NetScope: Fault Localization in Programmable Networking Systems With Low-Cost In-Band Network Telemetry and In-Network Detection

Hongyang Chen, Benran Wang, Guangba Yu, Zilong He, Pengfei Chen, Chen Sun, and Zibin Zheng, *Fellow, IEEE*

***Abstract*—Recently, Software Defined Networking (SDN) has gained widespread adoption as a network infrastructure. Although the openness and programmability of SDN facilitate large complex network construction, diagnosing faults in datacenter-scale network remains challenging. Previous network diagnosis tools pose significant overhead in fine-grained telemetry and typically lack automated fine-grained fault diagnosis capabilities. Although on-demand monitoring methods have been proposed to reduce telemetry overhead, they struggle with effectively setting fixed thresholds, which requires expert experience. This paper presents NetScope, a lightweight system for real-time anomaly detection with self-adaptive thresholds and automatic root cause localization in programmable networking systems. NetScope estimates latency medians for each Flow (i.e., a pair of source and sink switches) within the switch using the proposed per-Flow quantile sketch and calculates the threshold accordingly for anomaly detection. Upon detecting anomalies, NetScope collects aggregated packet-level telemetry on demand and generates a ranked list of fine-grained fault culprits at multiple levels, including port-level, Flow-level, and switch-level. Extensive experiments demonstrate the effectiveness and efficiency of NetScope in anomaly detection and fault localization. Specifically, NetScope achieves a 32%∼116% relative improvement in anomaly detection and 6%∼197% improvement in root cause analysis compared with other baselines without causing any network bandwidth in anomaly detection while consuming 64.2% less telemetry bandwidth for localization.**

***Index Terms*—P4, in-band network telemetry, software defined networking, fault localization.**

I. Introduction

DUE to the demand for low latency and high throughput applications, a growing number of IT enterprises are deploying their applications in datacenters. As applications dynamically scale to deliver superior quality of experience (QoE), efficient data center network management becomes crucial. This necessitates networks that can scale to meet evolving application demands while maintaining reliable network management.

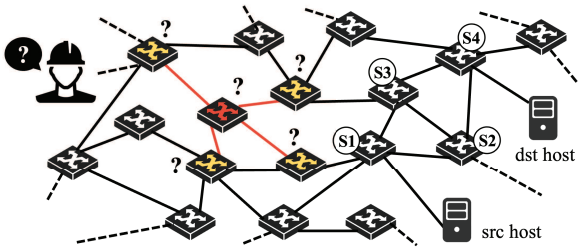


Fig. 1. A fault in a large-scale network: a switch failure (red) causes neighbor switches (orange) to behave abnormally.

Recently, the adoption of SDN and programmable networking has enhanced datacenter network infrastructure management while significantly increasing network scale and complexity. With rapidly expanding networks, failures have become commonplace rather than exceptions. Timely diagnosis of these failures is critical, as even a minor network performance degradation can substantially impact upper-layer application quality [1]. Fig.1 illustrates a complex datacenter network where a single switch failure induces cascading failures across multiple neighboring switches. For instance, prior studies [2], [3], [4] have shown that failures of top-of-rack (ToR) switches often impact all devices within the same rack. This cascading effect highlights the limitations of manual diagnosis and underscores the necessity for automated fault localization.

Network failure diagnosis typically includes monitoring, anomaly detection, and fault localization. The monitoring phase collects sufficient valuable information, the anomaly detection phase identifies abnormal network traffic, and the fault localization phase identifies the root cause. Traditional diagnosis methods monitor networks either through packet mirroring at switches [5], [6], [7], [8] or by actively injecting probe packets [9], [10], [11]. However, this additional network traffic negatively impacts performance. To reduce monitoring overhead, programmable switches have inspired extensive research [12], [13], [14], [15], [16], [17] on diagnosing failures in complex networks using In-band Network Telemetry (INT), introduced in §II, which embeds telemetry headers in packets to collect packet-level monitoring data. Nevertheless, these approaches remain inadequate due to three key limitations.

1) **Existing monitoring methods based on INT cause a high packet transmission overhead.** These methods require each switch to inject telemetry headers into packets, increasing their size linearly with the path length. Additionally, some enhanced methods [18]

Received 12 July 2024; revised 15 May 2025; accepted 8 July 2025; approved by IEEE TRANSACTIONS ON NETWORKING Editor C. Peng. This work was supported in part by the National Key Research and Development Program of China under Grant 2019YFB1804002, in part by the National Natural Science Foundation of China under Grant 62272495, and in part by the Guangdong Basic and Applied Basic Research Foundation under Grant 2023B1515020054. *(Corresponding author: Pengfei Chen.)*

Hongyang Chen, Benran Wang, Guangba Yu, Zilong He, and Pengfei Chen are with the School of Computer Science and Engineering, Sun Yat-sen University, Guangzhou 510275, China (e-mail: chenpf7@mail.sysu.edu.cn).

Chen Sun is with Huawei Technologies Company Ltd., Shenzhen 518000, China.

Zibin Zheng is with the School of Software Engineering, Sun Yat-sen University, Zhuhai 519040, China.

Digital Object Identifier 10.1109/TON.2025.3590034

reduce packet size by compressing telemetry data but require additional switch memory to maintain metadata.

2) **Existing anomaly detection methods are inefficient and unscalable.** To conserve bandwidth and reduce overhead, existing methods utilize trigger-based anomaly detection approaches [1], [19], [20], [21] that transmit monitoring data from data plane to control plane only when anomalies occur, such as sudden changes in end-to-end latency or queuing delay. However, traditional event trigger-based approaches typically rely on static or manually configured thresholds, which are impractical with numerous network flows. These methods require per-flow configuration and expert knowledge, and are generally initiated by the control plane, introducing additional latency and lacking adaptability in large-scale networks.
3) **Existing network failure diagnosis methods fail to comprehensively and automatically understand and identify fault causes due to their inability to extract insights from massive flows.** Jia et al. [13] rely on the set intersection of paths to locate the position of packet loss, but it lacks robustness and may fail when facing simple anomalies. IntSight [18] delegates diagnosis to the data plane but struggles with analyzing faults that propagate to adjacent switches. While query-based debugging [20] collects extensive data, it requires substantial expert knowledge, making it time-consuming and error-prone.

To overcome these limitations, NetScope is proposed to detect and diagnose performance-related **Net**work faults (e.g., microburst) that manifest as abnormal flow latency or packet loss, functioning as a "micro**Scope**" for network issues. NetScope is a comprehensive P4-based [22] system integrating low-cost telemetry, self-adaptive anomaly detection, and high-accuracy fault localization. Regardless of switch count, NetScope collects telemetry metadata (e.g., path sequences from path IDs, termed "path-aware") by inserting only one fixed-width telemetry header per packet, temporarily storing this data at edge switches (§V). Diagnostic telemetry data is transmitted to the control plane only when anomalies are detected in the data plane, reducing bandwidth consumption through on-demand collection (**Limitation 1**). For anomaly detection (§VI), NetScope introduces a novel per-Flow quantile estimation sketch that tracks latency medians in the data plane, dynamically updating detection thresholds for network delay detection (**Limitation 2**). For fault diagnosis (§VII), NetScope employs Frequent Sequence Mining (FSM) [23] to identify suspicious network positions using path information from telemetry data. Combined with scores calculated through Spectrum-Based Fault Localization (SBFL) [24] and telemetry metrics like throughput and queuing conditions, NetScope precisely determines port/switch/Flow-level root causes, providing operators with a prioritized list of culprits (**Limitation 3**). As NetScope does not inspect packet headers or payloads, it is not designed to detect application-layer attacks or deep-packet anomalies.

We demonstrate NetScope's superiority through extensive experiments (§VIII). Compared to in-network per-Flow quantile sketches, NetScope achieves a 44% relative improvement in estimation accuracy. With this sketch, NetScope delivers 32%∼116% better anomaly detection performance versus other baselines and 6%∼197% improvement in root cause analysis compared to alternatives. Additionally, NetScope operates with low overhead, requiring no network bandwidth during anomaly detection and consuming 64.2% less telemetry bandwidth for localization than competing approaches. To summarize, our work makes the following contributions.

- **On-demand Monitoring.** We propose a novel path-aware method to monitor network traffic and report data on demand. This method is independent of path length and does not incur additional costs as the network scales.
- **Self-adaptive Anomaly Detection.** We propose an in-network per-Flow quantile sketch. Based on it, we propose self-adaptive in-network anomaly detection, where the data plane accurately detects anomalies in a versatile network without the need of the control plane.
- **Automatic Root Cause Analysis.** We combine Frequent Sequence Mining and Spectrum-Based Fault Localization to precisely locate the faulty switch. The root cause can be automatically localized at different levels in a rank list.
- In the consideration of five different fault scenarios, we prove the effectiveness and low overhead of NetScope.

II. Background

SDN & PDP. Software Defined Networking (SDN) and Programming Data Plane (PDP) have gained widespread adoption for their programmability. SDN separates the control plane (decision-making) from the data plane (packets forwarding), enabling centralized management. PDP refers to the programmability of the SDN data plane, allowing switches to be configured with customized processing logic.

OpenFlow & P4. OpenFlow [25], an SDN communication protocol, abstracts traditional networking into flow tables and match-action list. P4 (Programming Protocol-independent Packet Processors) extends it by providing protocol independence via a high-level language that enables defining packet processing pipelines without dealing with hardware details.

Quantile Estimation. Quantile estimation, which characterizes data stream (e.g., network flows) distributions, has gained attention [26], [27], [28], [29] in network monitoring. The primary research focus involves designing various quantile sketches based on sketch data structures, which provide diverse summaries of a data stream while processing each data item only once. Specifically, a quantile sketch approximates quantile values of a single data stream of n elements. A query for quantile $\phi(\phi \in [0,1])$ estimates the $\lfloor n\phi \rfloor$th element. As network flow volumes grow, research has shifted from single-stream to multi-stream concurrent quantile estimation.

In-band Network Telemetry (INT). Proposed by P4 organization, INT [30] is a technique to monitor and collect network data within the data plane of a network. INT can gather telemetry data in real-time by adding or modifying fields of the packet header, without the requirement of external probes or monitoring agents. By gathering fine-grained and up-to-date information on network performance and behavior, INT helps network administrators resolve issues more efficiently.

III. Motivation

a) Reducing Monitoring Overhead of Core Switches:

To further reduce the monitoring overhead, SpiderMon [21] saves network bandwidth by collecting telemetry in the control plane on demand. It requires all switches to store data and report diagnosis data to the control plane. However, regardless of the type of network, core switches are always busier than

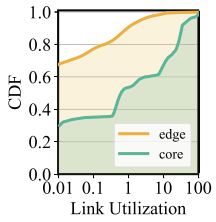


Fig. 2. The link utilization of the core layer and the edge layer.

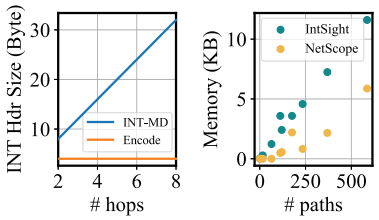


Fig. 3. Encoding path with ID can reduce INT header size. Among the methods that encode path, NetScope spares more switch memory.

edge switches. Benson et al. published network traffic datasets collected from various data centers [31]. We used one of these datasets to examine link utilization in both the core and edge layers by analyzing SNMP data from switches over several hundred 5-minute intervals. Fig. 2 displays the Cumulative Distribution Function (CDF) for a representative interval, clearly demonstrating that the possibility of low link utilization in the core layer is lower than in the edge layer [32]. To relieve the strain on core switches, NetScope is expected to offload the burden to edge switches by gathering telemetry data on edge switches and maintain the aggregation of telemetry. As a result, we are inspired to collect aggregated data at the packet level and record only in edge switches.

b) Reducing Header Size for Path Recording:

Packet transmission paths are crucial for root cause diagnosis. The INT-MD (eMbed Data) mode [13] records switch sequences by injecting telemetry data into packet headers at each hop, but this increases packet size as paths lengthen. To evaluate the impact of preserving path information, we measure the growth of INT header size as path length increases. During the experiment, only a 4-byte switch-ID [30] is injected in the packet as telemetry data per hop. The left part of Fig. 3 shows that header size increases proportionally with hop count, reaching 32 bytes after 8 hops. Previous studies [32], [33] have consistently shown that packet sizes in data center networks follow a bimodal distribution, with approximately 50% of packets being smaller than 200 bytes. While a 32-byte INT header represents only 16% of a 200-byte packet, its cumulative effect across numerous packets significantly impacts network bandwidth. To mitigate path transmission overhead, IntSight [18] encodes paths into fixed-width IDs but requires numerous Match-Action Table (MAT) entries for path ID maintenance, consuming switch memory. The right graph of Fig. 3 shows switch memory usage across network with varying path counts, confirming IntSight's substantial memory overhead. Specifically, IntSight consumes 11KB to support 586 paths. Although it seems modest, it takes up a significant portion of programmable switches' limited SRAM, where each pipeline stage typically has only 1MB SRAM shared among various data plane programs. Consequently, we propose a path-aware method that reduces monitoring overhead in larger networks while minimizing switch memory consumption.

c) Adaptively Detecting Anomalies Within Programmable Switches:

Existing methods fail to achieve a good trade-off between detection accuracy and inefficiency. Some [8], [34] require the controller to collect monitoring data from the data plane frequently and run detection models. Although these methods collect abundant data, the frequent data collection causes high extra overhead. Conversely, others [1], [19], [20], [21] rely on static thresholds preset in switches without onerous collection. However, expert knowledge is necessary for manually setting thresholds, which is not suitable for complex and rapidly evolving network. In particular, the vast number of flows and the highly variable latency distributions among these flows [32] make it challenging to configure effective static thresholds. Moreover, the latency of network flows varies over the day [35]. As a result, manually setting and updating static thresholds for each network flow is error-prone and costly. NetScope focuses on a better trade-off among detection accuracy and inefficiency by calculating adaptive thresholds entirely in the data plane of the programmable switch.

d) Automated and Intelligent Analysis of Diverse Root Causes:

Existing network diagnosis methods typically focus on analyzing a single type of failure [19], [28], [36]. While recent studies [18], [20], [21] uses in-band telemetry data to support the diagnosis of various types of faults, they rely on predefined static rules based on domain knowledge and a precise understanding of the specific fault. Such manual rules are difficult to maintain in dynamic and heterogeneous network. Additionally, with the increasing volume of in-band telemetry data, manually analyzing each flow to extract diagnostic evidence is inefficient and error-prone. Consequently, NetScope introduces an automated and data-driven root cause analysis method to enable accurate diagnosis of diverse network faults, which minimizes reliance on manual rules and expert intervention.

