

NeuFSD: Rethinking Flow Size Distribution via Neural Decoding

QILONG SHI*, Tsinghua University, China

PINZE REN*, Tsinghua University, China

YUFEI LI*, Tsinghua University, China

DAYU WANG, Peking University, China

YAOMING LI, Peking University, China

YUANYUAN ZHANG, Zhongguancun Laboratory, China

TONG YANG, Peking University, China

YANGYANG WANG, Tsinghua University, China

MINGWEI XU, Tsinghua University, China

Estimating the Flow Size Distribution (FSD) is crucial for network management, security, and capacity planning. Existing approaches face a difficult trade-off: sampling has a simple online path but loses accuracy under tight memory, while sketch-based methods such as MRAC and its descendants rely on iterative Expectation-Maximization (EM)-style decoding that is too slow for real-time measurement. We introduce NeuFSD, a neural decoding framework that replaces iterative statistical reconstruction with a learned decoder. NeuFSD keeps the data plane compact with a Hot Filter and a counter array, reshapes the counter state into 2-D views, and uses a Vision Transformer decoder to infer the FSD in one forward pass. For online deployment, NeuFSD combines offline pre-training, low-rate sampling, and continuous supervised fine-tuning, enabling adaptation without exact per-window FSD labels. A lightweight residual-mass calibration further reduces burst errors by enforcing an online packet-mass constraint. We evaluate NeuFSD on four large real-world traces against MRAC, Elastic Sketch, DaVinci Sketch, and sampling baselines. At a fixed 16KB counter-array budget, NeuFSD achieves lower WMRD and MRD while reducing decoding from seconds-scale EM iterations to millisecond-scale neural inference on both CPU and GPU. NeuFSD is also deployable across hardware and software data planes: it uses only a few Tofino pipeline stages, and Redis/BESS deployments maintain practical online throughput. These results show that NeuFSD is an accurate, adaptive, and deployable solution for real-time FSD measurement.

CCS Concepts: • **Networks** → **Network measurement**; • **Computing methodologies** → *Neural networks*.

Additional Key Words and Phrases: Sketch, flow size distribution, network measurement, data stream

ACM Reference Format:

Qilong Shi, Pinze Ren, Yufei Li, Dayu Wang, Yaoming Li, Yuanyuan Zhang, Tong Yang, Yangyang Wang, and Mingwei Xu. 2026. NeuFSD: Rethinking Flow Size Distribution via Neural Decoding. *Proc. ACM Meas. Anal. Comput. Syst.* 10, 3, Article 72 (December 2026), 46 pages. <https://doi.org/10.1145/3857375>

*These authors contributed equally to this work.

Authors' Contact Information: [Qilong Shi](mailto:sql23@mails.tsinghua.edu.cn), Tsinghua University, Beijing, China, sql23@mails.tsinghua.edu.cn; [Pinze Ren](mailto:rpz21@mails.tsinghua.edu.cn), Tsinghua University, Beijing, China, rpz21@mails.tsinghua.edu.cn; [Yufei Li](mailto:yfli25@mails.tsinghua.edu.cn), Tsinghua University, Beijing, China, yfli25@mails.tsinghua.edu.cn; [Dayu Wang](mailto:victorwang030401@stu.pku.edu.cn), Peking University, Beijing, China, victorwang030401@stu.pku.edu.cn; [Yaoming Li](mailto:yibachenggong@gmail.com), Peking University, Beijing, China, yibachenggong@gmail.com; [Yuanyuan Zhang](mailto:zhangyy@mail.zgclab.edu.cn), Zhongguancun Laboratory, Beijing, China, zhangyy@mail.zgclab.edu.cn; [Tong Yang](mailto:yangtongemail@gmail.com), Peking University, Beijing, China, yangtongemail@gmail.com; [Yangyang Wang](mailto:wangyy-13@tsinghua.edu.cn), Tsinghua University, Beijing, China, wangyy-13@tsinghua.edu.cn; [Mingwei Xu](mailto:xumw@tsinghua.edu.cn), Tsinghua University, Beijing, China, xumw@tsinghua.edu.cn.



This work is licensed under a Creative Commons Attribution 4.0 International License.

© 2026 Copyright held by the owner/author(s).

ACM 2476-1249/2026/12-ART72

<https://doi.org/10.1145/3857375>

1 Introduction

Estimating the **Flow Size Distribution (FSD)** is a core primitive for network visibility: for each flow size i , the monitor estimates n_i , the number of flows containing exactly i packets. FSD supports anomaly detection, load balancing, QoS, accounting, and capacity planning [38, 46, 53, 54]. In high-speed networks, however, FSD estimation must fit within a small data-plane memory budget while producing results quickly enough for online decisions. These requirements become increasingly difficult to satisfy as link rates grow and monitoring windows become shorter, leaving little room for either large per-flow state or expensive control-plane computation.

FSD estimation belongs to the broader class of streaming and network telemetry tasks that approximate frequency distributions under strict memory constraints. Classical frequency summaries, counter sketches, and distribution synopses establish the algorithmic foundation for compact observation structures [5, 8–12, 28, 30]. Network measurement systems then adapt these ideas to packet streams, where every update must fit a line-rate data-plane pipeline [21, 25, 27, 36, 50, 52, 56].

Existing approaches sit on an uncomfortable accuracy–latency trade-off. Flow sampling has a simple online path, but low sampling rates miss many sparse hot-flow bins and high rates require large flow tables that do not fit in fast SRAM [1, 6, 40]. Sketch-based methods such as MRAC are more compact, but their control-plane decoding relies on iterative EM. This EM step can take seconds to minutes for a single window, which is incompatible with real-time measurement. Later sketches improve hot-flow handling, compact counter usage, and telemetry integration, yet they still inherit the slow iterative decoding path for the full distribution [15, 17, 18, 24, 44, 48, 49, 51, 57]. Thus, improving only the data-plane representation is insufficient if reconstructing the final FSD remains computationally expensive. This raises the central question: *can an FSD estimator keep the data plane compact while making both insertion and decoding fast?*

We answer yes by rethinking decoding as a learned mapping from sketch states to FSDs. Rather than repeatedly solving an inverse problem from scratch for every measurement window, NeuFSD amortizes this computation into a learned decoder whose inference cost is fixed once the model is trained. NeuFSD uses a lightweight Hot Filter plus counter array in the data plane, reshapes the counter state into multiple 2-D views, and uses a Vision Transformer decoder to infer the FSD in one forward pass. For online operation, the model is pre-trained on diverse skewed distributions and continuously fine-tuned from low-rate sampled labels, so that sampled FSD labels can support adaptation. In summary, this paper makes the following contributions:

- **First Neural FSD Decoder:** We present NeuFSD, the first approach to apply neural networks to the FSD network measurement task. This paradigm shift breaks the long-standing "impossible triangle" of memory, accuracy, and speed that has constrained previous algorithms.
- **Design Analysis:** We go beyond black-box application and analyze the Hot Filter capture behavior and the counter-array statistics that determine the neural decoder's input distribution.
- **Superior Performance:** Extensive evaluations on real-world traffic traces demonstrate that NeuFSD reduces decoding latency by orders of magnitude, with separate CPU-only and GPU-accelerated control-plane measurements. Furthermore, it reduces recorded estimation error by up to two orders of magnitude under the stated scoring conventions and counter-array budgets.
- **Hardware Deployability:** We implement the online insertion logic of NeuFSD on a commodity Barefoot Tofino programmable switch and quantify its hardware resource overhead, showing that NeuFSD requires only a few pipeline stages and modest SRAM/stateful resources compared to prior art. These results demonstrate that NeuFSD's data-plane design is practical for line-rate deployment without introducing prohibitive hardware overhead.

Section 2 reviews the problem and baselines. Section 3 presents NeuFSD. Section 4 gives the design analysis, and Section 5 evaluates accuracy, speed, and deployability. Source code is available at [32].

2 Background

2.1 Problem Statement

In this section, we formally define the network traffic model and the FSD estimation problem.

Definition 2.1 (Network Stream Model). A network stream is modeled as an ordered packet sequence $S = (p_1, p_2, \dots, p_N)$. Each packet belongs to one and only one flow, denoted as e . The induced packet multiset can be classified into n non-overlapping flows, $\mathcal{E} = \{e_1, e_2, \dots, e_n\}$. The number of packets in a flow e_i is called the size or frequency of e_i , denoted as $f_i = |\{t : p_t \in e_i\}|$.

Definition 2.2 (Flow Size Distribution (FSD)). The Flow Size Distribution (FSD) describes the distribution of flow sizes within the stream. Formally, for any positive integer i , we define n_i as the number of flows with a size of exactly i :

$$n_i = |\{e_j \in \mathcal{E} \mid f_j = i\}|$$

The complete FSD is the set of all such counts, $\{n_i\}_{i \geq 1}$. The objective of FSD estimation is to accurately and efficiently estimate the value of n_i for all relevant flow sizes i .

Why the small-flow population matters. Unlike total packet volume, FSD records the number of flows at each size. The same packet volume may be distributed across many short flows or a few large ones, so tracking hot flows alone cannot reveal the small-flow population. Errors in populous small-flow bins can contribute substantially to WMRD, which measures aggregate flow-count error (Section 5.1.6). Accurate FSD estimation must therefore account for both the small-flow population and the large-flow tail.

2.2 Related Work

The FSD estimation problem is a classic and extensively studied task in network measurement. Existing solutions can be broadly categorized into two main families: sampling-based algorithms and sketch-based algorithms that rely on EM decoding.

Beyond these two FSD-specific families, the measurement literature has developed many data-plane summaries for related tasks, including reversible and invertible sketches, compact hash tables, counter-sharing structures, unbiased updates, and spread estimators [14, 20, 23, 33, 37, 42, 43, 45]. Other systems explore learned or programmable telemetry, compact counter encodings, and stable large-scale summaries [16, 19, 22, 39, 47]. These designs improve particular measurement primitives, but NeuFSD targets the harder full-distribution query: recovering the complete FSD quickly from a compact line-rate data-plane state.

