pp. 226–239.

[39] Y. Du, H. Huang, Y.-E. Sun, S. Chen, and G. Gao, "Self-adaptive sampling for network traffic measurement," in *Proc. IEEE INFOCOM*, May 2021.

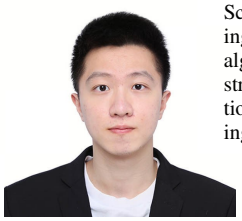
[40] Y. Shi and M. Wen, "srouting: Towards a better flow size estimation performance through routing and sketch configuration," in *Proc. ACM ICPP*, 2021, pp. 1–11.

Long Chen (Member, IEEE) received the B.E. and Ph.D degrees in Software Engineering from Shanghai Jiao Tong University, China in 2016 and 2022. Currently, he is a postdoctoral researcher in the School of Computing Science at Simon Fraser University, Canada. His research interests include software defined networking, space-terrestrial integrated networks, and machine learning.

Mingwei Cui received the B.S. degree of Information and Communication Engineering from Beijing University of Posts and Telecommunications, China in 2019. He is currently pursuing a Doctorate degree at Beijing University of Posts and Telecommunications. His research interests include future network architecture, In-band Network telemetry, Network measurement, and P4.

Qiuheng Yin is a graduate student at Peking University specializing in computer networks. His research covers network measurement, network security, and in-network computing.

Hanglong Lyu is currently a Master's student at the School of Computer Science, Peking University. His research focuses on large language models (LLMs), natural language processing (NLP), and the optimization and evaluation of intelligent systems. His specific research interests include LLM pretraining and enhancing coding capabilities through post-training techniques. He attaches great importance to academic writing and paper publication, with the goal of applying research outcomes to real-world applications. Looking ahead, he aims to continue exploring the intersection of artificial intelligence and data science, pursuing more efficient and interpretable intelligent systems.



Yisen Hong is currently pursuing a PhD at the School of Computer Science and Technology, Tsinghua University. His research focuses on sketch algorithms and network measurement. He places strong emphasis on academic writing and publication, and is committed to translating research findings into practical, real-world applications.

Tong Yang (Member, IEEE) received the Ph.D. degree in computer science from Tsinghua University in 2013. He visited the Institute of Computing Technology, Chinese Academy of Sciences. Now he is an associate professor with School of Computer Science, Peking University. His research interests include LLM, network measurements, sketches, IP lookups, Bloom filters, and KV stores. He has served as a TPC Member for several premier conferences such as ICDE, INFOCOM, IMC, ICNP, etc.. He is currently an Associate Editor for Knowledge and Information Systems. He published dozens of papers in IEEE/ACM ToN, IEEE JSAC, IEEE TPDS, IEEEToC, IEEE TKDE, SIGCOMM, SIGKDD, SIGMOD, NSDI, USENIX ATC, ICDE, VLDB, INFOCOM, etc.

Yangyang Bai received the Master's degree in Computer Science and Cloud Computing from Hangzhou Dianzi University, China, in 2021. He is currently an engineer in the Network Team at ByteDance, specializing in the development and optimization of network monitoring and alerting systems. His research interests include network monitoring and alerting, as well as high-performance packet capture and related fields.

Ziwei Zhao received the Bachelor's degree in Cyberspace Security from XI'AN University of Posts and Telecommunications in 2024. He is currently an engineer in the Network Team at ByteDance, specializing in the development of network alerting and automation processes. His research interests include the automation of network alerting and high-performance network detection.