IV. Overview

A. Definition

*Definition (*Source/Transit/Sink *Switch):* For a packet traversing in the network, the switch that it first enters is referred to as the **source switch**. The switch that the packet last exits from, before reaching its destination host, is called **sink switch**. Other switches between source switch and sink switch are named as **transit switches**. Note that one switch may play multiple roles (source, transit, sink) for different Flows.

*Definition (*FlowID*):* FlowID is defined as $\langle s_{\text{source}}, s_{\text{sink}} \rangle$ without host information, as NetScope focuses on the problems that happen in the network, i.e., the anomaly between/in switches. Furthermore, FlowID spares more bits than 5-tuple.

*Definition (*PathID*):* PathID is updated per hop as the packet traverses across switches. At each hop, the PathID is updated through hash operation introduced in §V-C.

B. Overview

Fig. 4 provides an overview, illustrating both data plane and control plane. The design is driven by 4 key motivations in Section III. For **monitoring (§V)**, NetScope periodically

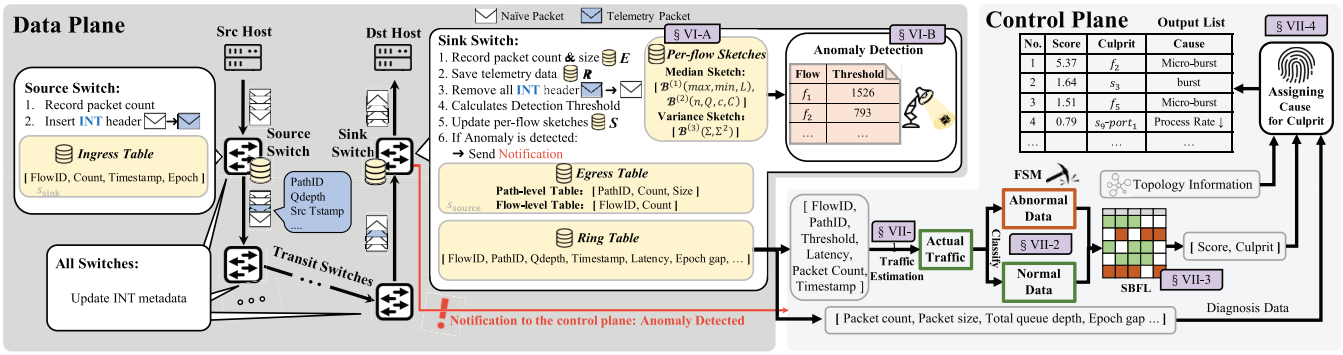


Fig. 4. The overview of NetScope.

samples network packets as telemetry packets and inserts critical information into telemetry headers. To reduce the overhead on core switches (Motivation a), NetScope stores traffic telemetry data in edge switches' memory and reported on demand. To reduce telemetry header size (Motivation b), NetScope compresses the entire path into a fixed-width ID, rather than recording hop-by-hop switch IDs. For **anomaly detection (§VI)**, the sink switch monitors for drop events or latency exceeding corresponding thresholds. If so, it sends a notification to the control plane. To balance detection accuracy and efficiency (Motivation c), NetScope calculates the adaptive threshold by estimating the median quantile and the variance of each Flow's latency data, with estimates updated upon each packet arrival. These estimations and threshold selections are performed within the sink switches. For **root cause analysis (§VII)**, culprits are located by FSM and scored by SBFL with minimal manual effort (Motivation d). Additional telemetry data helps pinpoint diverse fault causes across various levels, including port-level packet loss, switch-level ECMP load imbalance, and Flow-level micro-burst. After combining the report data, the control plane diagnoses root cause and sends a rank list of recommended culprits to operators.

V. Telemetry Collection

As network behavior typically remains stable in a short time under normal conditions [15], NetScope adopts a sample strategy to collect essential information without inducing substantial bandwidth overhead. The source switch periodically generates a sample packet to collect telemetry data, including statistics like the number of packets during the sample period, referred to as an **epoch**. Additionally, NetScope adopts a path-aware encoding method (§V-C) to reduce packet transmission overhead, enabling more efficient telemetry collection.

A. Packet Category

NetScope uses the IP layer's reserved field (e.g., IPv4 Option) to distinguish between naïve and telemetry packets, maintaining fixed packet size regardless of path length.

Naïve Packet. Naïve packet is a packet without a telemetry header inserted by the switch, as indicated by the white packet shown in Fig. 4. For every naïve packet, only a field of small size (e.g., 8 bits) is inserted into the header to record PathID, which will be carried and updated through the network.

Telemetry Packet. During each epoch, the source switch inserts an 11-byte telemetry header into a naïve packet per Flow, creating a telemetry packet (blue packet in Fig. 4). The controller can dynamically adjust epoch period. Telemetry packets from different Flows are staggered to mitigate transient network burden of telemetry data. The telemetry header includes the source timestamp (when the packet enters the source switch), the packet count of the packet's FlowID in the previous epoch, total queue depth, and telemetry epoch ID. The timestamp can be compressed to reduce size [21]. Specifically, the total queue depth is recalculated at each hop via in-network computing, i.e., by summing each switch's queue depth. Therefore, the size of telemetry data carried by packets remains constant regardless of path length.

For all packets, both source and sink switches count the total packet number and total packet size in each epoch.

B. Switch Actions

Switches perform both packet monitoring and anomaly detection. All switches update PathID for each packet. For telemetry packets, switches update telemetry data (e.g., total queue depth) in telemetry headers. Sink switches perform anomaly detection, notifying the control plane when latency thresholds are exceeded or drop events are detected (§VI). To conserve resources, each sink switch sends at most one notification per an epoch. To prepare and record telemetry data for diagnosis, edge switches (source switches and sink switches) have additional tasks to perform.

Source Switch handles telemetry data in two steps. **(1)** It counts the number of incoming packets of each Flow per epoch and records them in the Ingress Table (IT). **(2)** It inserts PathID filed in all packets and also modifies reserved fields to distinguish between naïve and telemetry packets. IT records the timestamp and epoch ID of the latest telemetry packet per-Flow, enabling only one telemetry packet per Flow per epoch.

Sink Switch stores telemetry data in three steps. **(1)** For all incoming packets, it records the packet count and packet size corresponding to PathID and FlowID in the Egress Table (ET). **(2)** For telemetry packets, it extracts the telemetry data to the Ring Table (RT). In addition, RT records the latency, packet counts, and packet size at path-level and Flow-level in this epoch. When RT is full, the oldest data will be covered by the newest data, that is why the table is called as "ring". As a result, RT retains the most recently telemetry data. **(3)** All INT headers will be removed at the end of the sink switch, ensuring the monitoring is transparent to end hosts.

Note that the node ID of the source switch and sink switch already covers half information of FlowID ($\langle s_{\text{source}}, s_{\text{sink}} \rangle$). Hence, the FlowID can be simplified as s_{sink} at source switches and s_{source} at sink switches, saving the switch's memory usage.

C. Path-Aware Encoding

In complex networks with dynamic routing strategies like different load balancing methods, packets from the same flow may traverse different paths. For instance, in Fig. 1, packets of a Flow whose FlowID is $\langle s_1, s_4 \rangle$ are transmitted along two paths $\langle s_1, s_3, s_4 \rangle$ and $\langle s_1, s_2, s_4 \rangle$. To efficiently record packet paths without excessive telemetry overhead, NetScope adopts a path-aware encoding method that assigns a PathID to each traversed path. Specifically, PathID is updated as the packet traverses across switches. At each hop, the updated PathID is hashed by {PathID, switchID, ingress port, egress port, control}. The control field defaults to zero and changes only when the hashed value has conflicts with another Flow. Consequently, NetScope installs MAT entries only when a hash conflict happens, reducing switch memory consumption.

In the control plane, NetScope calculates PathID for each potential path using the same hashing algorithm. For each Flow, NetScope enumerates all possible paths between source and sink switches and performs per-hop hash updates with a corresponding control option array for each candidate path. The control option array records the control value per hop along the path. If the generated PathID have conflicts with an existing one, the control option array is updated and the hash computation is repeated until the conflict is resolved. Because both the control and data planes use a consistent hashing scheme, the control plane can decompress fixed size bits field to path information from the data plane's report in later fault localization analysis.

VI. Anomaly Detection

The left part of Fig. 4 shows an example of the detection process in practice. When a packet reaches the sink switch, NetScope parses the INT metadata and calculates the Flow's detection threshold using its estimated median and deviation stored in per-Flow sketches (*median sketch* and *variance sketch*) as in §VI-A. These sketches are then updated with the received packet's latency. NetScope compares packet latency against the threshold. If the latency exceeds it, an anomaly is detected and a notification is sent to the control plane. Additionally, NetScope detects drop events with packet counts and epoch ID. The further details are described in §VI-B.

A. Per-Flow Latency Quantile Estimation

Tracking latency quantiles across multiple network Flows in programmable switches is challenging due to constrained on-chip memory and limited processing flexibility, making it feasible to store or sort latency per flow. Recent studies [29], [37], [38] have proposed quantile estimation methods. EasyQuantile [29] supports fully in-data-plane quantile tracking without control-plane interaction, but is limited to single-flow scenarios. SketchPolymer [37] and HistSketch [38] extend support to per-flow estimation but require multiple control plane queries to retrieve results. In contrast, NetScope introduces a per-flow quantile estimation method that operates entirely in the data plane without the need of control plane. This method selectively updates estimated quantile values based on whether their current rank deviates from the real rank. Specifically, the real rank of an estimated quantile is its position within sorted observed latency samples, while the current rank represents the number of packets with latency below the current estimate. NetScope determines when to adjust estimated quantiles by comparing these ranks, rendering quantile estimation without explicitly storing or sorting the latency values. This procedure includes 2 stages: **Step Calculation** and **Quantile Estimation**.

1) Step Calculation Stage: The estimated quantile value adjusts when its current rank deviates from the real rank, with adjustment magnitude denoted as step β_i for Flow *i*. This step is updated upon receiving new latency data. Since precisely calculating exact ranks or determining the rank of specific values is challenging, β_i is approximated using the average of maximum and minimum latency intervals between adjacent packets, enabling quantile estimation through interpolation between two values whose ranks are near the target rank.

Algorithm 1 Quantile Estimation: Step Update

Input: 5-tuple T_i and latency $l_{i,j}$ of packet j in Flow i
for $1 \le k \le d$ **do**
  $D \leftarrow |l_{i,j} - \mathcal{B}_k^{(1)}[h_k(T_i)].L|$
  $\mathcal{B}_k^{(1)}[h_k(T_i)].max \leftarrow \max\{D, \mathcal{B}_k^{(1)}[h_k(T_i)].max\}$
  $\mathcal{B}_k^{(1)}[h_k(T_i)].min \leftarrow \min\{D, \mathcal{B}_k^{(1)}[h_k(T_i)].min\}$
  $\mathcal{B}_k^{(1)}[h_k(T_i)].L \leftarrow l_{i,j}$
end

Algorithm 2 Quantile Estimation: Step Query

Input: 5-tuples T_i of Flow i
Output: Step β_i for Flow i
$\beta_i \leftarrow +\infty$
for $1 \le k \le d$ **do**
  $t \leftarrow (\mathcal{B}_k^{(1)}[h_k(T_i)].min + \mathcal{B}_k^{(1)}[h_k(T_i)].max)/2$
  $\beta_i \leftarrow \min\{\beta_i, t\}$
end
return β_i

In this stage, we use a Count-Min Sketch to approximate the maximum and minimum intervals. The Count-Min sketch [39] consists of *d* arrays, each with *n* buckets. Each Flow is hashed into one bucket per array using *d* hash functions $h_1, \ldots, h_d$. These buckets are updated independently, with their minimum approximating the Flow's statistics. In our design, each bucket has three fields including the maximum interval ($\max$), the minimum interval ($\min$) and latency (*L*). The maximum and minimum interval fields record the maximum and minimum latency intervals for a Flow, while the latency field records the latency of the preceding packet. As shown in Algorithms 1–2, upon the arrival of a packet *j* of Flow *i* with IP 5-tuple T_i, *d* hash functions map T_i to *d* buckets $\mathcal{B}_1^{(1)}[h_1(T_i)], \ldots, \mathcal{B}_d^{(1)}[h_d(T_i)]$. We calculate *D* as the difference between current latency $l_{i,j}$ and the preceding packet's stored latency. The maximum (minimum) interval field updates to the maximum (minimum) of its initial value and *D*, while the latency field becomes $l_{i,j}$. When querying β_i, the maximum and minimum interval fields of these *d* buckets are averaged to *t*, and the step β_i of Flow *i* is the minimum of *t* and the original step β_i.