Sampling-based methods operate by sampling flows with a probability $q \in (0, 1]$. For a given flow, all its packets are either sampled or discarded. If $n_i^{(q)}$ is the observed number of flows of size i in the sampled stream, the original count n_i can be estimated as $n_i^{(q)}/q$. This provides an unbiased estimator for n_i . However, sampling faces a severe accuracy-storage dilemma.

- **Low sampling rates** ($q \ll 1$) result in poor accuracy. If the original n_i is small, it is highly probable that none of these flows will be sampled, yielding $n_i^{(q)} = 0$. This makes it impossible to reconstruct the original distribution's tail.
- **Medium sampling rates**, while more accurate, introduce a significant storage bottleneck. If a *hash table* is used to store sampled flows, each entry becomes large (~ 160 bits) to store flow labels (~ 100 bits), collision-handling pointers (~ 32 bits), and packet counters (~ 32 bits). On backbone links, such a table can only fit in DRAM, but DRAM's access speed cannot keep up with line rates. Alternatively, if a *flow table* (e.g., an array) is used to save pointer overhead, packet insertion requires a linear scan to find the flow entry, which is prohibitively slow and negates the high-speed advantage of sampling.

Sketch-based methods (via EM decoding) were pioneered by MRAC [18]. This approach is supported by Bayesian statistics. In the data plane, MRAC hashes packets into a compact, single-row counter array (akin to a Count-Min Sketch). After insertion, the sketch is passed to the control plane for offline decoding. MRAC uses an iterative *EM* algorithm to recover the original FSD until convergence. *In other words, these methods achieve line-rate data-plane measurement by shifting much of the reconstruction complexity to the control-plane decoder.*

While this method is memory-efficient and more accurate than sampling, its critical flaw is the extremely slow decoding speed. The EM algorithm is computationally intensive, with a runtime proportional to the number of packets and counters. For example, processing a stream of 100,000 packets with 10,000 32-bit counters can take approximately **two minutes** for the decoder to converge. In modern networks processing millions of packets per second, this high offline latency is unacceptable for real-time analysis. Moreover, this iterative decoding cost is incurred for every measurement window, making the decoder a persistent bottleneck in online monitoring. Subsequent improvements, such as Elastic Sketch [49], FCM [41], and DaVinci Sketch [44], essentially add modules to the MRAC structure (e.g., to separate hot flows and cold flows), which improves accuracy. However, they are still fundamentally reliant on an EM-based decoder, inheriting its high latency and failing to achieve true real-time query speeds. This motivates replacing iterative reconstruction with a faster decoding paradigm while retaining a compact data-plane sketch.

3 The NeuFSD Framework

In this section, we provide a detailed description of the NeuFSD architecture and its algorithmic framework. We begin by introducing a direct, or "naive," neural decoding approach for FSD estimation. We will then analyze its limitations, which will motivate the full, optimized design of NeuFSD presented in the subsequent sections.

3.1 A Direct Neural Decoding Approach

As motivated in Section 1, the most straightforward idea is to directly replace the slow, iterative EM decoding algorithm of MRAC with a neural network. This direct approach, illustrated in Figure 1, operates in two distinct phases.

1. Online Data Plane Insertion.

This phase operates in the high-speed data plane, which could be implemented on various platforms such as a P4-programmable switch, an FPGA, or a high-performance software switch (e.g., Open vSwitch, BESS). A single counter array, A , of length m is maintained in the data plane memory (e.g., SRAM). Upon receiving each packet p , its flow label is hashed to an index $j = h(p)$, and the counter is incremented: $A[j] \leftarrow A[j] + 1$.

2. Offline Control Plane Decoding. When an FSD estimation is required, the entire counter array A (of size m) is transferred to the control plane (e.g., a host CPU). Before this array can be used, a critical preprocessing step is performed: **the counter array A is sorted** to produce a vector $\mathbf{a}_{\text{sorted}} \in \mathbb{R}^m$. The rationale for sorting is crucial. The index of any counter is arbitrary and determined by the hash function $h(\cdot)$. Sorting removes this counter-address order and exposes the

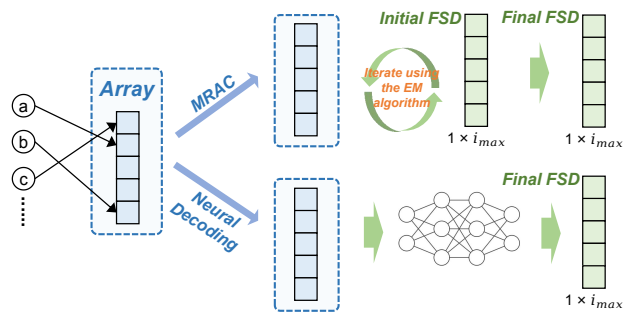


Fig. 1. Comparison of MRAC and Neural Decoding.

empirical distribution of counter values in a canonical order. Different hash realizations can still produce different collision patterns; training with varied hash seeds covers this variation.

Formally, we define the neural network as a parameterized function f_θ , where θ represents all the learnable weights and biases. This function performs a mapping from the space of the sorted counter array to the space of the Flow Size Distribution. As defined in Section 2.1, the FSD is a set of counts $\{n_i\}$. We represent this as an output vector $\hat{\mathbf{n}} = [\hat{n}_1, \hat{n}_2, \dots, \hat{n}_{i_{\max}}]$, where $i_{\max}$ is the maximum flow size of interest.

The decoding process is thus a single, non-linear transformation: $\hat{\mathbf{n}} = f_\theta(\mathbf{a}_{\text{sorted}})$. This single forward pass replaces the entire iterative EM algorithm, providing an almost instantaneous estimation.

Training this Network. The network's parameters θ are learned by minimizing a loss function $\mathcal{L}$ that measures the discrepancy between the predicted FSD ($\hat{\mathbf{n}}$) and the ground-truth FSD ($\mathbf{n}$). The training objective is:

$$\min_{\theta} \mathbb{E}_{(\mathbf{a}_{\text{sorted}}, \mathbf{n}) \sim \mathcal{D}} [\mathcal{L}(\hat{\mathbf{n}}, \mathbf{n})]$$

where $\mathcal{D}$ is the distribution of training data.

A practical advantage is that synthetic training pairs $(\mathbf{a}_{\text{sorted}}, \mathbf{n})$ can be generated on demand. We vary the target flow-size distribution, for example by changing the parameters of a Zipfian distribution, and simulate insertion using different hash seeds. For a fixed target FSD, different seeds can produce different counter arrays paired with the same ground-truth FSD.

For deployment, there are three primary strategies:

- **Offline Pre-training:** A large, general-purpose model is trained once on synthetic distributions with varied skewness, as specified in Section 3.4.
- **Online Training:** The model is trained (or re-trained) periodically, using recent traffic from the target network (e.g., a campus network with its own unique profile) to build the training set. This allows the model to adapt to the specific and evolving characteristics of live traffic [55].
- **Hybrid (Fine-tuning):** A pre-trained model is used as a starting point and then rapidly fine-tuned on a small amount of recent, local data.

In this paper, we primarily utilize the **online training** strategy in our experiments. We find that it is fast enough to keep the model current and achieves high accuracy, demonstrating the practicality of the neural decoding approach.

3.2 Potential Challenges of the Direct Approach

The "straw man" method described in Section 3.1 seems plausible at first glance, but it suffers from several critical challenges that must be addressed to create a practical solution.

- (1). **Indeterminate Output Dimension:** The neural network is defined as a mapping $f_\theta : \mathbb{R}^m \rightarrow \mathbb{R}^{i_{\max}}$. The input dimension, m (the counter array size), is a fixed hyperparameter chosen at design time. However, the output dimension, which corresponds to the maximum flow size $i_{\max}$, is **unknown a priori**. The value of $i_{\max}$ is dynamic and data-dependent; we do not know in advance whether the largest flow in a given measurement interval will have 10,000 packets or 10,001. This makes it impossible to define a fixed output layer architecture for the neural network.
- (2). **Potential for Parameter Explosion:** Even if we could determine a fixed, worst-case $i_{\max}$, this value can be extraordinarily large. As shown in Figure 2, real million-packet measurement windows from CAIDA2016, CAIDA2018, IMC, and MAWI already contain hot flows with thousands to hundreds of thousands of packets; in the selected windows, $i_{\max}$ ranges from 4,989 to 197,410. The input dimension m is also large; a 128KB sketch with 32-bit counters yields $m = 32,768$. A simple MLP with an input layer of 32,768 neurons and an output layer sized for such a worst-case $i_{\max}$ would be massive. Even a single, modest hidden layer would result in millions of parameters,

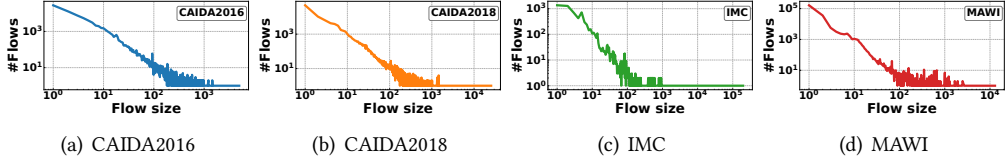


Fig. 2. FSD curves from one real million-packet window in each dataset.

leading to significant GPU memory consumption and prohibitively slow training times. Therefore, a parameter-efficient architecture is essential.

(3). Need for High-Fidelity Tail Estimation: The FSD of real-world network traffic is notoriously long-tailed, as illustrated in Figure 2. The dense body is dominated by cold flows, while the tail of the distribution contains sparse hot flow bins whose counts n_i are often small integers (e.g., 1, 3, or 6). Accurately estimating these small counts is extremely difficult for a standard neural network, yet it is **critically important** for many network applications.

Consider the severe consequences of mis-estimating n_{10000} from a true value of 3 to a predicted value of 6. While the absolute error is small, this 100% relative error can be disastrous. For a load balancer or QoS system, this error means provisioning dedicated resources for three non-existent hot flows, leading to significant resource waste. Conversely, underestimating n_{10000} from 3 to 1 would fail to provision for two real hot flows, causing them to contend with general traffic and leading to severe network congestion and SLA violations. This highlights the need for a mechanism to specifically enhance estimation accuracy for the sparse, high-magnitude tail of the distribution.

Successfully resolving these three challenges—an indeterminate output dimension, parameter explosion, and poor tail accuracy—is the key to unlocking a practical and high-performance neural FSD decoder. This is precisely what the NeuFSD framework, detailed next, is designed to do.

3.3 The Complete NeuFSD Framework

Figure 3 illustrates the complete architecture of NeuFSD, which is meticulously designed to solve the three challenges identified in Section 3.2. It consists of a data plane insertion stage and a control plane decoding stage.

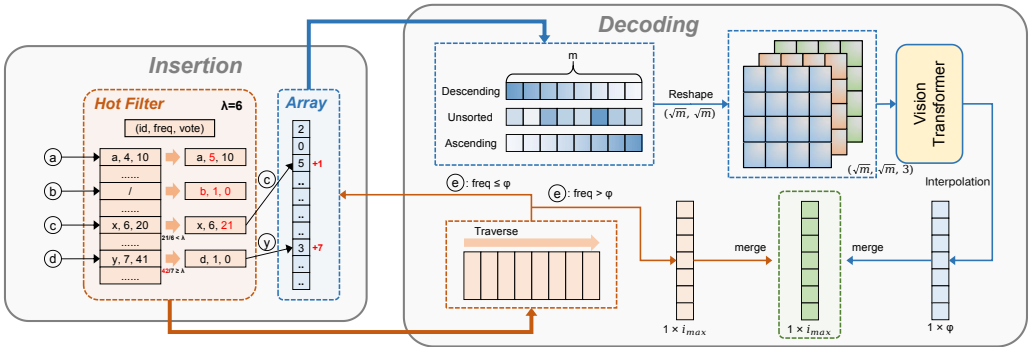


Fig. 3. Complete architecture of the NeuFSD framework.

3.3.1 Online Insertion. To solve Challenge #3 (poor tail accuracy), we introduce a **Hot Filter** that operates in front of the main counter array A , designed to accurately isolate and store hot flows. The design of the Hot Filter draws inspiration from the hot flow component of Elastic Sketch [49],

Algorithm 1: NeuFSD Online Insertion: Insert(p)

Input: A packet p with flow label e ; Array A ; Hot Filter B ; Hash functions h_B, h_A ; Eviction threshold λ

- 1 Assume $B[h_B(e)]$ is the target bucket, and $\langle e_{\min}, f_{\min} \rangle$ is the entry with min frequency in $B[h_B(e)]$;
- 2 **if** e exists in an entry $B[h_B(e)][j]$ **then**
- 3 $B[h_B(e)][j].frequency++$; ▷ **Case 1**
- 4 **else if** an empty entry $B[h_B(e)][j]$ exists **then**
- 5 put $\langle e, 1 \rangle$ into $B[h_B(e)][j]$; ▷ **Case 2**
- 6 **else**
- 7 $B[h_B(e)].vote++$;
- 8 **if** $\frac{B[h_B(e)].vote}{f_{\min}} < \lambda$ **then**
- 9 $A[h_A(e)]++$; ▷ **Case 3**
- 10 **else**
- 11 Replace $\langle e_{\min}, f_{\min} \rangle$ with $\langle e, 1 \rangle$; ▷ **Case 4**
- 12 $B[h_B(e)].vote = 0$;
- 13 $A[h_A(e_{\min})] += f_{\min}$; ▷ Replay evicted flow into A

with modifications to optimize for FSD estimation: demoted flows are deterministically replayed into the counter array, and at query time all non-hot entries in the filter are folded into the same array before neural decoding. This keeps the residual sketch mass in a single estimator and makes the final FSD splice explicit.

The Hot Filter consists of a table B with w buckets, i.e., $B[1..w]$. Each packet p (belonging to flow e) is mapped to a bucket $B[h_B(e)]$ using a hash function h_B . Each bucket contains E entries and a single *vote* field. Each entry is a tuple $\langle e, f \rangle$, dedicated to recording a distinct flow label e and its frequency f .

The complete insertion logic for an arriving packet p is detailed in Algorithm 1. We hash its flow label e to locate the bucket $B[h_B(e)]$ and check all E entries, encountering one of the following **four cases**:

Case 1: If flow e is already stored in an entry, we increment its frequency by 1.

Case 2: If e is not in the bucket, but there is an empty entry, we insert $\langle e, 1 \rangle$ into that entry.

Case 3: If e is not in the bucket and there are no empty entries, we increment $B[h_B(e)].vote$. Let $\langle e_{\min}, f_{\min} \rangle$ be the entry with the smallest frequency in the bucket. If $\frac{vote}{f_{\min}} < \lambda$ (the eviction threshold), the new flow e is not promoted. We treat this packet p as a "cold" packet and insert it directly into the main counter array A (i.e., $A[h_A(e)] \leftarrow A[h_A(e)] + 1$).

Case 4: Continuing from Case 3, if $\frac{vote}{f_{\min}} \geq \lambda$, we deem the new flow e to be "hot." We replace the "coldest" flow $\langle e_{\min}, f_{\min} \rangle$ with the new flow $\langle e, 1 \rangle$ and reset *vote* to 0. The evicted flow $\langle e_{\min}, f_{\min} \rangle$ is now demoted; we "replay" its packets into the main array A by adding its last known frequency $f_{\min}$ to the corresponding counter (i.e., $A[h_A(e_{\min})] \leftarrow A[h_A(e_{\min})] + f_{\min}$).

Rationale. This logic encourages hot flows to remain in the Hot Filter. The key lies in the eviction threshold λ , which is inspired by the concept of *ostracism*. A high *vote* value signals that many new flows are competing for a slot. When $\frac{vote}{f_{\min}} \geq \lambda$, we "exile" the current coldest flow $x_{\min}$ to the array A to make space for the new, potentially hot flow x . This mechanism helps hot flows rapidly climb into and stay within the Hot Filter.

The deterministic update rule makes packet accounting explicit. Let ϕ denote the hot-flow threshold used during decoding, and let $M_{\text{hot}} = \sum_{\langle e, f \rangle \in B, f > \phi} f$ be the total packet count stored in Hot Filter entries above this threshold. After all remaining entries are folded into A , the counters sum to exactly $N - M_{\text{hot}}$, where N is the number of packets in the window. When all flows larger than ϕ are fully captured, this residual equals the total packet count of flows of size at most ϕ . Otherwise, packets from missed or partially captured hot flows remain in A and are accounted for by the splice-error analysis in Section 4.3.

We further validate the practicality of this data-plane design by implementing the online insertion logic on a Barefoot Tofino switch and reporting its resource overhead in Section 5.5.

3.3.2 Offline Decoding. At query time, both the Hot Filter B and the counter array A are transferred to the control plane. The decoding process is designed to precisely solve the challenges of output dimension and parameter explosion.

1. Solving Challenge 1 & 3 (Tail & Output Dimension). We first establish a hot-flow threshold ϕ (the final evaluation uses $\phi = 100$). We iterate through all entries $\langle x, f \rangle$ in the Hot Filter B .

- All flows with $f > \phi$ are extracted. We directly count them to form the high-fidelity tail of the FSD, $\text{fSD}_{\text{hot}} = \{n_i \mid i > \phi\}$. This provides accurate accounting for the hot flows retained by the Hot Filter.
- All remaining flows in the filter (where $f \leq \phi$) are then "folded" into the array A by adding their frequencies: $A[h_A(x)] \leftarrow A[h_A(x)] + f$.

After this step, array A contains the residual sketch component that ideally corresponds to flows with size $\leq \phi$. If a hot flow is missed, demoted, or only partially captured, its residual mass is also folded into A and is treated as splice error in Section 4.3. The neural network's task is therefore to estimate the residual FSD over $\{1, \dots, \phi\}$, a fixed-dimensional output.

2. Solving Challenge 2 (Parameter Explosion). Instead of a simple MLP, we use a **Vision Transformer (ViT)** architecture with shared patch projection and a short token sequence.

Input Preprocessing. We first transform the 1D counter array A (of size m) into a 3-channel "image." We construct three "views" of the array: the ascending view A_{asc} , the unsorted view A_{unsort} (which preserves counter indexing), and the descending view A_{desc} . The sorted views represent the empirical counter-value distribution in opposite directions, as discussed in Section 3.1. For reshaping, we choose $m = d^2$ counters, where d is a positive integer. Each view is reshaped in row-major order into a $d \times d$ matrix. Stacking the ascending, original, and descending views as channels gives $\mathbf{X} \in \mathbb{R}^{3 \times d \times d}$. The default configuration uses $d = 64$ ($m = 4096$ counters).

ViT Architecture. This "image" $\mathbf{X}$ is fed into the ViT.

- (1) *Patch Embedding:* $\mathbf{X}$ is divided into a sequence of N_p non-overlapping patches. Each patch is flattened and linearly projected into a D -dimensional embedding vector. Positional embeddings $\mathbf{E}_{\text{pos}}$ are added to this sequence.
- (2) *Transformer Encoder:* The sequence of patch embeddings is processed by a stack of L standard Transformer layers. Each layer consists of Multi-Head Self-Attention (MHSA) and an MLP block. $\mathbf{z}'_l = \text{MHSA}(\text{LN}(\mathbf{z}_{l-1})) + \mathbf{z}_{l-1}$, $\mathbf{z}_l = \text{MLP}(\text{LN}(\mathbf{z}'_l)) + \mathbf{z}'_l$
- (3) *Output Head:* The final output state $\mathbf{z}_L$ is passed to a small MLP "head" to produce FSD regression coordinates.