2) Quantile Update Stage: This stage evaluates whether the current rank of the estimated quantile $\hat{Q}_\phi$ deviates from its actual rank and updates it accordingly. Algorithm 3 describes the update procedure which uses a *d*-array Count-min sketch with *n* buckets per array. Each bucket has four fields: packet count field (*n*), estimated quantile field ($\hat{Q}_\phi$) and two fields

Algorithm 3 Quantile Estimation: Quantile Update

Input: 5-tuples T_i, latency $l_{i,j}$, and Step β_i of Flow i, quantile ϕ

1 **for** $1 \le k \le d$ **do**
2 $\quad \mathcal{B}_k^{(2)}[h_k(T_i)].n \leftarrow \mathcal{B}_k^{(2)}[h_k(T_i)].n + 1$
3 $\quad \delta \leftarrow \mathcal{B}_k^{(2)}[h_k(T_i)].n \times \phi$
4 $\quad$ **if** $l_{i,j} < \mathcal{B}_k^{(2)}[h_k(T_i)].\hat{Q}_\phi$ **then**
5 $\quad\quad$ **if** $\mathcal{B}_k^{(2)}[h_k(T_i)].c + 1 > \mathcal{B}_k^{(2)}[h_k(T_i)].n - \delta$ **then**
6 $\quad\quad\quad \mathcal{B}_k^{(2)}[h_k(T_i)].\hat{Q}_\phi \leftarrow \mathcal{B}_k^{(2)}[h_k(T_i)].\hat{Q}_\phi - \beta_i$
7 $\quad\quad\quad \mathcal{B}_k^{(2)}[h_k(T_i)].C \leftarrow \mathcal{B}_k^{(2)}[h_k(T_i)].C + 1$
8 $\quad\quad$ **end**
9 $\quad\quad$ **else**
10 $\quad\quad\quad \mathcal{B}_k^{(2)}[h_k(T_i)].c \leftarrow \mathcal{B}_k^{(2)}[h_k(T_i)].c + 1$
11 $\quad\quad$ **end**
12 $\quad$ **end**
13 $\quad$ **if** $l_{i,j} > \mathcal{B}_k^{(2)}[h_k(T_i)].\hat{Q}_\phi$ **then**
14 $\quad\quad$ **if** $\mathcal{B}_k^{(2)}[h_k(T_i)].C + 1 > \delta$ **then**
15 $\quad\quad\quad \mathcal{B}_k^{(2)}[h_k(T_i)].\hat{Q}_\phi \leftarrow \mathcal{B}_k^{(2)}[h_k(T_i)].\hat{Q}_\phi + \beta_i$
16 $\quad\quad\quad \mathcal{B}_k^{(2)}[h_k(T_i)].c \leftarrow \mathcal{B}_k^{(2)}[h_k(T_i)].c - 1$
17 $\quad\quad$ **end**
18 $\quad\quad$ **else**
19 $\quad\quad\quad \mathcal{B}_k^{(2)}[h_k(T_i)].C \leftarrow \mathcal{B}_k^{(2)}[h_k(T_i)].C + 1$
20 $\quad\quad$ **end**
21 $\quad$ **end**
22 **end**

Algorithm 4 Quantile Estimation: Quantile Query

Input: 5-tuples T_i of Flow i
Output: Estimated quantile of Flow i

1 $\hat{Q}_\phi^i \leftarrow +\infty$
2 **for** $1 \le k \le d$ **do**
3 $\quad Q_i \leftarrow \min\{Q_i, \mathcal{B}_k^{(2)}[h_k(T_i)].\hat{Q}_\phi\}$
4 **end**
5 **return** $\hat{Q}_\phi^i$

(c and C) record the number of packets with latency smaller or greater than $\hat{Q}_\phi$. The received packet with IP 5-tuple T_i is mapped to buckets $\mathcal{B}_1^{(2)}[h_1(T_i)], \ldots, \mathcal{B}_d^{(2)}[h_d(T_i)]$. If $\hat{Q}_\phi$ is the true ϕ-quantile, c and C should be $n \times \phi$ and $n \times (1-\phi)$ after n packets are received [29], which are set as their upperbound thresholds. The update of $\hat{Q}_\phi$ is determined by whether c and C exceed their corresponding thresholds. In details, when a received packet whose latency $l_{i,j}$ is below $\hat{Q}_\phi$ (Lines **4-12**), c is compared with the threshold $n \times \phi$. If c exceeds the threshold, it indicates that $\hat{Q}_\phi$ is larger than the true ϕ-quantile. Consequently, $\hat{Q}_\phi$ is updated by subtracting the step β_i and C is incremented by one. Otherwise, c is incremented by one. Conversely, when $l_{i,j}$ is greater than $\hat{Q}_\phi$, the update procedure (Lines **13-21**) is similar, except the comparison is conducted with C and its threshold $n \times (1-\phi)$. To query the estimated quantile Q_i for Flow i, the minimum of the estimated quantile fields in these d buckets is returned, as shown in Algorithm 4.

3) Mathematical Analysis: We now present an error bound for the quantile estimation algorithm, which is determined by sketch-based rank estimation (§VI-A.2) and step adjustment (§VI-A.1). Specifically, the rank of the current estimate is implicitly tracked using two sketch bucket fields c and C. Due to space constraint, we only list the conclusion in this section. Detailed proofs can be seen in Appendix.

Theorem 1: Let Q_ϕ and $\hat{Q}_\phi$ be the true and the estimated ϕ-quantile of a flow i after observing n_i packets. Let the Count-Min sketch used in rank tracking have estimation error bounded by ϵn_i with failure probability at most δ. Assume the estimated step β_i is a conservative upper bound on the latency difference between adjacent ranks for flow i. Then the estimation error satisfies:

$$\mathbb{P}\left(|\hat{Q}_\phi - Q_\phi| \le 2\beta_i \cdot \left\lceil \frac{\epsilon n_i}{\min(\phi, 1-\phi)} \right\rceil\right) \ge 1 - \delta \quad (1)$$

Proof: See in Appendix.□

B. In-Network Anomaly Detection

1) High Latency Detection: During packet traversal, sink switches detect anomalies by checking if packet latency exceeds Flow-specific thresholds, alerting the control plane when high latency occurs. Due to limited arithmetic operations and constrained memory space provided by switches, implementing a complex algorithm to choose adaptive thresholds is challenging in switches. Thus, NetScope employs statistical anomaly detection using the $\mu + 3\sigma$ rule [40]. Since median values remain more stable than averages [41] when dealing with latency outliers, the threshold θ for detecting outliers is calculated as $\theta = m + 3\sigma$, where m is the median and σ is the standard deviation.

a) Calculation of Median: Given n as the number of received packets of Flow i and $l_i = l_{i,1}, \cdots, l_{i,n}$ as their latency, the proposed quantile estimation method is used to compute the median m_i for l_i, with the quantile ϕ set to 0.5. The median of each Flow is estimated with the *Median Sketch*.

b) Calculation of Variance: The variance of latency data of Flow i is computed following the formula: $\sigma_i^2(l_i) = E(l_i^2) - E(l_i)^2 = \frac{\sum_{k=1}^n l_{i,k}^2}{n} - \left(\frac{\sum_{k=1}^n l_{i,k}}{n}\right)^2$. However, due to the lack of support for division operations in switches, $\sigma_i^2(l_i)$ is converted to $\sigma_i^2(n \cdot l_i) = E((n \cdot l_i)^2) - E(n \cdot l_i)^2 = n \cdot \sum_{k=1}^n l_{i,k}^2 - (\sum_{k=1}^n l_{i,k})^2$. NetScope adopts a d-array Count-min sketch (Variance Sketch) to calculate the variance of each Flow's latency. Each array has n buckets. Each bucket uses two fields to respectively save the sum (Σ) and the perfect square sum (Σ^2) of each Flow's latency data. d hash functions are applied to map the Flow i whose IP 5-tuple is T_i to buckets $\mathcal{B}_1^{(3)}[h_1(T_i)], \ldots, \mathcal{B}_d^{(3)}[h_d(T_i)]$ and the buckets fields are updated accordingly.

c) Calculation of Standard Deviation: Given the variance $\sigma_i^2(n \cdot l_i)$ for Flow i, the standard deviation is $\sigma_i(n \cdot l_i) = \sqrt{\sigma_i^2(n \cdot l_i)}$. However, since the *sqrt* operation is not supported in P4, NetScope adopts a similar approximation algorithm as described in [42]. Specifically, to get the square root of s, the integer representation bit string b of s is first used to compute the floating point representation f by setting the exponent of f as the index of the most significant bit (MSB) of b and setting the mantissa of f as the bits after the MSB of b. Then f is shifted to the right by one and the result f_1 is converted back to its integer representation b_1. The MSB of b_1 is set as the exponent of f_1 and the least significant bits are set as the mantissa of f_1. The integer representation b_1 is the square root of s.

d) Detection Workflow: For the latency data of Flow i, NetScope instead tracks the standard deviation $\sigma_i(n \cdot l_i)$, which characterizes the distribution $n \cdot l_i = \{n \cdot l_{i,1}, \ldots, n \cdot l_{i,n}\}$. Similarly, the quantile of $n \cdot l_i$ is computed as $n \cdot m_i$ and the threshold is obtained as $\theta'_i(n) = n \cdot m_i + t \cdot \sigma_i(n \cdot l_i)$, where t is the hyper parameter set as 3 in this study, following the widely used $\mu + 3\sigma$ rule in statistical anomaly detection. The choice of 3 corresponds to a 99.7% confidence interval in a near-normal distribution, enabling NetScope to detect statistically significant latency spikes. Upon detection, the latency of packet $n+1$ of Flow i is magnified by $n+1$ times and compared with the threshold $\theta'_i(n)$. Subsequently, the quantile and the standard deviation is updated which are then used to recompute threshold $\theta'_i(n+1)$.

2) Drop Event Detection: If packet loss occurs in a epoch, packet counts at the source switch (c_s) and sink switch (c_d) differ, with the difference $c_s - c_d$ indicating dropped packets. When it exceeds a threshold, the switch sends a notification to the control plane. For persistent packet drops spanning multiple epochs, the source switch compares telemetry packet epoch IDs, where non-adjacent epoch IDs indicate a drop event. The gap between epoch IDs reveals the drop event duration, shown in the "epoch gap" field in RT.

VII. Root Cause Analysis

The right part of Fig. 4 shows an example of the root cause analysis process in practice. Upon receiving a data plane notification, the control plane initiates diagnosis by retrieving recent telemetry data (*diagnosis data*) from sink switches' memory (Ring Table). To prevent notification floods, each switch only sends one notification per epoch, and the control plane only processes one notification per switch per epoch. NetScope identifies root causes using this telemetry data. NetScope also retains per-Flow thresholds from the data plane. As depicted in Fig. 4, the root causes analysis comprises four parts: **(1)** NetScope estimates actual traffic from sample data (§VII-.1) and classifies it into abnormal and normal sets using the obtained thresholds. **(2)** NetScope applies FSM on the abnormal set to mine frequent sequence patterns, identifying them as suspect anomaly locations, termed culprits (§VII-.2). **(3)** Then, NetScope integrates risk ratios into SBFL to score each culprit using both sets. Patterns frequently appearing in the abnormal set but rarely in the normal set receive higher scores (§VII-.3). **(4)** With diagnosis data, NetScope assigns causes to each culprit through signature matching and calculates cause scores based on culprit scores. Finally, NetScope merges duplicate causes and outputs a prioritized list of culprits and their causes (§VII-.4).

1) Actual Traffic Estimation:

As NetScope adopts a sampling strategy to collect telemetry data where only one telemetry packet is transmitted per epoch, actual traffic must be estimated for accurate root cause analysis. Due to varying packet counts across paths per epoch, NetScope reconstructs the actual traffic from sample telemetry packets recording packet counts ($p.count$) per path (PathID) in each sample epoch. As shown in Alg. 5, NetScope iterates through each PathID and its corresponding sample packets S_{PathID}. For each sample p with count $p.count$, NetScope replicates it $p.count$ times to form an estimated packet set. NetScope then adopts an interpolation strategy [43] to assign inferred arrival timestamps to each estimated packet. Specifically, for a sample packet occurring at time $p.t$ with inter-sample time gap T, the timestamp of the i-th replicated packet is calculated as $\hat{p}.t = p.t + \frac{i \times T}{p.count}$. This estimation enables NetScope to reconstruct the estimated packet set per path per sample epoch and each packet contains its arrival timestamp and observed latency. These packets are then classified into abnormal and normal sets based on whether their latency exceed the per-Flow threshold retrieved from the switch's data plane (§ VI).

Algorithm 5 Actual Traffic Estimation

Input: Sample packets S, Time gap between sample pkts T
Output: Estimated packets E

```
for PathID in S do
    for p ∈ S_PathID do ▷ per pkt from samples of PathID
        for i ∈ range(p.count) do
            p̂ ← copy p        ▷ copy sample as an estimation
            p̂.t ← p.t + (i×T)/p.count        ▷ estimate pkt's timestamp
            E ← E ∪ {p̂}
        end
    end
end
```

2) Frequent Sequence Mining (FSM):

To localize culprits, NetScope analyzes paths of abnormal packets and identifies sub-sequences of switches that are potentially responsible for anomalies. Each packet's path represents its traversed switch sequence. Although a length-L path theoretically contains up to $\frac{1}{2}L(L+1)$ contiguous subsequences, only those of length one or two are considered meaningful—representing individual switches and adjacent links, respectively. This design is based on the observation that switches or links that appear frequently in abnormal paths but rarely in normal ones are more likely to be the cause of failures. For instance, from a packet path $\langle s_1, s_2, s_3, s_4 \rangle$, subsequences like $\langle s_2 \rangle$ and $\langle s_3, s_4 \rangle$ represent a switch or a link. Under this constraint, NetScope restricts its analysis to at most $2L-1$ meaningful patterns per path. This significantly reduces the pattern space while retaining the most relevant features for root cause analysis. Nevertheless, even with this restriction, naïvely enumeration of these sub-sequences across many paths still remain computationally expensive in large-scale networks. To efficiently identify the most frequent sub-sequence in the abnormal set, Frequent Sequence Mining (FSM) is a prominent solution. Using depth-first search or pattern-growth techinques, FSM prunes infrequent sequences early to accelerate the algorithm and return frequent sequence patterns. Our evaluation (§VIII-E.4) shows that PrefixSpan [23] performs the best, requiring minimum running time and relative less memory. FSM algorithms process a list of sequences and output frequent sequence patterns with *support*. The *support* of sequence s_a is defined as the number of sequences that contain s_a. In NetScope, the frequent sequence pattern is switch or link (two switches).