Output coordinates. Two independent instances, each with its own encoder, predict 10 and 1,080 coordinates. These correspond to sizes $1, \dots, 10$ and $11, \dots, 1000 \cup 1001, 1101, \dots, 9901$, respectively. Their parameter counts are 86,440,962 and 87,512,032, totaling 173,952,994. At $\phi = 100$, the final neural estimate uses sizes 1–10 from the first model and 11–100 from the second; these coordinates are predicted directly. Patch projection shares weights across counter groups, while attention combines

Algorithm 2: NeuFSD Offline Decoding

Input: Hot Filter B ; Counter Array A ; Trained regressors f_{10}, f_{1080} ; Hot Flow Threshold ϕ ;
 Packet count N ; Clip interval $0 \leq s_{\min} \leq 1 \leq s_{\max} < \infty$

Output: The final FSD vector $\mathbf{fsd}$

```

1 Initialize  $\mathbf{fsd}_{\text{hot}}$  as an all-zero vector;
2  $A^{\text{fold}} \leftarrow A$ ;  $M_{\text{hot}} \leftarrow 0$ ;
3 foreach entry  $\langle e, f \rangle$  in Hot Filter  $B$  do
4   if  $f > \phi$  then
5      $\mathbf{fsd}_{\text{hot}}[f] \leftarrow \mathbf{fsd}_{\text{hot}}[f] + 1$ ;
6      $M_{\text{hot}} \leftarrow M_{\text{hot}} + f$ ;
7   else
8      $A^{\text{fold}}[h_A(e)] \leftarrow A^{\text{fold}}[h_A(e)] + f$ ;
9  $\mathbf{a}_{\text{asc}} \leftarrow \text{SortAscending}(A^{\text{fold}})$ ;
10  $\mathbf{a}_{\text{unsort}} \leftarrow A^{\text{fold}}$ ;
11  $\mathbf{a}_{\text{desc}} \leftarrow \text{SortDescending}(A^{\text{fold}})$ ;
12  $\mathbf{X} \leftarrow \text{ReshapeAndStack}(\mathbf{a}_{\text{asc}}, \mathbf{a}_{\text{unsort}}, \mathbf{a}_{\text{desc}})$ ;
13  $\mathbf{u} \leftarrow f_{10}(\mathbf{X})$ ;  $\mathbf{v} \leftarrow f_{1080}(\mathbf{X})$ ;
14  $\mathbf{n}[1..10] \leftarrow \mathbf{u}$ ;  $\mathbf{n}[\mathcal{G}] \leftarrow \mathbf{v}$ ;  $\triangleright \mathcal{G} = \{11, \dots, 1000\} \cup \{1001, 1101, \dots, 9901\}$ ;
15  $\mathbf{fsd}_{\text{cold}} \leftarrow \max(0, \text{SelectAndInterpolate}(\mathbf{n}, 1.. \phi))$ ;  $\triangleright$  All sizes  $1, \dots, 100$  are model outputs.
16  $R \leftarrow N - M_{\text{hot}}$ ;  $\widehat{R} \leftarrow \sum_{i=1}^{\phi} i \mathbf{fsd}_{\text{cold}}[i]$ ;
17 if  $\widehat{R} > 0$  then
18    $s \leftarrow \text{clip}(R/\widehat{R}, s_{\min}, s_{\max})$ ;
19    $\mathbf{fsd}_{\text{cold}} \leftarrow s \mathbf{fsd}_{\text{cold}}$ ;
20  $\mathbf{fsd} \leftarrow \mathbf{fsd}_{\text{hot}} + \text{ZeroPad}(\mathbf{fsd}_{\text{cold}})$ ;
21 return  $\mathbf{fsd}$ ;
```

information across the short token sequence. The counter views and flow-count regression replace the natural-image input and classification task of a conventional ViT; neighboring counters need not have image semantics. The complete decoder architecture and hyperparameter configuration are provided in Appendix A and illustrated in Figure 20.

This two-pronged approach—patch-based encoding and fixed output coordinates—provides a fixed-dimensional decoding interface for rapid training and inference. The final FSD is the union of the two parts: $\mathbf{fsd}_{\text{final}} = \mathbf{fsd}_{\text{hot}} \cup \mathbf{fsd}_{\text{cold}}$. The complete pseudo-code, including the residual-mass calibration in Section 3.6, is shown in Algorithm 2.

3.4 Online Measurement Protocol

The data-plane sketch defines what is exported at the end of a window; an online system must also decide how to keep the decoder current. Exact recent FSD labels are unavailable without exporting all flows, and a model trained once can become stale as traffic changes. NeuFSD therefore uses a **pre-training + online sampling + continuous supervised fine-tuning (SFT)** protocol. Offline pre-training on diverse Zipf-like and long-tailed streams gives the decoder a broad prior [7, 13, 26]. Online, a low-rate sampled stream supplies approximate FSD labels for model adaptation. The history-based workflow is described below; the implementation details are summarized separately.

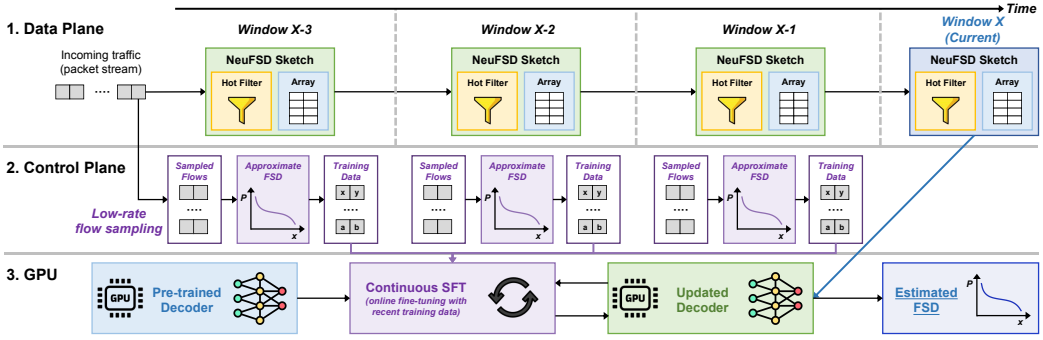


Fig. 4. Online measurement workflow of NeuFSD.

Sampling and label construction. Sampling selects entire flows within each observation window. A stable 64-bit BLAKE2b hash of the flow key, window index, and sampling seed selects a flow with probability q ; all its packets determine its observed size. Each selected flow of size i contributes $1/q$ to the reconstructed count at i . The matched history-based experiments use $q = 0.1$ and sampling seed 42; the sampled packet fraction may differ from q .

The reference label builder counts all window keys in a host dictionary before selection, with no configured finite capacity or overflow policy. Its state is separate from the counter-array budget. Sampled regression targets are obtained by linear interpolation and endpoint extrapolation of occupied size/count pairs, with negative values clipped to zero. Synthetic sketch/label pairs use rounded reconstructed flow populations, fresh flow identities, and varied counter hash seeds. Exact synthetic training labels instead retain zero counts at unoccupied output coordinates.

In the history-based protocol, at the beginning of window t , the control plane reconstructs approximate labels from sampled windows before t , synthesizes training counters with fresh hash seeds, and fine-tunes the decoder before querying the snapshot for window t . Each SFT update uses preceding-window sampled labels; the prior's training settings are specified in detail.

Update configuration. The first five windows provide history, and one update precedes each subsequent window's decoding. In the matched Avg-5 experiment, five synthetic identity replicas of the averaged preceding-window FSD, each with 400 counter seeds, produce 2,000 examples per model. A random 1,800/200 split supplies training and diagnostic holdout examples. Both models update all parameters for five epochs with batch size 256 and Adam at learning rate 0.01, using mean SmoothL1 loss with $\beta = 1$. Model weights carry forward, Adam state is reset at each update, and the final epoch supplies the updated decoder. The holdout separates counter instances of the reconstructed traffic, rather than traffic distributions.

Pre-training supplies a general prior, while continuous SFT tracks local traffic. Figure 4 summarizes the workflow. In this history-based workflow, the sampled stream supports model adaptation outside the packet-insertion path.

Pre-training data and split. The initial prior uses 60 synthetic Zipf datasets. Each dataset supplies 400 counter instances, giving 24,000 examples per model. Reserving the final three datasets of each input root gives 21,600 training and 2,400 validation examples. Pre-training uses 20 epochs, batch size 64, and Adam at learning rate 0.01; the checkpoint with lowest validation WMRD is retained. The generated datasets vary flow sizes and counts, so the corresponding input arrays cover diverse traffic shapes for testing model initialization and adaptation. These settings cover initialization with diverse synthetic inputs. Full model and training settings are given in Appendix D.

Decoder configuration. The final IMC comparison stores counter views in a fixed channel order and uses contiguous input tensors, keeping the model interface consistent across snapshots before

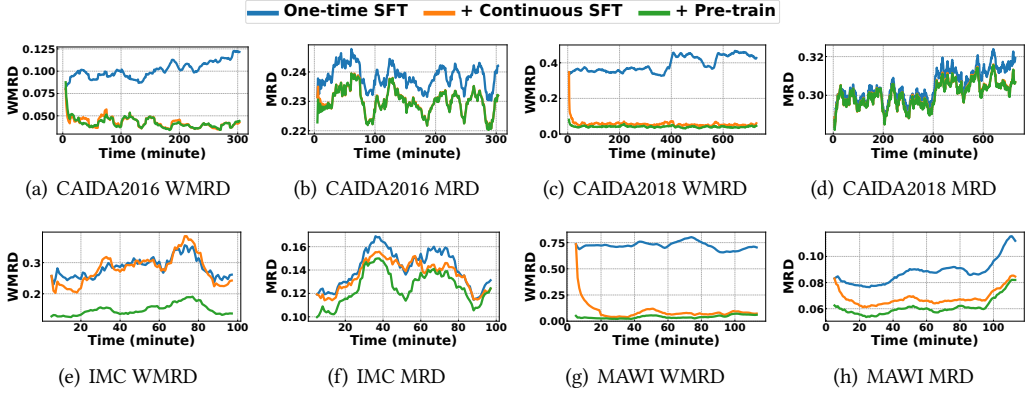


Fig. 5. Online measurement ablation on four real traces.

neural decoding. Output reconstruction variants use nonnegative clipping before merging decoded predictions. These details are documented alongside the model configuration in the experimental settings and Appendix D.

Figure 5 shows that one-time SFT is usually the weakest and can drift as the trace moves away from the initial windows. For display, the first continuous-SFT point is aligned to the fixed-model point before 15-window smoothing; the pre-trained curve also includes its configured gate correction. The displayed pre-trained configuration is the most stable overall; the supplementary matched replay reports unaligned, raw predictions. Dataset and metric definitions appear in Sections 5.1.3 and 5.1.6. Section 5 shows that sampling, training-set construction, and SFT take only seconds per 1-minute window.

3.5 Continuous SFT Window Selection

The remaining design choice is which past sampled FSDs form the SFT label. The history length K is configurable: larger K smooths noise, while smaller K reacts faster. We use $K = 5$ and compare three causal strategies (Figure 6): Last-1 uses only window $t - 1$, Window-5 uses the five FSDs from $t - 5$ to $t - 1$ as separate targets, and Avg-5 averages those five curves into one target.

In the matched strategy comparison, all three choices generate 2,000 examples per model and use the same 1,800/200 split. Window-5 uses five distinct preceding-window FSDs with 400 counter seeds each. Last-1 and Avg-5 each use five synthetic identity replicas of their selected or averaged FSD, again with 400 seeds per replica. Avg-5 averages counts over the union of observed sizes, retaining the count scale used by the regression loss.

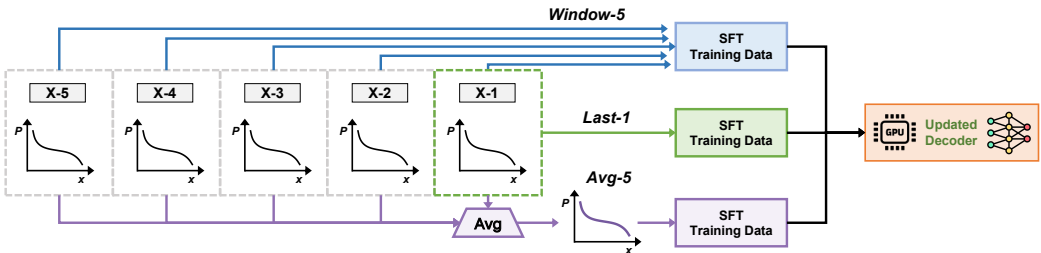


Fig. 6. Three causal strategies for continuous SFT labels.

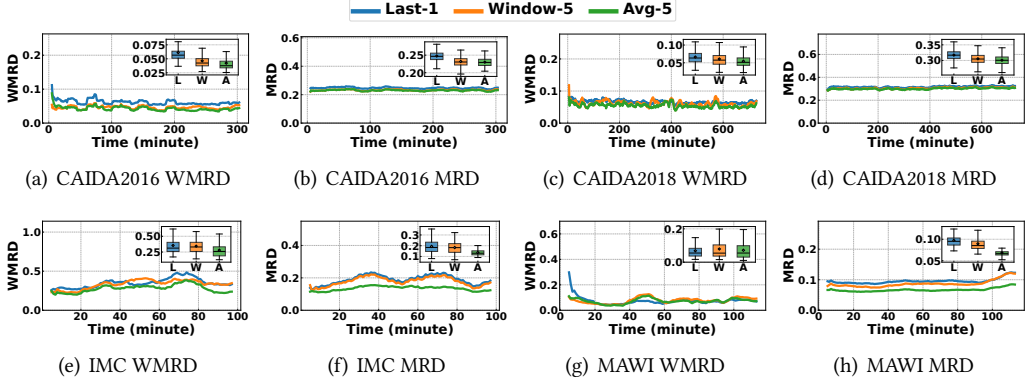


Fig. 7. Continuous SFT strategy comparison.

Figure 7 uses the same pre-trained decoder, SFT budget, and per-window update schedule for all strategies. Avg-5 gives the lowest overall mean error: WMRD 0.112 vs. 0.132 (Window-5) and 0.137 (Last-1), and MRD 0.184 vs. 0.202 and 0.214. We therefore use Avg-5 by default.

3.6 Optimization: Residual-Mass Calibration

Even with continuous SFT, short traffic bursts can produce a vertical scale error in the residual FSD. NeuFSD therefore applies a lightweight post-processing step: *residual-mass calibration*. Let N_t be the packet count in window t , and let $M_{\text{hot},t} = \sum_{(e,f) \in B_t, f > \phi} f$ be the packet mass directly explained by hot-spliced Hot Filter entries. The residual sketch mass is $R_t = N_t - M_{\text{hot},t}$.

It is available online and does not require the true FSD. If the decoder predicts nonnegative residual counts $\hat{n}_{t,x}$ for $x \leq \phi$, with mass $\hat{R}_t = \sum_{x=1}^{\phi} x \hat{n}_{t,x} > 0$, we keep the decoded shape and rescale the y axis: $\tilde{n}_{t,x} = s_t \hat{n}_{t,x}$, $s_t = \text{clip}(R_t / \hat{R}_t, s_{\min}, s_{\max})$, and leave the hot-flow portion unchanged; if $\hat{R}_t = 0$, calibration is skipped. Without clipping, the calibrated curve satisfies $\sum_{x=1}^{\phi} x \tilde{n}_{t,x} = R_t$ exactly. When hot-flow spillover remains in A , R_t is a residual sketch-mass constraint rather than the true cold-flow mass; clipping keeps the correction conservative under splice error. In our experiments, $s_{\max} = 1.20$, and $s_{\min} = 1$ for one-sided up-scaling or $s_{\min} = 0.70$ for validated two-sided calibration.

Figure 8 plots unsmoothed per-window errors, so it is more jagged than Figure 7. The configured corrections lower mean error on every trace and metric; for example, WMRD drops from 0.0545 to 0.0411 on CAIDA2018, from 0.2806 to 0.1501 on IMC, and from 0.0717 to 0.0430 on MAWI. The gain is larger for WMRD because WMRD is dominated by high-population cold-flow bins.

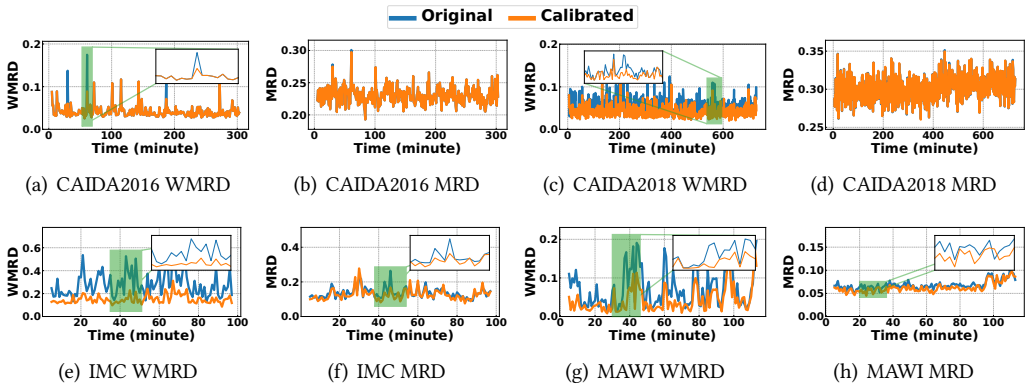


Fig. 8. Effect of dataset-specific residual and sampled-shape correction.

4 Mathematical Analysis

This section summarizes the design analysis; full definitions and proofs are in Appendix C. The main point is order awareness: the Hot Filter is deterministic once the hash seeds and packet order are fixed, so probability statements must specify whether randomness comes from hashing, packet order, or a traffic model.

4.1 Hot Filter: Order-Aware Capture Analysis

ASSUMPTION 1 (HASHING AND ARRIVAL-ORDER MODEL). *The traffic sequence is oblivious to the hash seeds, and for distinct flows $x \neq y$, $\Pr[h_B(x) = h_B(y)] = 1/w$. When second-moment bounds are used, the corresponding hash indicators are pairwise independent. For order-dependent capture probabilities we use either a random permutation of a fixed packet multiset or an independent Poisson arrival model. No nontrivial mass-only capture bound is claimed under adversarial packet order.*

For a target flow e , let f_e be its final flow size and $J_e = h_B(e)$ denote its corresponding Hot Filter bucket. The total packet mass contributed by all other flows colliding with e in this bucket is

$$G_e = \sum_{x \neq e} f_x \mathbf{1}\{h_B(x) = J_e\}.$$

The final colliding mass G_e alone does not determine the Hot Filter state. With $E = 1, \lambda = 2$, the two sequences (e, e, e, a, a, a) and (a, a, a, e, e, e) have the same f_e and G_e , but only the first captures e . A second construction in Appendix C.3 shows that a large final colliding mass can still end with e captured. The deterministic object used in the proof is therefore prefix pressure after admission.

LEMMA 4.1 (PREFIX-PRESSURE CONDITION). *Fix the hashes and any packet order. Suppose e is admitted at time τ with count $c_\tau(e) \geq 1$. Define $C_e(u) = c_\tau(e) + |\{t : \tau < t \leq u, e_t = e\}|$, which equals its stored count while it remains resident. Let $M_e(u)$ count non- e packets hashing to J_e after τ up to u . If e is later evicted, then for some prefix u , $M_e(u) \geq \lambda C_e(u)$.*

Let $\text{Miss}_B(e)$ denote the event that e is absent from its Hot Filter bucket at the final snapshot. Under random order, Lemma 4.1 gives a conservative admission-survival bound.