Specifically, if the control plane receives data with 4 $path_1 = \langle s_3, s_2, s_4 \rangle$ and 2 $path_2 = \langle s_6, s_2, s_7 \rangle$, with FSM configured to set the max pattern length as 2 and the min relative support as 50%, i.e., $\min support = (4+2)/2 = 3$ in this case, the frequent sequence patterns are ($\langle s_2 \rangle$, $\langle s_2, s_4 \rangle$, $\langle s_3 \rangle$,

$\langle s_3, s_2\rangle$, $\langle s_4\rangle$). The pattern $\langle s_2\rangle$ has the highest *support*(6), while others have support of 4. Patterns like $\langle s_6\rangle$ are pruned because their *support* is lower than the minimum support.

3) Spectrum-Based Fault Localization (SBFL):

After indentifying frequent sequences as culprits, NetScope ranks them to filter out *Top-N* culprits. As a program fault localization approach, SBFL uses various program spectra from software tests and corresponding test results to calculate each test case's suspicious score [24]. SBFL is extended to the network domain, ranking each culprit sequences pattern by comparing the proportion of normal/abnormal datasets with/without the pattern. Relative risk, a statistical analysis technique in medical studies [44], is integrated into SBFL by NetScope to calculate the suspicious score of each pattern as

$$Score(pattern) = \frac{N_{pf}/(N_{pf}+N_{ps})}{N_{nf}/(N_{nf}+N_{ns})}, \tag{2}$$

where N_{pf} is the number of packets in abnormal set (**f**ailing test) whose path contains the specific **p**attern, N_{ps} is the number of packets in normal set (**s**uccessful test) whose path contains the specific **p**attern, N_{nf} and N_{ns} is the number of packets whose path does **n**ot contain the specific *pattern* in **f**ailing test and **s**uccessful test. To prevent division by zero, N_{nf} is incremented by 1 when the processing dataset is too small and all abnormal data share a same pattern. In this case, the equation variation can be written as $(N_{pf}/(N_{pf}+N_{ps}))/((N_{nf}+1)/(N_{nf}+N_{ns}))$. The numerator of (2) is the abnormal proportion of this *pattern*, and the denominator represents the abnormal proportion of other patterns. A higher score value indicates a higher possibility that the pattern is the source of failure, as compared to other patterns.

Algorithm 6 Culprit Localization

Input: Frequent pattern F, Diagnosis data D
Output: Culprits Set C

```
for pattern ∈ F do
    for Flow that traverse pattern do
        culprit.cause ← SignatureD
        culprit.score ← pattern.score · (# pkts of Flow) / (# pkts go through pattern)
        C ← C ∪ culprit
    end
end
C ←mergeC
```

4) Culprit Localization:

The root cause of node/link abnormalities varies. To give comprehensive help to network operators, NetScope uses signature matching to assign causes to culprit patterns with telemetry data. By querying paths containing frequent patterns in the diagnosis data, NetScope determines causes based on edge switch packet counts and sizes (to calculate throughput), total queue depth along the path, and the topology information. Each culprit will be assigned a score calculated by multiplying the pattern score by the corresponding path proportion, as shown in Alg. 6.

We give 5 signatures for common root causes at Flow level, switch level, and port level. The signatures can be extended

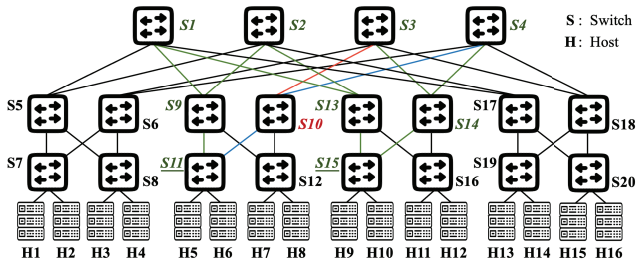


Fig. 5. Fat-tree topology.

to include additional root causes. We focus on these causes as they frequently occur in real-world network operations and significantly impact network performance and stability.

(1) Micro-burst is a Flow-level cause. It is a short-lived spike of flow that exceeds average traffic, leading to queue buildup and resulting in a high latency or packet loss [32]. Its signature is whether a Flow's pps (packets per seconds) raises sharply in the problematic period.

(2) ECMP Load Imbalance is a switch-level cause. While multiple equal paths typically exist between edge switches (e.g., four paths from s_{11} to s_{15} in Fig. 5), imperfect hash algorithm and uneven Flow distribution under time and space can create load imbalances [45]. For example, if s_9 fails to balance the traffic between two paths, more Flows are set towards to s_1. Though high delay happens at s_1 with queue building up, the root cause is at another switch s_9. The signature of the root cause is whether the throughput of each path in an ECMP group is evenly distributed when a switch's queue experiences sudden congestion.

(3) Process Rate Decrease is a port/switch-level cause. Due to limitation of resource (switch CPU, memory, etc.) or imperfect schedule scheme, the processing rate of the switch may decrease. As switch cannot process packet in time, a low process rate will bring about queuing buildup, therefore the latency raises up. Its signature is that pps remains relatively stable when a queue buildup occurs.

(4) Delay is a port/switch-level cause. Besides process rate decrease, switch errors like interrupts, insufficient power supply and configuration errors can result in delays outside the queue. The identifying characteristic of this root cause is that there is not obvious change in both pps and queue depth, yet the culprit pattern still has a high suspicious score.

(5) Drop is a port/switch-level cause. While transient drops may occur due high latency faults, unanticipated packet loss can also result from link failure like poor cable connection, unwell-behaved network updates, and missing forwarding table entries [1]. NetScope detects drops by verifying the sequence ID of the sample epoch. Unlike latency-related root causes, NetScope applies different analytical logic for drop event diagnosis. The time range of the drop event is identified through the "epoch gap" field in diagnosis data (as in §VI-B.2). NetScope calculates the number of dropped packets with the packet counter in the source and sink switches to identify the affected Flows and estimate the number of dropped packets. These Flows are then categorized into the abnormal set, while unrelated Flows form normal set. NetScope then runs another instance of SBFL with these sets, identifying the link or switch with the highest suspicion score as the most probable drop event culprit.

While different frequent patterns may attribute to the same culprit, NetScope merges the same cause at last. The actual

abnormal localization dominates the causes' score eventually, i.e., the cause score should be lower than the SBFL score of the culprit pattern. Therefore, the merged score of a Flow-level cause is the maximum score of repeat items, while other kinds of causes' merged score is calculated by summation. In addition, NetScope merges port-level causes of the same type into a single switch-level cause when they are assigned to multiple ports within the same switch.

VIII. Evaluation

A. Research Questions

We conduct experiments to answer the research questions.

Quantile Estimation Methods Evaluations:

- **RQ1. Estimation Accuracy.** Can the quantile estimation method of NetScope outperform other baseline method in providing a more accurate estimation?
- **RQ2. Effect of Memory on Estimation Accuracy.** Can the quantile estimation of NetScope achieve better accuracy under a fixed target memory usage?
- **RQ3. Estimation Efficiency I.** What is the efficiency of update operations of different methods?
- **RQ4. Estimation Efficiency II.** What is the efficiency of query operations of different methods?

Anomaly Detection & Root Cause Analysis Evaluations:

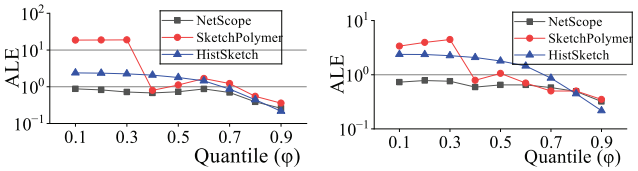
- **RQ5. Anomaly Detection Effectiveness.** Can NetScope outperforms other baselines in detecting anomalies?
- **RQ6. Root Cause Analysis Effectiveness.** Can NetScope outperform other baselines in analysing root causes?

Overhead Evaluations:

- **RQ7. Network Bandwidth Overhead.** What is the diagnosis overhead of NetScope on Network Bandwidth?
- **RQ8. Switch Resource Overhead.** What is the overhead of NetScope on switch resources?
- **RQ9. Latency Overhead.** What is the overhead of the NetScope INT collection on packet transmission latency?
- **RQ10. FSM Algorithms Efficiency.** What is the efficiency of different FSM algorithms?

B. Experiment Setup

The evaluation is performed in 2 complementary environments. The performance and accuracy of quantile estimation, anomaly detection, and root cause analysis are assessed in a simulation environment supporting flexible testing of large-scale typologies like fat-trees. The simulation uses Mininet with BMv2 [46] software P4 switches on a physical machine with 56-core CPU, 32 GB memory. While BMv2 does not reflect hardware-level latency or throughput, it faithfully executes P4 logic and supports accurate algorithmic validation. In contrast, the overhead of NetScope is evaluated in a physical environment using a Barefoot Tofino switch [47] running ONL (Open Network Linux) [48] as the switch OS, equipped with the Tofino ASIC for hardware packet processing and an 8-core CPU for the control plane program. The Tofino switch emulates multiple logical switches connected port-to-port at 10 Gbps and connects to Linux hosts via fiber links. The P4 code in the data plane consists of 1493 LOC (lines of code). The control plane is implemented in Python with 855 LOC, while root cause analysis uses the SPMF data-mining library [49], with 793 LOC written in Python. Evaluation code is available at https://github.com/chenhy97/NetScope.



(a) Web (b) Seattle

Fig. 6. ALE on two datasets.

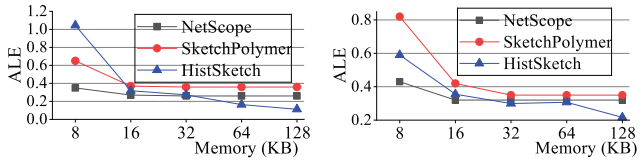
C. Quantile Estimation Comparisons

1) Evaluation Baseline and Metrics: NetScope is compared to SketchPolymer [37] and HistSketch [38]. SketchPolymer uses one sketch to measure per-item tail quantile in switches and HistSktech proposes a histogram-in-sketch design to support per-flow distribution monitoring. However, both rely on control plane to perform query and calculation to estimate. Moreover, the open-source implementation of HistSketch is executed on the CPU and does not provide a switch version. Therefore, in our evaluation, we compare SketchPolymer and HistSketch in terms of estimation accuracy and memory-accuracy trade-off, while SketchPolymer is used for efficiency comparisons, as HistSketch does not support in-switch updating. The comparisons are conducted through experiments repeated for 10 times, with average results calculated.

- **Average Logarithm Error (ALE)** measures the estimation accuracy. Suppose $L_1, \ldots, L_n$ are the true latency quantile of all network flows, and $\hat{L}_1, \ldots, \hat{L}_n$ are the estimated latency quantiles, ALE is defined as $\frac{1}{n}\sum_{i=1}^{n} |log_2 \frac{L_i}{\hat{L}_i}|$.
- **Average Latency** measures the efficiency of update operations. It measures the average time it takes for packets to travel from source to destination, reflecting the impact of update operations on transmission.
- **Average Query Time** measures the average time taken to query the estimated quantile for each flow.

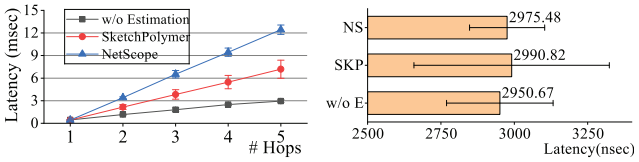
2) Datasets: We use 2 real-world datasets: the Seattle Dataset [50] and the Web Latency Dataset [51]. The former includes round trip times (RTTs) between several nodes in the Seattle network. We regard RTTs between two nodes as latency between them. The latter collects the fetch time of requests using Webget [51]. We regard the fetch time of requests to the same destination as the latency of a flow.

3) RQ1: Estimation Accuracy: We evaluate the accuracy of different methods under various quantiles ϕ ranging from 0.1 to 0.9 using the same memory budget. Fig. 6 shows their accuracy on two datasets. NetScope consistently achieves the highest accuracy across quantiles. Compared to SketchPolymer, it reduces the ALE by an average of 44%. When ϕ is less than 0.3, SketchPolymer provides a more inaccurate estimation. Specifically, its ALE is about $20\times$ higher than NetScope on the Web dataset and about $4\times$ higher than NetScope on the Seattle dataset. The suboptimal performance of SketchPolymer under smaller quantile is attributed to its stage that records the frequency of each value. In this stage, SketchPolymer uses a Count-Min sketch to record frequency and employs hash functions to map the combination of the network 5-tuple identifier and the corresponding latency into a specific bucket. Due to the variance of the latency, the probability of hash collisions increases as the expected quantile decreases, leading to the suboptimal performance. HistSketch achieves moderate accuracy across most quantiles, outperforming SketchPolymer under small quantile. However, it still shows 62.5% higher



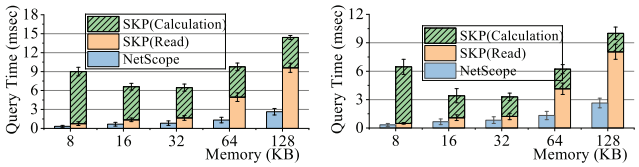
(a) Web (b) Seattle

Fig. 7. ALE on two datasets with different memory.



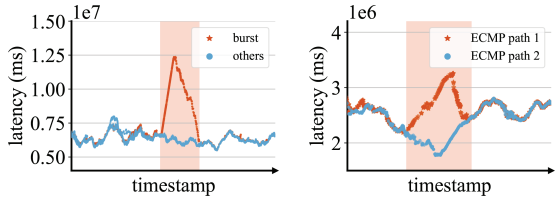
(a) Update on per hop (b) Update on last hop

Fig. 8. Average Latency of update operations on two cases.



(a) Webget (b) Seattle

Fig. 9. Average query time on two datasets.



(a) Micro-burst (b) ECMP

Fig. 10. Examples of anomaly scenario.

ALE than NetScope on average. Moreover, since HistSketch is implemented on CPU and requires control-plane cooperation to compute quantiles, it does not support online in-switch quantile querying like NetScope.

4) RQ2: Effect of Memory on Estimation Accuracy: Fig. 7 shows the 0.9-quantile estimation accuracy across various memory allocations. Since programmable switches provide only about 1 MB per stage [52], the allocated memory ranges from 8KB to 128KB. NetScope is more accurate than SketchPolymer with the same memory limit. NetScope achieves optimal performance (0.3 ALE on average) using merely 16KB, while SketchPolymer requires 32 KB to achieve its best (0.4 ALE on average). This is because NetScope only maps the 5-tuple identifier to Count-min sketch buckets, while some stages of SketchPolymer map the combination of the 5-tuple identifier and the corresponding latency value to the bucket. The variance in latency values increases the probability of hash collisions, necessitating more memory for SketchPolymer to reduce these collisions. HistSketch achieves lower ALE at large memory sizes (e.g., 64KB and 128KB), but remains less accurate with less memory.

5) RQ3: Estimation Efficiency I: Fig. 8 presents the average packet transmission time when update operations of NetScope and SketchPolymer are performed either on all switches along paths of various lengths (number of hops) (Fig. 8(a)) or exclusively on the final switch of 5-hop paths (Fig. 8(b)). These results are obtained from the simulation environment where packet processing is executed by BMv2, a software switch running on CPU. The measured delays include host stack scheduling and software forwarding latency, reflecting end-to-end latency in emulation. This differs from Tofino switch, where per-hop packet processing occurs at line rate. Although the absolute latency values are higher than those in hardware, the relative trends between methods remain valid. The update operations of NetScope take more time than those of SketchPolymer, with costs escalating as hop count rises due to the more complex update logic of NetScope. However, as Fig. 8(b) shows, if only the last switch performs updates, average latencies of both methods become comparable, exceeding the baseline (no switch performs estimation) by merely 0.84%∼1.36%. Therefore, since NetScope conducts the anomaly detection at the edge switch, its update operations have the minimal impact on transmission.

6) RQ4: Estimation Efficiency II: Fig.9 presents the average execution time to get per-flow quantiles. NetScope significantly outperforms SketchPolymer in query efficiency, reducing average query time by 87.4%. Specifically, NetScope's reduces 89.5% query time on the Webget dataset and 85.2% on the Seattle dataset. Although SketchPolymer's per-flow query time remains under 15ms, this latency is a practical bottleneck. In modern data center networks, a single switch may handle thousands of active flows [32]. Querying quantiles across all flows introduces substantial delay and impacts time-sensitive tasks like anomaly detection. Additionally, Fig.9 details the breakdown of SketchPolymer's query time. When the memory budget is below 32KB, the control plane's computation dominates the query latency. When the memory budget exceeds 32KB, the register data read time increases significantly due to the larger volume of data retrieved from the data plane.

In conclusion, NetScope outperforms SketchPolymer in quantile estimation with an average relative decrease of 44% in ALE and an average reduction in query time of 87.4%. Additionally, the slight increase of only 0.84% in transmission time due to update operations does not impede the adoption of quantile estimation in anomaly detection.

D. Anomaly Detection & Root Cause Analysis Evaluation

1) Dataset: To evaluate NetScope's anomaly detection and fault localization, we replay the network traffic based on the UW datacenter trace dataset [32]. The packet size and inter-packet gap of the background flows conform to this trace dataset, and the ECMP strategy relies on path weight.

2) Fault Injection: The transmission speed of background flow is about 200 packets per second. Micro-bursts are generated by injecting one transient flow in a great amount, over 1000 pps within a second. The burst flow occupies the queue in switches quickly, leading to a transient high latency event (Fig. 10(a)). ECMP Load Imbalance is generated by setting a randomly picked switch's ECMP strategy from 1:1 to an imbalance ratio (random from 1:4 to 1:10). As a result, the throughput of two paths in an ECMP group varies (Fig. 10). As the number of packets forwarded to one ECMP path increases, the latency of that path increases, while the latency of the other path decreases. Process Rate Decrease scenarios randomly select a port of a randomly picked switch and decrease its packet process rate lower than 100 pps. Delay and Drop event are generated with Chaosblade [53] by injecting the anomaly to the switch's interface(s).

TABLE I
RECALL AND EXAM SCORE OF NETSCOPE (NS), SPIDERMON (SM), INTSIGHT (IS) AND SYNDB (SN). HERE ONLY SN IS AIDED BY EXPERT KNOWLEDGE TO DIAGNOSE

Anomaly Cause	$R@1$ (%)				$R@2$ (%)				$R@3$ (%)				$R@5$ (%)				Exam Score			
	NS	SM	IS	SN	**NS**	SM	IS	SN	**NS**	SM	IS	SN	**NS**	SM	IS	SN	**NS**	SM	IS	SN
Micro-burst	**75**	50	10	44	**85**	62	39	73	**92**	73	60	79	**96**	75	81	94	**0.3**	0.3	2.4	1.5
ECMP Load Imbalance	**88**	70	29	50	**100**	96	50	79	**100**	96	54	96	**100**	100	96	100	**0.1**	0.4	2.5	0.8
Process Rate Decrease	**94**	100	71	100	**100**	100	100	100	**100**	100	100	100	**100**	100	100	100	**0.1**	0	0.3	0
Delay	**73**	-	-	100	**83**	-	-	100	**87**	-	-	100	**93**	-	-	100	**0.4**	10	10	0
Drop Event	**67**	-	-	100	**94**	-	-	100	**100**	-	-	100	**100**	-	-	100	**0.4**	10	10	0
Overall	***83***	*44*	*21*	*79*	***95***	*52*	*32*	*90*	***97***	*54*	*40*	*95*	***99***	*55*	*55*	*99*	***0.3***	*4.1*	*5.0*	*0.5*

3) Evaluation Baselines and Metrics for Anomaly Detection: To demonstrate the effectiveness of the NetScope detection model, we compare it with a model based on 4 static thresholds (i.e., 100ms, 500ms, 1000ms, 5000ms) and 2 detection models, including our previous version (MARS [34]) and SketchPolymer based method (SKP). Although HistSketch supports per-flow quantile estimation, its current implementation performs updates and queries on the CPU, and does not support in-switch quantile computation. Thus, it is not suitable for evaluating in-network anomaly detection. Although MARS and SKP calculate adaptive thresholds at runtime by estimating latency median and standard deviation, they require assistance from the control plane. Specifically, the control plane of MARS retrieves latency data of sample packets to calculate the latency median and standard deviation, while the control plane of SKP pulls sketch data that store medians for different Flows and calculates the median for each Flow using varying hash operations. SKP uses a similar method as NetScope to estimate standard deviation.

For the evaluation, we use a commonly-used metrics **F1-score** [21], [54], [55] and show F1-score of each model in detecting different types of faults. F1-score is the harmonic mean of Precision and Recall and the higher F1-score means the better detection performance. Since the classical point-based Precision and Recall is inadequate to represent domain-specific performance for detection on time series data (e.g., latency in this study), a range-based strategy [56] is adopted to augment our evaluation. The strategy is more strict due to its consideration for more aspects of the predictions, including the existence, size, position and cardinality. More details can be found in the original work [56].

4) Evaluation Baselines and Metrics for Root Cause Analysis: To demonstrate the effectiveness of NetScope in localizing faults in multiple scenarios, we compare it with three up-to-date models, including SpiderMon [21], IntSight [18], and SyNDB [20], which are used to produce an ordered list of culprits. The result of SpiderMon is ordered by the degree in its Wait-For Graph (WFG). The result of IntSight and SyNDB is ordered according to the data from the conditional flow report and *p*-record, respectively. The result of SpiderMon is based on the level in its Wait-For Graph (WFG), while the ranking of IntSight and SyNDB is based on the query data from the flow report and *p*-record, respectively. The effectiveness of SyNDB may be overstated (represented by gray color in Table I) as it is query-based and requires expert knowledge to determine which telemetry data it should query.

To evaluate the effectiveness of NetScope, we introduce two metrics that are widely used in root cause localization. **Recall of *Top-k*** ($R@k$) reveals the probability that the root cause can be located within the top k culprits provided by the algorithm [57], [58]. A survey [59] conducted that over 73% developers only consider *Top-5* ranked elements. Thus, this paper splits the results into $R@k(k = 1, 2, 3, 5)$. **Exam Score** is a metric to measure the percentage of the false positives that need to be excluded manually by admin before locating the real root cause [60], [61], [62]. If the root cause is out of *Top-5*, we set a default 10 false positive cause before it. Note that the higher $R@k$ is better, while the lower Exam Score is better.

5) RQ5: Anomaly Detection Effectiveness: Fig. 12 presents the overall F1-Score of various anomaly detection models. The NetScope anomaly detection model achieves state-of-the-art overall results. Specifically, in terms of overall F1-score, compared with baselines based on static thresholds, models using adaptive thresholds attains a relative improvement of 47%∼116% on average. Furthermore, compared with other adaptive baselines, our NetScope anomaly detection model demonstrates a relative improvement of 32%∼42%.

The inferior performance of static threshold-based models is partly due to their instability in detecting various faults. Fig. 11 presents the experimental results in the form of violin charts to illustrate the performance of different detection models against various faults. Specifically, the higher F1-score median for appropriate static thresholds in Fig.11(a)-(c) indicates that these thresholds perform well in detecting faults (e.g., Delay, Micro-burst, Process Rate Decrease) that cause higher latency. However, their performance is unstable even for the same type of faults due to dynamic traffic workloads in the UW trace dataset. Meanwhile, in contrast to MARS and SKP, which rely on the control plane to obtain latency data for threshold calculation and to install thresholds in the data plane, NetScope detects faults solely during packet processing, without the assistance of control plane. This in-network detection makes NetScope more sensitive and rapid in fault detection. Therefore, NetScope attains better performance than other adaptive threshold-based models.

Additionally, the impact of hyper-parameter t on the detection accuracy is also analyzed. Fig. 14 shows the F1-score as t varies from 1 to 6. As it shows, the detection performance improves significantly when increasing t from 1 to 3, as smaller values produce higher false positive rates. The F1-score stabilizes around $t = 3$ with a slight increase at $t = 6$, indicating low sensitivity to minor variations. This confirms that $t = 3$ provides an optimal balance between precision and recall. The F1-score stability also demonstrates that NetScope is resilient to moderate threshold parameter variations.

6) RQ6: Root Cause Analysis Effectiveness: Table I presents the effectiveness of different analysis models. Overall from the comparison, NetScope achieves the best localization performance, with a relative improvement of 6%∼197% on $R@2$ and a relative decrement of 40%∼94% on Exam Score.

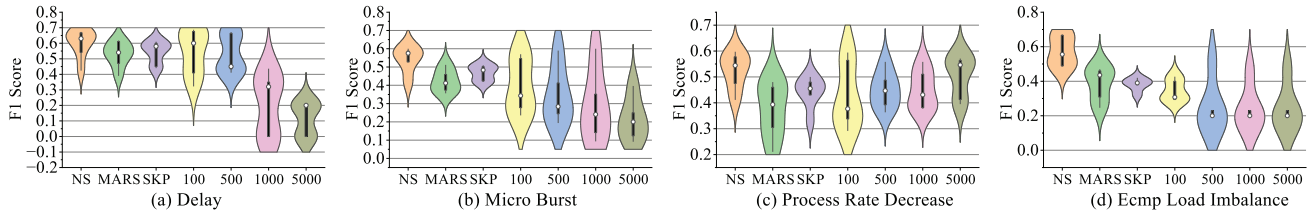


Fig. 11. Range-based F1-Score of anomaly detection models in detecting different faults (shown with violin charts).

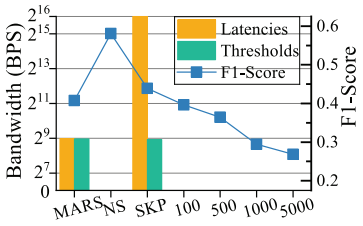


Fig. 12. Bandwidth usage and F1-Score of detection models.

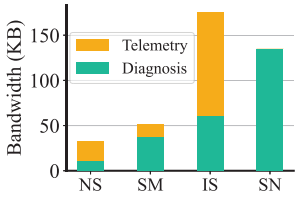


Fig. 13. Bandwidth usage of fault localization models.

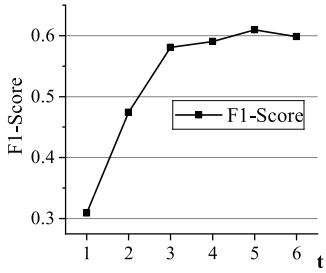


Fig. 14. F1-Score with different t value.