THEOREM 4.2 (ADMISSION-SURVIVAL BOUND UNDER RANDOM ORDER). *Fix h_B , condition on integer $G_e = g$, and let $f = f_e \geq 2$. Under the random permutation model,*

$$\Pr(\text{Miss}_B(e) \mid G_e = g) \leq \min\{1, p_{\text{adm}}^c(f, g, E) + p_{\text{surv}}^c(f, g, \lambda)\},$$

where $p_{\text{adm}}^c(f, g, E) = \binom{g}{E} / \binom{f+g}{E}$ for $g \geq E$ and 0 otherwise. Moreover, $p_{\text{adm}}^c \leq (g/(f+g))^E$. With $\Delta = (f-1)/(f+g-1) - 1/(\lambda+1)$, one conservative survival term is

$$p_{\text{surv}}^c(f, g, \lambda) = \begin{cases} \min\{1, \sum_{\ell=1}^{f+g-1} \exp(-2\ell\Delta^2)\}, & \Delta > 0, \\ 1, & \Delta \leq 0. \end{cases}$$

COROLLARY 4.3 (HASH-AVERAGED CAPTURE IMPLICATION). *Let $F_{-e} = \sum_{x \neq e} f_x$ and assume $f_e \geq 2$. For any $g_0 > 0$,*

$$\Pr(\text{Miss}_B(e)) \leq \min\left\{1, \frac{F_{-e}}{wg_0} + \sup_{g \in \{0, \dots, \lfloor g_0 \rfloor\}} [p_{\text{adm}}^c(f_e, g, E) + p_{\text{surv}}^c(f_e, g, \lambda)]\right\}.$$

Thus the Hot Filter analysis separates sufficient design bounds: larger w reduces expected collision mass, larger E improves the early-admission bound, and larger λ increases the sufficient survival margin but can delay replacement-based admission. These bounds do not establish a pathwise monotone ordering of actual multi-entry eviction outcomes.

4.2 Counter Array Statistics

At decoding time, entries with count $> \phi$ form the direct hot-flow tail and all other residual mass is folded into A . Because a flow may contribute several chunks over time, the analysis first aggregates by key. Condition on the packet sequence, h_B , and the resulting residual key masses $\{a_x : x \in \mathcal{U}\}$. Then

$$C_j = A[j] = \sum_{x \in \mathcal{U}} a_x \mathbf{1}\{h_A(x) = j\}. \quad \text{Let } R_A = \sum_x a_x, \quad F_{2,A} = \sum_x a_x^2.$$

THEOREM 4.4 (COUNTER EXPECTATION AND VARIANCE). *Assume h_A is independent of h_B and the oblivious traffic sequence. If h_A maps each distinct key uniformly to one of m counters and is pairwise independent across keys, then, conditional on the residual masses, $\mathbb{E}[C_j] = \frac{R_A}{m}$, $\text{Var}(C_j) = \frac{m-1}{m^2} F_{2,A}$.*

In the ideal split, $a_x = f_x$ for $f_x \leq \phi$; in the real algorithm, missed or demoted hot-flow residuals also enter A . This mismatch is the splice error below.

4.3 End-to-End Error Decomposition

PROPOSITION 4.5 (END-TO-END L_1 AND WMRD DECOMPOSITION). *Let n be the true FSD and n_B^{hot} the FSD vector from Hot Filter entries with count $> \phi$. After folding the remaining entries into the array, let $q_0 \in \mathbb{R}_{\geq 0}^\phi$ be the residual FSD over array-represented keys with residual mass at most ϕ ; any residual key with mass $> \phi$ is charged to splice error. Let $n_B^{\text{splice}} = n_B^{\text{hot}} + q_0$ after zero-padding to the full FSD support. Assume $0 \leq s_{\min} \leq 1 \leq s_{\max} < \infty$. For any window,*

$$\|n - \hat{n}\|_1 \leq \epsilon_{\text{splice}} + \epsilon_{\text{counter}} + \epsilon_{\text{approx}} + \epsilon_{\text{drift}} + \epsilon_{\text{sample}} + \epsilon_{\text{gen}} + \epsilon_{\text{mass}},$$

where $\epsilon_{\text{splice}} = \|n - n_B^{\text{splice}}\|_1$ and the next five terms capture counter ambiguity, neural approximation, online drift, sampled-label noise, and finite-training error. The additional term $\epsilon_{\text{mass}} = \|C_{R_A}(q_5) - C_{\hat{R}}(q_5)\|_1$ distinguishes the actual calibration target $R_A = N - M_{\text{hot}}$ from $\hat{R} = \sum_{i=1}^\phi i q_{0,i}$. Here q_5 is the nonnegative decoded vector and C is the clipped mass-rescaling operator. Residual keys above ϕ can make these masses unequal even when packets are conserved. Appendix C.8 gives all terms and proves $\epsilon_{\text{mass}} \leq |R_A - \hat{R}|$. For a positive denominator, $\text{WMRD}(n, \hat{n}) \leq \frac{\sum \epsilon}{(\|n\|_1 + \|\hat{n}\|_1)/2}$.

The bound is a decomposition, not a tight prediction; it identifies where system improvements act. The calibration-target term extends the decomposition to residual mass outside the decoded size support and vanishes when the two target masses coincide.

4.4 Residual-Mass Calibration

Calibration is applied only to the residual component. Let $n, \hat{n} \in \mathbb{R}_{\geq 0}^\phi$ be the true residual vector and the nonnegative decoder output after interpolation. Define $r = (1, 2, \dots, \phi)^\top$, $D = \text{diag}(1, \dots, \phi)$, $R = r^\top n > 0$, and $\hat{R} = r^\top \hat{n} > 0$. The unclipped scale is $s_0 = R/\hat{R}$ and the clipped scale is $s = \text{clip}(s_0, s_{\min}, s_{\max})$.

PROPOSITION 4.6 (MASS CONSTRAINT). *For $\tilde{n} = s_0 \hat{n}$, $r^\top \tilde{n} = R$. With clipping, $|r^\top(s\hat{n}) - R| = \hat{R}|s - s_0|$.*

THEOREM 4.7 (CONDITIONAL WEIGHTED- L_1 IMPROVEMENT). *Let $\pi = Dn/R$ and $\hat{\pi} = D\hat{n}/\hat{R}$. Then $\|D(s_0 \hat{n} - n)\|_1 = R\|\hat{\pi} - \pi\|_1$, while $\|D(\hat{n} - n)\|_1 \geq |R - \hat{R}|$. Hence unclipped calibration strictly reduces weighted L_1 error whenever $R\|\hat{\pi} - \pi\|_1 < |R - \hat{R}|$.*

This is a conditional statement: arbitrary shape errors can make scalar rescaling harmful. Appendix C.9 gives the proof, a count- L_1 multiplicative-bias condition, a counterexample, and the clipping interpretation. It also accounts for a target R_A different from the true decoded-support mass and distinguishes the weighted- L_1 objective from the WMRD objective.

5 Performance Evaluation

5.1 Experiment Setup

5.1.1 Test Platform. Our experiments were conducted on a hybrid platform that mirrors the deployment model of NeuFSD: lightweight packet insertion in the data plane and FSD decoding in the control plane. CPU-based operations, including the baseline sketches and the CPU version of NeuFSD inference, were executed on the same x86 server. GPU-accelerated NeuFSD training and inference were executed on NVIDIA H800 GPUs. Supplementary GPU experiments use an RTX 4090, with configurations specified in Appendix D. For decoding-time comparisons, we report both NeuFSD-GPU and NeuFSD-CPU, since the baselines run their EM decoders on CPU. All algorithms process the same 1-minute windows unless otherwise stated.

5.1.2 Hardware Testbed. To evaluate hardware deployability, we evaluate the *online insertion* of NeuFSD and the baselines on a Barefoot Tofino programmable switch. The P4 implementations are compiled with the Tofino toolchain, and we report the compiler-generated resource utilization, including the number of physical pipeline stages, SRAM/TCAM usage, hash-bit consumption, and the number of stateful ALU operations. Note that NeuFSD’s *neural decoding* runs in the control plane and is therefore not part of the switch data-plane resource accounting.

5.1.3 Datasets. We evaluate NeuFSD on four real-world traffic traces: **CAIDA 2016**, **CAIDA 2018** [4], **IMC** [2, 3], and **MAWI** [29]. The CAIDA traces were collected from high-speed backbone links, IMC contains data-center traffic, and MAWI is collected from a trans-Pacific backbone link. Each stored observation block contains 1-minute packet records. After discarding incomplete final chunks, CAIDA2016, CAIDA2018, IMC, and MAWI contain 304, 729, 98, and 114 complete blocks, respectively. For IMC and MAWI, we parse the 5-tuple key stream with the same 13-byte-per-packet format as the CAIDA preprocessing pipeline. The first five blocks supply the initial history. The comprehensive memory sweep uses the first ten measurable windows (indices 5–14), and the time series uses the first 100 measurable windows where available, or all 93 measurable IMC windows (5–97). Plot indices refer to physical elapsed time that depends on the packet timestamps.

5.1.4 Comparison Algorithms. Our NeuFSD, implemented in C++ for the data-plane logic and PyTorch for the control-plane decoder, is benchmarked against five representative approaches:

- **Elastic Sketch** [49]: A sketch capable of adaptive monitoring for hot flows and cold flows.
- **MRAC** [18]: The classic FSD estimation algorithm using Count-Min sketch and EM decoding.
- **DaVinci Sketch** [44]: A recent sketch that augments the MRAC-style EM pipeline with a hot flow part, a multi-layer Tower sketch, and a Fermat structure.
- **Array Sample**: A fixed-rate flow-level sampler backed by an array table.
- **Hash Sample**: The same fixed-rate flow-level sampler backed by a hash table.

Each algorithm is implemented faithfully according to the specifications in its original publication or its released implementation. Sampling uses a fixed 1% flow-level sampling rate: a sampled flow is kept entirely, an unsampled flow is dropped, and the restored FSD divides the sampled FSD by the sampling rate. If the sampling table is full, newly sampled flows are dropped.

5.1.5 NeuFSD configuration. Unless otherwise specified, NeuFSD uses a 64×64 array of 32-bit counters (16 KiB), $\lambda = 8$, $\phi = 100$, Avg-5 continuous SFT with 2,000 examples per update, and residual-mass calibration. Baselines are evaluated at nominal memory budgets from 16 KiB to 256 KiB, while NeuFSD keeps its counter array fixed at 16 KiB.

Resource accounting. The 16 KiB setting covers only the counter array; Hot Filter storage and control-plane resources are accounted for separately. The reference label builder counts all flows in

host memory before sampling flows to construct training labels. Appendices D and E report the detailed storage requirements and the costs of label preparation, adaptation, and decoding.

Accuracy inputs and scoring. For the trace-based accuracy experiments, we build counter arrays from all packets in each window. We combine the neural estimates for small flows with separately supplied heavy-flow counts. For IMC and MAWI, these counts are computed exactly from the trace. The bounded Hot Filter insertion prototypes are evaluated separately. NeuFSD predictions are clipped to be nonnegative and evaluated on flow-size bins present in the ground truth, up to size 10,000. Baseline results use their native union support and retain signed estimates. The comparisons therefore reflect these differences in inputs and scoring conventions. Appendix E also reports matched adaptation experiments and common-support diagnostics.

5.1.6 Tasks and Metrics. We evaluate the performance of algorithms using the following metrics:

1. Insertion Speed: This metric measures the online processing speed, calculated as N/T , where N is the total number of packets processed and T is the total insertion time. The unit is million packets per second (**Mpps**).

2. Decoding Latency: This measures the time required to recover the full FSD from the data structure after the measurement interval ends. For NeuFSD, this includes the neural network inference time. The unit is seconds (**s**).

3. Error Metrics: Let n_i be the actual number of flows of size i , and $\hat{n}_i$ be the estimated number. We use two metrics to quantify the distance between the estimated and actual distributions:

- **(Weighted) Mean Relative Difference (WMRD and MRD):** MRD averages relative errors across evaluated flow sizes, whereas WMRD normalizes the total absolute count error by the average total flow count (across all flow sizes $1, \dots, z$):

$$MRD = \frac{1}{z} \sum_i \frac{|n_i - \hat{n}_i|}{\left(\frac{n_i + \hat{n}_i}{2}\right)}, \quad WMRD = \frac{\sum_i |n_i - \hat{n}_i|}{\sum_i \left(\frac{n_i + \hat{n}_i}{2}\right)}$$

The sums are over the stated evaluation support, with zero-denominator bins excluded from MRD. For nonnegative counts, each symmetric bin error and each defined aggregate metric lies in $[0, 2]$. For MRD, the Zipfian tail contains low-population bins; their relative discrepancies can contribute substantially even when the absolute error is small. Signed baseline estimates are interpreted using their native scorer convention specified above. To address the limitations, we adopt WMRD as our primary accuracy metric. WMRD assigns a weight proportional to the population of flows at each size i , focusing the metric on the main body of the distribution.

4. Temporal Stability (Error vs. Time): To evaluate online robustness, we run each algorithm on consecutive 1-minute windows. For the history-based NeuFSD setting, the model follows the pre-train+continuous-SFT pipeline described in Section 3.4: the control plane updates the decoder using recent sampled windows and then applies the resulting model to the current sketch snapshot. The comprehensive IMC series uses the separately specified decoder configuration. The time-series plots report WMRD and MRD over the first 100 measurable windows; IMC contains 93 measurable windows, and its plots use the same x-axis range with an empty tail.

5.2 Parameter Sensitivity

We study two key parameters on the first 50 measurable 1-minute windows (indices 5–54) of all four traces: the Hot Filter replacement threshold λ and the hot-flow splice threshold ϕ . Smaller λ reacts faster to newly active hot flows, while larger λ avoids churn. The threshold ϕ controls where exact Hot Filter accounting ends and neural residual decoding begins.

Figure 9 shows the representative MAWI result; the full four-trace figure is in Appendix B. NeuFSD is not highly sensitive to λ : mean WMRD over the four traces changes only from 0.1117 at $\lambda = 2$ to 0.1119 at $\lambda = 32$, and MRD is similarly stable. We use $\lambda = 8$, a conservative point in this plateau.

Figure 10 shows the representative ϕ sweep. The sweep varies the splice of cached neural and heavy-flow predictions: a smaller ϕ uses the heavy branch over more sizes, while a larger ϕ uses a longer neural tail. Across all traces, $\phi = 100$ gives the best WMRD and MRD, so we use $\lambda = 8$ and $\phi = 100$ by default.

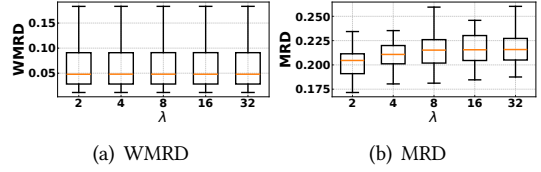


Fig. 9. λ sensitivity on MAWI.

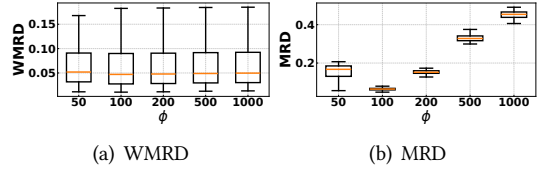


Fig. 10. ϕ sensitivity on MAWI.

5.3 Decoder Architecture Ablation

We next compare the decoder configurations. *Final* is the configured neural prediction with heavy-flow splicing and correction. *No Hot* removes the Hot Filter from cached *Final* predictions. *Origin+ViT* keeps the ViT but uses only the unsorted counter order. *Origin+CNN* and *Origin+MLP* use the same origin-only input with different decoders.

Figure 11 shows the representative MAWI result; Appendix B reports all traces. Averaged over the four traces, *Final* achieves WMRD 0.0898 and MRD 0.1805, improving over *Origin+ViT* (0.1108, 0.1833), *Origin+CNN* (0.1730, 0.2062), *Origin+MLP* (0.8380, 0.4529), and *No Hot* (0.1225, 0.5570).

The *No Hot* variant is especially poor on MRD. This is expected because the Hot Filter is designed precisely for the sparse hot flow region. WMRD is dominated by bins containing many flows, typically the cold flow body of the FSD, so removing the hot component only moderately degrades WMRD on some traces. MRD, however, averages relative error across flow sizes and is therefore much more sensitive to the tail: missing or misplacing a small number of hot flows can create a large relative error even when the absolute count error is small. The large MRD gap between *Final* and *No Hot* shows the contribution of the Hot Filter.

The gap between *Final* and *Origin+ViT* highlights the value of the three-view counter representation. The unsorted view preserves the actual collision pattern induced by the hash table, but it is tied to arbitrary hash indices. The ascending and descending views remove the arbitrary index order and expose the empirical counter-value distribution, which is closely related to the underlying FSD. The descending view further emphasizes large counters at a consistent spatial location, while the ascending view makes the cold-flow body prominent from the opposite end of the representation. Stacking these views gives the ViT both hash-dependent collision information and counter-order-independent distributional information. *Origin+ViT* lacks two of these views

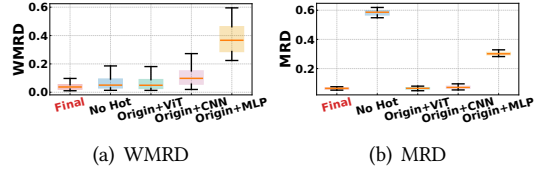


Fig. 11. Decoder ablation on MAWI.

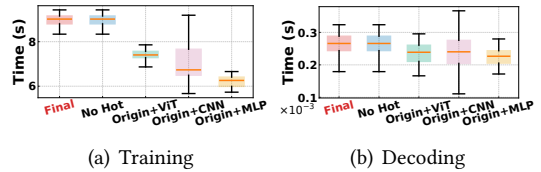


Fig. 12. Decoder cost on MAWI.

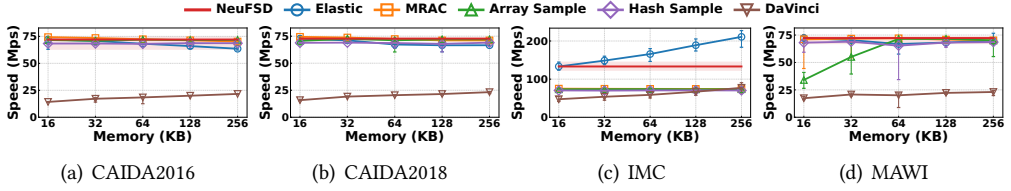


Fig. 13. Insertion speed vs. memory.

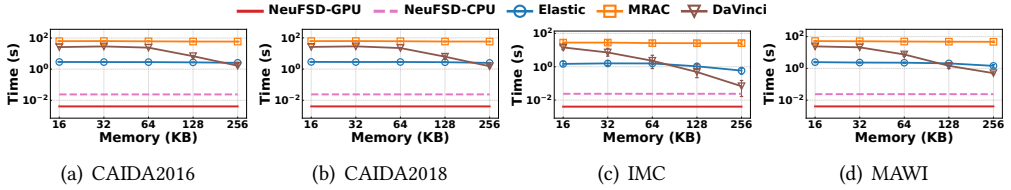


Fig. 14. Decoding time vs. memory.

and has higher recorded error, especially on CAIDA2018, IMC, and MAWI. The compared configurations also differ in initialization and correction settings; this comparison does not isolate the input representation alone.

The CNN and MLP variants complement our design motivation for a ViT: hashed counters have weak local image semantics, whereas attention can model non-local counter-value patterns. The recorded comparison concerns these configured models rather than an equal-parameter or equal-initialization architecture study.

Figure 12 reports the corresponding computational cost. Final takes about 8.4 seconds per online SFT update on average, and inference is roughly $2.2\text{--}2.7 \times 10^{-4}$ seconds per sketch when amortized over a batch of 400 across variants. The observation windows contain 1-minute packets. Sketch latency and the separate preparation, sampling, and adaptation scopes are reported in Appendix E.