The inferior performance of SpiderMon and IntSight is partially due to their inability to localize delay and drop events while the inferior performance of SyNDB is in part caused by its untimely anomaly detection. Specifcally, SpiderMon focuses on the micro-burst flow that occupies most line-rate. In this case, most flows wait for the culprit flow, thus the degree of the culprit vertex is high, with a large indegree and a small outdegree. However, when a flow bursts in a great amount, most wait-for relation is between burst flow itself, i.e., the indegree and outdegree are similar and SpdierMon would fail to localize the root cause. As SpiderMon only carries a cumulative latency of queuing delta time, which is based on whether to send the "spider" notice packet, it cannot sense the anomaly outside the queue. Therefore, SpiderMon cannot detect the exceptions of delay and drop, thus failing to start a root cause analysis. Though IntSight can sense the drop event at flow-level by comparing the source count and destination, it fails to locate the drop event at a switch/port-level. Similar to SpiderMon, IntSight updates contention points according to queuing delta time. As a result, IntSight falls short to locate the deeper root cause. As IntSight is not good at aggregate reports into a ranked metric, its recall is relatively low until *Top-5*. Since SyNDB cannot decide which data should be queried and diagnose without expert knowledge, it has to iterate the diagnosis process for all kinds of failure causes to find the root cause. Therefore, we have to assume SyNDB knows the root cause at first to conduct the corresponding diagnosis process, rendering SyNDB with expert knowledge to outperform other approaches in many circumstances. Besides, SyNDB does not have a trigger rule for drop events except for updating a forwarding rule. For packet loss caused by other reasons, SyNDB cannot sense it timely. Moreover, to check whether a drop event happened in history, traversing the entire database is needed, which is time-consuming.

In conclusion, the results demonstrate the effectiveness of NetScope in anomaly detection and improves F1-score by 36%∼127%. Additionally, NetScope efficiently localizes root causes without expert experience, with 6%∼197% improvements on $R@2$ and a 40%∼94% reduction on Exam Score. While SyNDB also demonstrates strong performance across all scenarios due to its high coverage of network data, it incurs significant overhead, as discussed in Section VIII-E.1.

E. Overhead

1) RQ7: Network Bandwidth: Fig. 12–13 present the comparison of NetScope on the network bandwidth with different detection baselines and different localization baselines. The network bandwidth is utilized for communication between the control plane and the data plane.

Fig. 12 demonstrates the network bandwidth during anomaly detection. Orange bars ("Latencies") indicate the bandwidth usage when all telemetry data is sent to the control plane, while green bars ("Thresholds") represent the bandwidth usage for threshold transmission to the switch. Specifically, the control plane of MARS and SKP pull the raw latency data or the latency median data from the data plane of the switch to calculate adaptive thresholds at runtime, which are then transmitted to the switch. However, by computing adaptive thresholds within the switch without the need of the control plane, NetScope does not cause overhead on network bandwidth in anomaly detection.

Fig. 13 demonstrates the network bandwidth during fault localization. Orange bars ("Telemetry") refer to the additional bandwidth required for packet information, such as INT headers. Green bars ("Diagnosis") refer to the data sent from the data plane to the control plane, including telemetry data for root cause analysis. NetScope achieves the lowest total network bandwidth consumption, reducing it by 35.7%∼81.2% compared to other baselines. Specifically, SyNDB does not consume telemetry bandwidth as it does not require INT headers. However, it requires all switches to send recorded data to control plane, causing a significant amount of diagnosis bandwidth. IntSight requires a large INT header (33B) to perform anomaly detection and root cause analysis in switches,

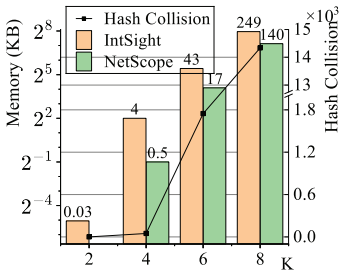


Fig. 15. Memory usage & Hash collisions in different scale.

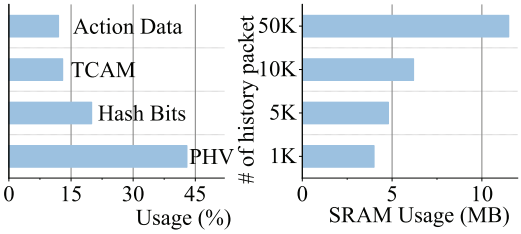


Fig. 16. The resource usage in Tofino.

consuming a significant amount of telemetry bandwidth. It sends data to the control plane conditionally, resulting in less diagnosis bandwidth compared to SyNDB. SpiderMon has much lower telemetry bandwidth compared to IntSight as its INT header only contains latency information. Unlike SyNDB and SpiderMon which collect data from all switches, NetScope only requires edge switches to send diagnosis data. In addition, NetScope collects less data per edge switch compared with IntSight. For example, the bit map of IntSight that indicates a specific switch in packet's transmission path is usually set at 48 bits per map. Thus, NetScope consumes less diagnosis bandwidth. However, NetScope requires more telemetry information in the INT header, leading to a slightly greater telemetry bandwidth than SpiderMon, but still much smaller than IntSight.

Overall, NetScope achieves the least total bandwidth and diagnosis bandwidth comsumption by a average reduction of 64.2% and 82.7%, respectively. Though the total bandwidth overhead of SpiderMon and NetScope is similar, NetScope collects information in a more comprehensive manner, providing a more dimensional and thorough analysis.

2) RQ8: Switch Resource Usage: Fig. 16 shows the usage of switch resource.[1] As most of the telemetry data temporarily accumulated on edge switches is stored in the Ring Table, which records recent per-packet telemetry metedata to support root cause analysis. Fig. 16 also illustrates how NetScope scales with the Ring Table size. Ring Table size indicates the number of history packet can be collected to the control plane on each switch once a time. The history data is saved in the SRAM (Static RAM) register of switch. NetScope fits in the Tofino pipeline comfortably for now, and can scale to record more data as switch memory size increases over time [63].

To evaluate the efficiency of the telemetry path data compression, we calculate the switch memory usage for PathID in IntSight and in NetScope. Specifically, the memory usage of IntSight is $M_{\text{IS}} = \sum_{p \in paths} \#hop_p \times size(\text{MAT}_{\text{IS}})$, and

[1]The PHV (Packet Header Vector) carries packet data throughout the Tofino pipeline. Hash Bits are used in hash generators, e.g., ECMP, and to calculate PathID. TCAM (Ternary Content Addressable Memory) is used for matching in MATs, such as the longest-prefix match. Action Data are the stage data for PHV ALUs (Arithmetic Logic Units).

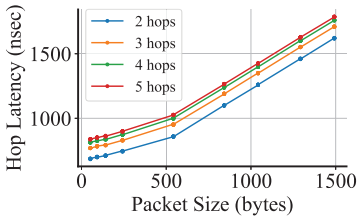


Fig. 17. Average hop latency under paths of different lengths.

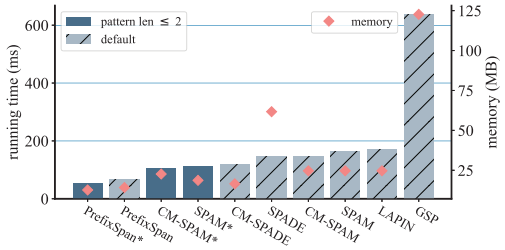


Fig. 18. Overhead of FSM.

NetScope requires $M_{\text{MS}} = \sum_{p \in paths} \sum_{h_p \in hops(p)} p_{hash} \times size(\text{MAT}_{\text{MS}})$, where $p_{hash} \in (0, 1)$ is the probability of hash conflict. Fig. 15 depicts the memory usage of both methods and the hash collisions count in NetScope under varing fat-tree sizes K. The results confirm that NetScope achieves lower memory consumption than IntSight across various network scales. For example, in a $K = 8$ fat-tree topology with 7392 paths between edge switches, IntSight allocates 36448 MAT entries (one per hop), consuming 249KB of memory (7 bytes per entry). In contrast, NetScope requires only 14336 entries (10 bytes per entry) and consumes 140KB to distinguish all paths, with CRC16/CRC32 as the hash algorithm, achieving a 43.8% memory reduction. Although hash collisions count increases with network size (e.g., 14K collisions at $K = 8$), it remains acceptable relative to the total hops size (over 36K). This evaluation confirms that NetScope's path-aware encoding outperforms IntSight in memory usage in large-scale networks.

3) RQ9: Latency Overhead: Fig. 17 presents the average hop latency on paths of various lengths (number of hops) with NetScope. As packet size increases, the latency per hop also increases, but remains below 2000 nanoseconds. The hop latency of the largest packet (MTU size, i.e., 1500 bytes) is higher than that of the smallest packet (without payload) by 121.7% on average. Therefore, the NetScope INT data collection impacts packet transmission latency, and this impact becomes more significant as the inserted telemetry data increases with the number of hops.

4) RQ10: FSM Algorithms Efficiency: Fig. 18 shows the comparison of different FSM algorithms in our fault scenarios, including PrefixSpan [23], Lapin [64], GSP [65], Spade [66], Spam [67], SM-Spade and SM-Spam [68]. Besides the $\min support$, Some algorithms can limit the maximum pattern length. Since NetScope only considers patterns whose length is less than two, the algorithms with maximum pattern length equals two perform relatively better. Most algorithms finish in 200 ms and consume less than 30 MB of memory. PrefixSpan performs the best among all algorithms.

In conclusion, NetScope consumes no network bandwidth in anomaly detection and reduces total bandwidth by 64.2% compared with other baselines. Moreover, NetScope uses affordable switch resources and the diagnosis method in NetScope's control plane only uses 30MB memory.

IX. Related Work

a) Quantile Sketch:

Quantile sketches have gained attention since they rapidly estimate data distributions in streaming datasets. Previous methods either treat the entire stream as a single flow without differentiating between flow keys [26], [27], [28], [29], or can only run as CPU programs and cannot be implemented in switches [69], [70]. For instance, DDSketch [26] partitions the value range into segments and tracks segments counts, but runs on the CPU and only supports single-flow analysis. QPipe [28] and EZQ [29], the first estimation methods fully implementable in switches, only supports single-flow estimation. Extending them to multi-flow estimation by duplicating data structures per flow incurs prohibitive memory overhead. SQUAD [69] tracks multi-flow quantiles using a compound data structure but requires unsupported operations (e.g., random number generation) in switches. M4 [70] proposes a framework that adapts single-flow sketches (like DDSketch) for per-flow estimation, but it is designed for CPU platforms and relies on complex data structures unsuitable for switches. Recent studies propose single data structures to estimate multi-flow quantiles. HistSketch [38] proposes a histogram-in-sketch design for tracking per-flow distributions. SketchPolymer [37] estimates quantiles by recording values with multiple Count-Min sketches and hash operations. SQUID [71] adopts sample-sketching method to estimate quantile. Although they are implementable in switches, they still require the control plane to either iterate complex queries to obtain results or estimate quantile from the sampled data. QuantileFilter [72] supports fully in-network anomaly detection without control-plane, but uses a manually-configured, static, shared threshold for all flows and does not maintain per-flow distributions. In contrast, NetScope enables adaptive per-flow quantile estimation with fully in-network updates and queries, enabling efficient in-network anomaly detection.

b) Network Failure Diagnosis:

Network failure diagnosis is a crucial for the performance of complex networks. **Out-of-band network diagnosis** either use packet mirroring at switches [5], [6], [7], or send probes into the network [35], [73]. NetSight [5] copies all packets to the control plane for further diagnosis, but incurs excessive overhead due to packets collection unrelated to the diagnosis. While sampling technique [5], [6], [7] reduce monitoring overheads, they often miss unexpected events. Pingmesh [35] and NetBouncer [73] use end-host agents to send probes, but since probe traffic may follow different paths than production traffic, they struggle with accurate diagnosis of transient failures. Another line of research have proposed **Programmable switch assisted diagnosis**. Since out-of-band network diagnosis methods introduce extra overhead, existing works leverage programmable switches for monitoring and failure localization. BurstRadar [19], ConQuest [36], *Flow [74], and PrintQueue [75] depict switch queue status but focus on specific failure types, whereas NetScope aims at a broader network failure. SwitchPointer [76] and OmniMon [77] combines the host computation and switch visibility for comprehensive monitoring. However, they indiscriminately monitor and collect telemetry data from switches. More recent studies address continous overhead. Newton [78] enables on-demand, intent-driven queries, while AutoSketch [79] automates query compilation into efficient sketches, though both assume operators know what anomalies to target. SpiderMon [21] implements on-demand collection triggered by anomaly detection, but ignores varying link utilization across network layers where a subset of the core links often experience high utilization [32]. It is challenging to minimize the usage of core switches due to the monitoring and data collection. NetScope address this by collecting the telemetry data from the edge switches for fault localization. While SpiderMon, IntSight [18] and Marple [80] rely on preset thresholds to detect anomaly that limit accuracy in versatile traffic environments, NetScope relies on dynamic thresholds for anomaly detection. Query-based diagnosis systems [1], [20], [81], [82] require the operators to understand the network and the potential locations of the failures in advance, hindering the timely and convenient diagnosis in a large-scale network. In contrast, NetScope automatically detects and localizes failures without any static queries.

X. Conclusion

In this paper, we introduce NetScope, a system designed for low-cost anomaly detection and precise root cause localization in programmable networks. To achieve low-cost monitoring, NetScope uses INT and an on-demand strategy for pulling data from the control plane upon the detection of an anomaly. To perform self-adaptive anomaly detection, NetScope proposes a per-Flow quantile sketch to estimate per-Flow latency median entirely in the data plane, which is further used to dynamically calculate thresholds accordingly. To localize the root cause, NetScope utilize FSM and SBFL to output an ordered list of culprits. Our evaluation of NetScope on a Tofino testbed and a simulated Mininet environment demonstrates that, despite a low overhead in network bandwidth and switch memory, it can achieve more accurate anomaly detection with an improvement of 32%∼116% in accuracy and accurate root cause localization (0.95 R@2) for diverse network faults.

Appendix
Proof in Section VI-A.3

Proof: Let Q_ϕ denote the true ϕ-quantile of flow i's latency values, and $\hat{Q}_\phi$ represent the estimated quantile. Let r be the true rank of $\hat{Q}_\phi$ (i.e., the number of packets with latency less than $\hat{Q}_\phi$), and $\tilde{r}$ be the estimated rank based on the Count-min Sketch.

Since the rank $\tilde{r}$ is estimated using a Count-Min sketch and it is obtained from the sketch buckets (specifically, from the field c in the bucket $\mathcal{B}_d^{(2)}$ that counts the number of packets with latency less than $\hat{Q}_\phi$). Assume that ϵ is the error bound of Count-Min Sketch. Due to the property of the Count-Min sketch error, $\tilde{r}$ is an overestimation of the true rank r, with additive error at most ϵn_i and failure probability at most δ. Thus the rank estimate satisfies:

$$r \in [\tilde{r} - \epsilon n_i, \tilde{r}] \quad \text{with probability at least } 1 - \delta.$$

According to Algorithm 3, our method updates $\hat{Q}_\phi$ only if the latency of observed number of packets smaller or greater than $\hat{Q}_\phi$ exceeds the expected count ($n_i \times \phi$ or $n_i \times (1-\phi)$, respectively). It means that the quantile estimate $\hat{Q}_\phi$ is updated only when the observed rank estimate $\tilde{r}$ deviates from the target rank ϕn_i by more than ϵn_i. This design is supported that if the deviation is smaller than or equal to ϵn_i, it may be caused by sketch overestimation and the estimated quantile does not

need to be updated. Therefore, updates are conservatively delayed until:

$$|\tilde{r} - \phi n_i| > \epsilon n_i$$

We now analyze the deviation bound between the true rank r could be from the target ϕn_i.

- If the estimated rank $\tilde{r}$ is smaller than $\phi n_i - \epsilon n_i$, and since $r \geq \tilde{r} - \epsilon n_i$, we get get $r \geq \phi n_i - 2\epsilon n_i$. Moreover, since $r < \tilde{r} < \phi n_i - \epsilon n_i$, we can also get the upper bound of r. Therefore, we find that r lies within the interval $(\phi n_i - 2\epsilon n_i, \phi n_i - \epsilon n_i)$ and hence:
$$|r - \phi n_i| < 2\epsilon n_i.$$
- If the estimated rank $\tilde{r}$ is larger than $\phi n_i + \epsilon n_i$, a symmetric argument applies. The sketch guarantees $r \leq \tilde{r}$ and $r \geq \tilde{r} - \epsilon n_i$, so r lies between $(0, \phi n_i + \epsilon n_i)$, leading to:
$$|r - \phi n_i| < \epsilon n_i < 2\epsilon n_i.$$

Therefore, in either case, we can conservatively bound the deviation of the true rank from the desired rank by $2\epsilon n_i$, which can be presented as:

$$|r - \phi n_i| < 2\epsilon n_i \quad \text{(with probability at least } 1 - \delta).$$

Finally, we analyze how this rank error translates to an error in the estimation quantile value. Since $\hat{Q}_p hi$ is adjusted in discrete steps, and the step β_i is dynamically estimated as the upper-bound of the latency interval per packet in flow i, we can assert that unit rank deviation contributes at most $beta_i$ to the quantile error. Since the true rank r of $\hat{Q}_p hi$ deviates from the target rank ϕn_i by at most $2\epsilon n_i$, the quantile value estimation error satisfies:

$$|\hat{Q}_\phi - Q_\phi| \leq 2\epsilon n_i \cdot \beta_i.$$

To account for the increased estimation sensitivity near the distribution tails, where the quantile function changes more rapidly, we normalize the minimum of ϕ and $1 - \phi$. This gives us the final error bound of quantile estimation:

$$\mathbb{P}\left(|\hat{Q}_\phi - Q_\phi| \leq 2\beta_i \cdot \left\lceil \frac{\epsilon n_i}{\min(\phi, 1-\phi)} \right\rceil\right) \geq 1 - \delta$$

Hence Equation 1 holds.□

REFERENCES

[1] Y. Zhou et al., "Flow event telemetry on programmable data plane," in *Proc. Annu. Conf. ACM Special Interest Group Data Commun. Appl., Technol., Archit., Protocols Comput. Commun.* New York, NY, USA: ACM, Jul. 2020, pp. 76–89.

[2] E. Zhai et al., "Check before you change: Preventing correlated failures in service updates," in *Proc. 17th USENIX Symp. Netw. Syst. Design Implement. (NSDI)*. Santa Clara, CA, USA: USENIX, 2020, pp. 575–589.

[3] P. Gill et al., "Understanding network failures in data centers: Measurement, analysis, and implications," *SIGCOMM Comput. Commun. Rev.*, vol. 41, no. 4, pp. 350–361, Aug. 2011.

[4] X. Wu et al., "NetPilot: Automating datacenter network failure mitigation," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 42, no. 4, pp. 419–430, Aug. 2012.

[5] N. Handigol, B. Heller, V. Jeyakumar, D. Mazières, and N. McKeown, "I know what your packet did last hop: Using packet histories to troubleshoot networks," in *Proc. 11th USENIX Symp. Networked Syst. Design Implement. (NSDI 14)*, Apr. 2014, pp. 71–85.

[6] J. Rasley et al., "Planck: Millisecond-scale monitoring and control for commodity networks," in *Proc. ACM Conf. SIGCOMM*, Aug. 2014, pp. 407–418.

[7] O. Tilmans, T. Bühler, I. Poese, S. Vissicchio, and L. Vanbever, "Stroboscope: Declarative network monitoring on a budget," in *Proc. 15th USENIX Symp. Netw. Syst. Design Implement. (NSDI)*, May 2018, pp. 467–482.

[8] H. Chen et al., "Graph neural network based robust anomaly detection at service level in SDN driven microservice system," *Comput. Netw.*, vol. 239, Feb. 2024, Art. no. 110135.

[9] K. Agarwal, E. Rozner, C. Dixon, and J. Carter, "SDN traceroute: Tracing SDN forwarding without changing network behavior," in *Proc. 3rd Workshop Hot Topics Softw. Defined Netw.*, New York, NY, USA, Aug. 2014, pp. 145–150.

[10] M. J. Zekauskas et al., *A One-way Active Measurement Protocol (OWAMP)*, document RFC 4656, 2006.

[11] J. Babiarz et al., *A Two-Way Active Measurement Protocol (TWAMP)*, document RFC 5357, 2008.

[12] C. Fang et al., "VTrace: Automatic diagnostic system for persistent packet loss in cloud-scale overlay network," in *Proc. Annu. Conf. ACM Special Interest Group Data Commun. Appl., Technol., Archit., Protocols Comput. Commun.*, Jul. 2020, pp. 31–43.

[13] C. Jia et al., "Rapid detection and localization of gray failures in data centers via in-band network telemetry," in *Proc. IEEE/IFIP Netw. Oper. Manag. Symp.*, Budapest, Hungary, Apr. 2020, pp. 1–9.

[14] Z. Liu, J. Bi, Y. Zhou, Y. Wang, and Y. Lin, "NetVision: Towards network telemetry as a service," in *Proc. IEEE 26th Int. Conf. Netw. Protocols (ICNP)*, Sep. 2018, pp. 247–248.

[15] S. Tang, D. Li, B. Niu, J. Peng, and Z. Zhu, "Sel-INT: A runtime-programmable selective in-band network telemetry system," *IEEE Trans. Netw. Service Manage.*, vol. 17, no. 2, pp. 708–721, Jun. 2020.

[16] Y. Zhu et al., "Packet-level telemetry in large datacenter networks," in *Proc. ACM Conf. Special Interest Group Data Commun.*, Aug. 2015, pp. 479–491.

[17] R. Ben Basat, S. Ramanathan, Y. Li, G. Antichi, M. Yu, and M. Mitzenmacher, "PINT: Probabilistic in-band network telemetry," in *Proc. Annu. Conf. ACM Special Interest Group Data Commun. Appl., Technol., Archit., Protocols Comput. Commun.*, New York, NY, USA, Jul. 2020, pp. 662–680.

[18] J. Marques, K. Levchenko, and L. Gaspary, "IntSight: Diagnosing SLO violations with in-band network telemetry," in *Proc. 16th Int. Conf. Emerg. Netw. Exp. Technol.*, Nov. 2020, pp. 421–434.

[19] R. Joshi, T. Qu, M. C. Chan, B. Leong, and B. T. Loo, "BurstRadar: Practical real-time microburst monitoring for datacenter networks," in *Proc. 9th Asia Pacific Workshop Syst.*, New York, NY, USA, Aug. 2018, pp. 1–8.

[20] P. G. Kannan, N. Budhdev, R. Joshi, and M. C. Chan, "Debugging transient faults in data centers using synchronized network-wide packet histories," in *Proc. 18th USENIX Symp. Netw. Syst. Design Implement. (NSDI)*, Jun. 2021, pp. 253–268.

[21] W. Wang et al., "Closed-loop network performance monitoring and diagnosis with SpiderMon," in *Proc. 19th USENIX Symp. Networked Syst. Design Implement. (NSDI)*, Jun. 2022, pp. 267–285.

[22] P. Bosshart et al., "P4: Programming protocol-independent packet processors," in *Proc. ACM SIGCOMM Comput. Commun. Rev.*, vol. 44, May 2014, pp. 87–95.

[23] J. Pei et al., "PrefixSpan: Mining sequential patterns efficiently by prefix-projected pattern growth," in *Proc. 17th Int. Conf. Data Eng.*, Heidelberg, Germany, May 2001, pp. 215–224.

[24] A. Arrieta, S. Segura, U. Markiegi, G. Sagardui, and L. Etxeberria, "Spectrum-based fault localization in software product lines," *Inf. Softw. Technol.*, vol. 100, pp. 18–31, Aug. 2018.

[25] N. McKeown et al., "OpenFlow: Enabling innovation in campus networks," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 38, no. 2, pp. 69–74, Mar. 2008.

[26] C. Masson, J. E. Rim, and H. K. Lee, "DDSketch: A fast and fully-mergeable quantile sketch with relative-error guarantees," *Proc. VLDB Endowment*, vol. 12, no. 12, pp. 2195–2205, Aug. 2019.

[27] F. Zhao, S. Maiyya, R. Wiener, D. Agrawal, and A. E. Abbadi, "KLL± approximate quantile sketches over dynamic datasets," *Proc. VLDB Endowment*, vol. 14, no. 7, pp. 1215–1227, Mar. 2021.

[28] N. Ivkin, Z. Yu, V. Braverman, and X. Jin, "QPipe: Quantiles sketch fully in the data plane," in *Proc. 15th Int. Conf. Emerg. Netw. Exp. Technol.* New York, NY, USA: ACM, Dec. 2019, pp. 285–291.

[29] B. Wang, R. Chen, and L. Tang, "EasyQuantile: Efficient quantile tracking in the data plane," in *Proc. 7th Asia Pacific Workshop Netw.* New York, NY, USA: ACM, Jun. 2023, pp. 123–129.

[30] P4lang. (2020). *In-band Network Telemetry (int) Dataplane Specification*. Accessed: Feb. 12, 2025. [Online]. Available: https://p4.org/p4-spec/docs/INT_v2_1.pdf

[31] T. Benson, A. Akella, and D. A. Maltz. (2010). *Network Traffic Characteristics of Data Centers in the Wild (dataset)*. Accessed: Feb. 3, 2022. [Online]. Available: https://pages.cs.wisc.edu/

[32] T. Benson, A. Akella, and D. A. Maltz, "Network traffic characteristics of data centers in the wild," in *Proc. 10th Annu. Conf. Internet Meas. (IMC)*, Nov. 2010, p. 267.

[33] A. Roy, H. Zeng, J. Bagga, G. Porter, and A. C. Snoeren, "Inside the social network's (datacenter) network," in *Proc. ACM Conf. Special Interest Group Data Commun.*, Aug. 2015, pp. 123–137.

[34] B. Wang, H. Chen, P. Chen, Z. He, and G. Yu, "MARS: Fault localization in programmable networking systems with low-cost in-band network telemetry," in *Proc. 52nd Int. Conf. Parallel Process.* New York, NY, USA: ACM, Aug. 2023, pp. 347–357.

[35] C. Guo et al., "Pingmesh: A large-scale system for data center network latency measurement and analysis," in *Proc. ACM Conf. Special Interest Group Data Commun.*, Aug. 2015, vol. 45, no. 4, pp. 139–152.

[36] X. Chen et al., "Fine-grained queue measurement in the data plane," in *Proc. 15th Int. Conf. Emerg. Netw. Exp. Technol.*, New York, NY, USA, Dec. 2019, pp. 15–29.

[37] J. Guo, Y. Hong, Y. Wu, Y. Liu, T. Yang, and B. Cui, "SketchPolymer: Estimate per-item tail quantile using one sketch," in *Proc. 29th ACM SIGKDD Conf. Knowl. Discovery Data Mining*. New York, NY, USA: ACM, Aug. 2023, pp. 590–601.

[38] J. He, J. Zhu, and Q. Huang, "HistSketch: A compact data structure for accurate per-key distribution monitoring," in *Proc. IEEE 39th Int. Conf. Data Eng. (ICDE)*, Apr. 2023, pp. 2008–2021.

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

[40] V. Chandola et al., "Anomaly detection: A survey," *ACM Comput. Surv.*, vol. 41, no. 3, pp. 1–58, Jul. 2009.

[41] R. Chalapathy, N. L. D. Khoa, and S. Chawla, "Robust deep learning methods for anomaly detection," in *Proc. 26th ACM SIGKDD Int. Conf. Knowl. Discovery Data Mining*. New York, NY, USA: ACM, Aug. 2020, pp. 3507–3508.

[42] S. Gao, M. Handley, and S. Vissicchio, "Stats 101 in p4: Towards in-switch anomaly detection," in *Proc. 20th ACM Workshop Hot Topics Netw.* New York, NY, USA: ACM, Nov. 2021, pp. 84–90.

[43] Y. Li, R. Miao, M. Alizadeh, and M. Yu, "DETER: Deterministic TCP replay for performance diagnosis," in *Proc. 16th USENIX Symp. Networked Syst. Design Implement. (NSDI)*, Jan. 2019, pp. 437–452.

[44] J. A. Morris and M. J. Gardner, "Statistics in medicine: Calculating confidence intervals for relative risks (odds ratios) and standardised ratios and rates," *BMJ*, vol. 296, no. 6632, pp. 1313–1316, May 1988.

[45] B. Nathan, "Solving the mystery of link imbalance: A metastable failure state at scale," Meta, USA, Nov. 2014.

[46] (2022). *BMV2*. Accessed: Sep. 13, 2022. [Online]. Available: http://bmv2.org/

[47] Barefoot. (2023). *Open Tofino*. Accessed: Feb. 3, 2022. [Online]. Available: https://github.com/barefootnetworks/Open-Tofino

[48] *Open Network Linux*, O. C. Project, USA, 2023.

[49] P. Fournier-Viger et al., "SPMF: A Java open-source pattern mining library," *J. Mach. Learn. Res.*, vol. 15, no. 1, pp. 3389–3393, 2014.

[50] J. Cappos et al., "Seattle: A platform for educational cloud computing," in *Proc. 40th ACM Tech. Symp. Comput. Sci. Educ.* New York, NY, USA: ACM, 2009, pp. 111–115.

[51] A. S. Asrese, S. J. Eravuchira, V. Bajpai, P. Sarolahti, and J. Ott, "Measuring web latency and rendering performance: Method, tools, and longitudinal dataset," *IEEE Trans. Netw. Service Manage.*, vol. 16, no. 2, pp. 535–549, Jun. 2019.

[52] N. K. Sharma et al., "Evaluating the power of flexible packet processing for network resource allocation," in *Proc. 14th USENIX Symp. Networked Syst. Design Implement. (NSDI)*, Boston, MA, USA, Mar. 2017, pp. 67–82.

[53] (2021). *Chaosblade*. Accessed: Dec. 3, 2021. [Online]. Available: https://github.com/chaosblade-io/chaosblade

[54] X. Li, P. Chen, L. Jing, Z. He, and G. Yu, "SwissLog: Robust anomaly detection and localization for interleaved unstructured logs," *IEEE Trans. Dependable Secure Comput.*, vol. 20, no. 4, pp. 2762–2780, Apr. 2023.

[55] Z. He et al., "A spatiotemporal deep learning approach for unsupervised anomaly detection in cloud systems," *IEEE Trans. Neural Netw. Learn. Syst.*, vol. 34, no. 4, pp. 1705–1719, Apr. 2023.

[56] N. Tatbul, T. J. Lee, S. Zdonik, M. Alam, and J. Gottschlich, "Precision and recall for time series," in *Proc. Adv. Neural Inf. Process. Syst.*, S. Bengio, Ed. Red Hook, NY, USA: Curran Associates, Jan. 2018, pp. 1920–1930.

[57] J. Lin et al., "Microscope: Pinpoint performance issues with causal graphs in micro-service environments," in *Service-Oriented Computing*. Cham, Switzerland: Springer, 2018, pp. 3–20.

[58] X. Zhou et al., "Latent error prediction and fault localization for microservice applications by learning from system trace logs," in *Proc. 27th ACM Joint Meeting Eur. Softw. Eng. Conf. Symp. Found. Softw. Eng.*, New York, NY, USA, Aug. 2019, pp. 683–694.

[59] P. S. Kochhar et al., "Practitioners' expectations on automated fault localization," in *Proc. 25th Int. Symp. Softw. Test. Anal.*, New York, NY, USA, Jul. 2016, pp. 165–176.

[60] J. Jiang, R. Wang, Y. Xiong, X. Chen, and L. Zhang, "Combining spectrum-based fault localization and statistical debugging: An empirical study," in *Proc. 34th IEEE/ACM Int. Conf. Automated Softw. Eng. (ASE)*, Nov. 2019, pp. 502–514.

[61] S. Pearson et al., "Evaluating and improving fault localization," in *Proc. IEEE/ACM 39th Int. Conf. Softw. Eng. (ICSE)*, May 2017, pp. 609–620.

[62] G. Yu et al., "MicroRank: End-to-end latency issue localization with extended spectrum analysis in microservice environments," in *Proc. Web Conf.*, Apr. 2021, pp. 3087–3098.

[63] 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 *Proc. Conf. ACM Special Interest Group Data Commun.*, Aug. 2017, pp. 15–28.

[64] Z. Yang and M. Kitsuregawa, "LAPIN-SPAM: An improved algorithm for mining sequential pattern," in *Proc. 21st Int. Conf. Data Eng. Workshops (ICDEW)*, Jun. 2005, pp. 12–22.

[65] R. Srikant and R. Agrawal, "Mining sequential patterns: Generalizations and performance improvements," in *Advances in Database Technology EDBT*. Heidelberg, Germany: Springer, 1996, pp. 1–17.

[66] M. J. Zaki, "SPADE: An efficient algorithm for mining frequent sequences," *Mach. Learn.*, vol. 42, nos. 1–2, pp. 31–60, Jan. 2001.

[67] J. Ayres, J. Flannick, J. Gehrke, and T. Yiu, "Sequential pattern mining using a bitmap representation," in *Proc. 8th ACM SIGKDD Int. Conf. Knowl. Discovery Data Mining*, New York, NY, USA, Jul. 2002, pp. 429–435.

[68] P. Fournier-Viger et al., "Fast vertical mining of sequential patterns using co-occurrence information," in *Advances in Knowledge Discovery and Data Mining*. Cham, Switzerland: Springer, 2014, pp. 40–52.

[69] R. Shahout, R. Friedman, and R. B. Basat, "Together is better: Heavy hitters quantile estimation," *Proc. ACM Manag. Data*, vol. 1, no. 1, pp. 1–25, May 2023.

[70] S. Dong et al., "M4: A framework for per-flow quantile estimation," in *Proc. IEEE 40th Int. Conf. Data Eng. (ICDE)*, May 2024, pp. 4787–4800.

[71] R. Ben Basat, G. Einziger, W. Han, and B. Tayh, "SQUID: Faster analytics via sampled quantile estimation," *Proc. ACM Netw.*, vol. 2, no. 3, pp. 1–23, Aug. 2024.

[72] Y. Wu et al., "Online detection of outstanding quantiles with QuantileFilter," in *Proc. IEEE 40th Int. Conf. Data Eng. (ICDE)*, May 2024, pp. 831–844.

[73] C. Tan et al., "NetBouncer: Active device and link failure localization in data center networks," in *Proc. USENIX Symp. Netw. Syst. Design Implement. (NSDI)*, Boston, MA, USA, Aug. 2019, pp. 599–614.

[74] J. Sonchack, O. Michel, A. J. Aviv, E. Keller, and J. M. Smith, "Scaling hardware accelerated network monitoring to concurrent and dynamic queries with flow," in *Proc. USENIX Annu. Tech. Conf. (USENIX ATC)*, Jan. 2018, pp. 823–835.

[75] Y. Lei, L. Yu, V. Liu, and M. Xu, "PrintQueue: Performance diagnosis via queue measurement in the data plane," in *Proc. ACM SIGCOMM Conf.*, New York, NY, USA, Aug. 2022, pp. 516–529.

[76] P. Tammana, R. Agarwal, and M. Lee, "Distributed network monitoring and debugging with SwitchPointer," in *Proc. USENIX Symp. Networked Syst. Design Implement.*, Jul. 2018, pp. 453–456.

[77] Q. Huang, H. Sun, P. P. C. Lee, W. Bai, F. Zhu, and Y. Bao, "OmniMon: Re-architecting network telemetry with resource efficiency and full accuracy," in *Proc. Annu. Conf. ACM Special Interest Group Data Commun. Appl., Technol., Archit., Protocols Comput. Commun.*, New York, NY, USA, Jul. 2020, pp. 404–421.

[78] Y. Zhou et al., "Newton: Intent-driven network traffic monitoring," in *Proc. 16th Int. Conf. Emerg. Netw. Exp. Technol.*, Nov. 2020, pp. 295–308.

[79] H. Sun et al., "AutoSketch: Automatic sketch-oriented compiler for query-driven network telemetry," in *Proc. 21st USENIX Symp. Networked Syst. Design Implement. (NSDI)*, Santa Clara, CA, USA, Apr. 2024, pp. 1551–1572.

[80] S. Narayana et al., "Language-directed hardware design for network performance monitoring," in *Proc. Conf. ACM Special Interest Group Data Commun.*, Aug. 2017, pp. 85–98.
[81] A. Gupta, R. Birkner, M. Canini, N. Feamster, C. Mac-Stoker, and W. Willinger, "Network monitoring as a streaming analytics problem," in *Proc. 15th ACM Workshop Hot Topics Netw.*, New York, NY, USA, Jul. 2016, pp. 106–112.
[82] R. Teixeira et al., "Packetscope: Monitoring the packet lifecycle inside a switch," in *Proc. Symp. SDN Res.* New York, NY, USA: ACM, 2020, pp. 76–82.

Hongyang Chen received the Ph.D. degree in computer science from Sun Yat-sen University, China, in 2025. He is currently a Research Engineer at Huawei Technologies Company Ltd. He has published more than ten papers in international conferences, such as ICSE, ASE, and ICPP, and in top journals, including IEEE/ACM TRANSACTIONS ON NETWORKING (ToN), IEEE TRANSACTIONS ON DEPENDABLE AND SECURE COMPUTING (TDSC), and *Computer Networks*. His research interests include distributed systems, cloud-native systems, and programmable networks.

Benran Wang received the bachelor's degree from Sun Yat-sen University in 2021 and the master's degree in 2024. His current research areas focus on software defined networking, especially programmable switches with P4.

Guangba Yu received the M.Eng. and Ph.D. degrees from Sun Yat-sen University. He is a Post-Doctoral Researcher at The Chinese University of Hong Kong (CUHK). He has published more than 40 refereed papers in premier journals and conferences, including ICSE, FSE, ASPLOS, WWW, IEEE TRANSACTIONS ON SOFTWARE ENGINEERING, and IEEE TRANSACTIONS ON SERVICES COMPUTING. His research interests focus on system reliability in cloud-native and AI systems, with a particular emphasis on LLM4Ops and Ops4LLM. He has served as a program committee member for several international conferences and a reviewer for journals, such as IEEE TRANSACTIONS ON SOFTWARE ENGINEERING, IEEE TRANSACTIONS ON COMPUTERS, and IEEE TRANSACTIONS ON SERVICES COMPUTING.

Zilong He received the Ph.D. degree from Sun Yat-sen University, China, in 2025. His current research areas include software reliability, cloud computing, and AI driven operations. He was a recipient of several awards, including the Best Paper Award at ISSRE 2022, the ACM SIGSOFT Distinguished Paper Award at FSE 2024, and one ESI highly cited paper.

Pengfei Chen received the Ph.D. degree from the Department of Computer Science, Xi'an Jiaotong University, in 2016. He is currently a Professor with the School of Computer Science and Engineering, Sun Yat-sen University. Meanwhile, he is a Ph.D. Advisor. He has published more than 80 papers in some international conferences, including ACM ASE/ICSE/FSE, IEEE INFOCOM, WWW, ACM/IEEE CCGRID, IEEE ISSRE, IEEE ICWS, IEEE DSN, and ICPP and journals including, IEEE TRANSACTIONS ON DEPENDABLE AND SECURE COMPUTING, IEEE TRANSACTIONS ON PARALLEL AND DISTRIBUTED SYSTEMS, IEEE TRANSACTIONS ON NEURAL NETWORKS AND LEARNING SYSTEMS, IEEE TRANSACTIONS ON SERVICES COMPUTING, IEEE TRANSACTIONS ON EMERGING TOPICS IN COMPUTING, and IEEE TRANSACTIONS ON CLOUD COMPUTING. He is currently interested in distributed systems, AIOps, cloud computing, microservice, and network systems. Especially, he has strong skills in cloud computing. He serves as a program committee member for multiple conferences and a reviewer for some internal journals, such as IEEE TRANSACTIONS ON DEPENDABLE AND SECURE COMPUTING, IEEE TRANSACTIONS ON COMPUTERS, and *Information Science*.

Chen Sun received the Ph.D. degree from the Department of Computer Science and Technology in 2019. He is currently a Research Scientist at Huawei Technologies Company Ltd. He has published more than 30 papers in international conferences, including SIGCOMM, NSDI, ATC, and MM and top journals, including IEEE/ACM TRANSACTIONS ON NETWORKING, IEEE JOURNAL ON SELECTED AREAS IN COMMUNICATIONS, and IEEE TRANSACTIONS ON PARALLEL AND DISTRIBUTED SYSTEMS. His research interest lies in network measurement, network reliability, and data center network. He serves as a reviewer for multiple conferences and journals, including CoNEXT and IEEE/ACM TRANSACTIONS ON NETWORKING.

Zibin Zheng (Fellow, IEEE) received the Ph.D. degree in computer science and engineering from Chinese University of Hong Kong. He is currently a Professor and the Deputy Dean of the School of Software Engineering, Sun Yat-sen University, Guangzhou, China. He authored or co-authored more than 200 international journal and conference papers, including one ESI hot paper and six ESI highly cited papers. His research interests include blockchain, software engineering, and services computing. He is a fellow of IET. He was a recipient of several awards, including the Top 50 Influential Papers in Blockchain of 2018, the ACM SIGSOFT Distinguished Paper Award at ICSE2010, and the Best Student Paper Award at ICWS2010.