5.4 End-to-End Comparison

5.4.1 Evaluation Protocol. We now compare NeuFSD with the full set of baselines on the four traces, using the flow-key representations, memory accounting, and scoring conventions specified in the experimental setup. The memory-sweep experiments use measurable windows 5–14 of each trace. NeuFSD is fixed at its 16KB 64×64 array configuration, while the traditional methods are swept from 16KB to 256KB. This is the intended operating point for NeuFSD: the data plane stays small and fixed, and the control plane gains accuracy from learned decoding rather than from larger arrays. The time-series experiments fix all traditional methods at 16KB and evaluate over the first 100 measurable windows. This protocol separates the effect of additional baseline memory from temporal robustness under a fixed small-memory operating point.

5.4.2 Insertion Speed. Figure 13 reports baseline insertion rates and uses Elastic Sketch’s 16 KiB measurement as the NeuFSD insertion proxy. NeuFSD’s insertion path consists of a small Hot Filter lookup/update and one counter-array update; its direct insertion measurements are reported in Sections 5.6 and 5.7. DaVinci is consistently slower than the plotted proxy. This is not surprising: DaVinci combines a hot flow part, a multi-layer Tower sketch, and a Fermat structure. A packet may trigger multiple hash computations, counter updates across layers, swaps from the hot flow part, and overflow insertion into the Fermat component. These extra branches and memory accesses make the update path much heavier than a single-array sketch. Increasing DaVinci’s memory improves its insertion speed in several traces because fewer flows overflow into the expensive auxiliary structures, but it still remains slower than the plotted proxy.

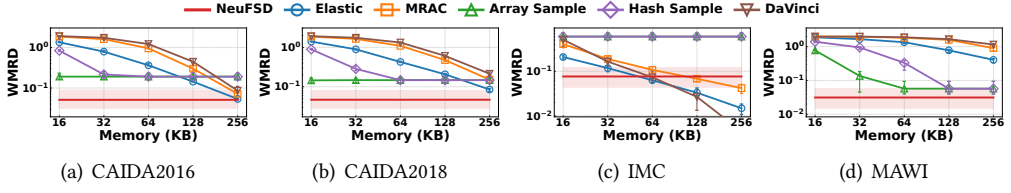


Fig. 15. WMRD vs. memory.

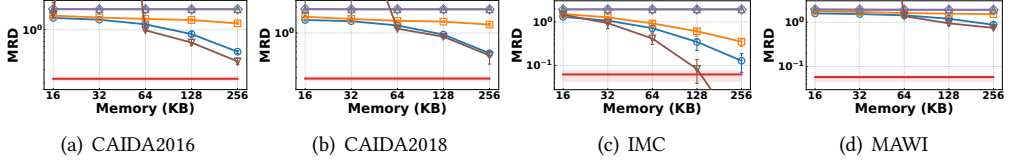


Fig. 16. MRD vs. memory.

Elastic Sketch exhibits an interesting memory-dependent behavior. On traces such as IMC, its insertion speed increases substantially with memory. A larger hot flow part captures more hot flows directly, so many packets are resolved by a hot-part hit and avoid cold-part updates and evictions. In this regime, extra memory improves not only accuracy but also update efficiency. On traces where the hot working set already fits reasonably well, the speed change is smaller. This observation also explains why NeuFSD uses a lightweight Hot Filter: separating hot flows is useful, but the insertion logic must remain shallow enough for high-speed data planes.

5.4.3 Decoding Latency. Figure 14 reports baseline decoding times together with a reference neural forward microbenchmark. The approximately 4.04 ms H800 and 23.84 ms CPU values use randomly initialized reference networks with 10/9,990 outputs and prepared random inputs. The accuracy experiments use the separately specified 10/1,080-output CustomViT pair; its measured GPU forward and CPU query costs are reported above and in Appendix E. MRAC requires tens of seconds: even the fastest MRAC point is about 25.9 seconds on IMC, and CAIDA/MAWI are often around 50–62 seconds. Elastic Sketch is faster than MRAC because its hot flow part removes some traffic from the cold flow EM problem, but it still needs hundreds of milliseconds to several seconds. DaVinci can be fast when memory is large and the Tower/Fermat structures are lightly stressed, but at small memory its 8-bit Tower EM and overflow handling can take more than 20 seconds. The actual-model measurements likewise demonstrate millisecond-scale neural computation, with preparation, adaptation, and auxiliary-state costs reported separately.

5.4.4 Accuracy Under Memory Scaling. Figures 15 and 16 show that NeuFSD’s 16KB configuration is already highly competitive with, and usually better than, much larger traditional sketches. At 16KB, NeuFSD obtains WMRD 0.051 on CAIDA2016, 0.047 on CAIDA2018, 0.077 on IMC, and 0.031 on MAWI. The best 16KB baseline WMRD is 0.191, 0.145, 0.200, and 0.774 on these traces, respectively. The MRD gap is even larger: NeuFSD achieves 0.227, 0.295, 0.062, and 0.057, while the best 16KB baseline MRD is at least 1.30 on every trace. This highlights the contribution of the separately supplied heavy-flow counts, which are exact in the IMC/MAWI accuracy inputs: hot-flow bins are sparse but important for MRD, and explicitly accounting for them avoids reconstructing the tail solely from noisy counters.

Sampling has a characteristic asymmetric behavior. Array Sample and Hash Sample can achieve reasonable WMRD on some traces, because WMRD is dominated by the cold flow body where many small flows exist and a scaled sample can approximate the aggregate shape. However, their MRD is consistently poor, often close to 1.9. The reason is that low-rate flow-level sampling rarely captures enough hot flow bins. Missing a sparse hot flow bin contributes little to WMRD but creates a large

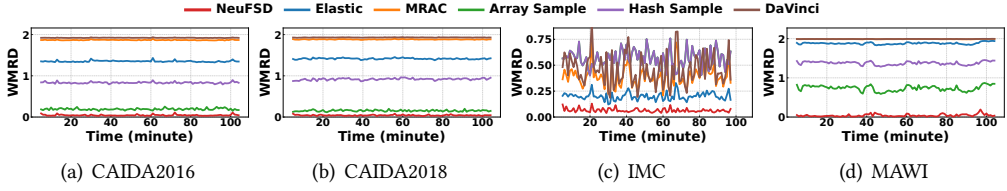


Fig. 17. WMRD over time.

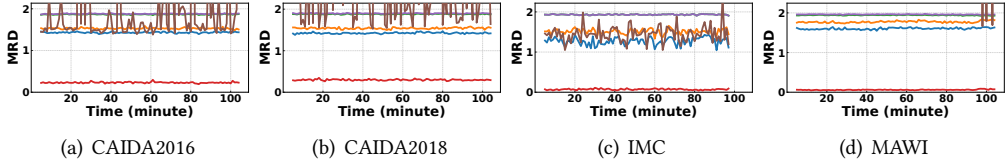


Fig. 18. MRD over time.

relative error in MRD. Hash Sample can also be worse than Array Sample at small memory because the hash table stores fewer sampled flows once collision and metadata overhead are considered; after the table is full, new sampled flows are dropped.

DaVinci shows another failure mode. At small memory, its WMRD is often close to the maximum-error region, and its MRD can explode by several orders of magnitude. The native signed-score convention can produce MRD outside the nonnegative symmetric metric's range; capped counters and overflow recovery also affect the distribution supplied to that scorer. When the Tower layer saturates or the Fermat component cannot cleanly recover overflowed flows, a small number of hot flow mistakes can dominate MRD. Increasing memory reduces this pressure and can make DaVinci accurate on IMC at 256KB, but this comes with a much larger memory budget and still does not remove the EM-style decoding cost. NeuFSD instead uses the neural decoder for the cold flow body and the supplied heavy-flow counts for the tail, avoiding this unstable failure mode at 16KB.

5.4.5 Temporal Stability. Figures 17 and 18 evaluate the setting that matters most for online measurement: a fixed 16KB data-plane structure running over time. NeuFSD remains stable across all traces. On CAIDA2016, CAIDA2018, IMC, and MAWI, the average WMRD is 0.044, 0.042, 0.064, and 0.041, respectively. The best 16KB baseline WMRD on the same windows is 0.192, 0.154, 0.200, and 0.735. MRD shows an even larger separation: NeuFSD is lower than the best 16KB baseline by 6.2 $\times$, 4.8 $\times$, 17.8 $\times$, and 26.4 $\times$ on the four traces.

The IMC result is particularly informative. IMC has stronger window-to-window distribution shifts than the backbone traces. The final IMC series uses fixed input views, output coordinates, and common scoring rules for each snapshot; its decoder combines the predicted small-flow counts with a separate heavy-flow branch. Its observed trajectory characterizes this decoder implementation. History-based continuous adaptation is evaluated separately in Section 3.5 and in the matched supplementary replay in Appendix E. MRAC and Elastic Sketch rely on the current sketch and EM, sampling is limited by which flows happen to be sampled, and DaVinci can be affected by overflow recovery under tight memory.

Overall, the results support the central design claim of NeuFSD. Traditional sketches spend little time inserting packets but pay a large and unstable decoding cost; sampling is simple and sometimes adequate for WMRD but loses the hot flow tail; DaVinci improves some high-memory regimes but can become slow and unstable under tight memory. NeuFSD occupies a different point in the design space: it keeps the data plane compact, uses online samples to adapt the model and, in the indicated variants, to correct the sampled shape, and replaces iterative reconstruction with a

millisecond-scale learned decoder. The result is consistently low WMRD and MRD at 16KB, stable behavior over time, and decoding latency that is orders of magnitude lower than EM-based methods. These advantages are observed across both backbone and data-center traces, rather than being driven by a single workload.

5.5 Evaluation on PISA Hardware

In this section, we evaluate the hardware deployability of NeuFSD on a PISA programmable switch. Following common practice, we focus on the *data-plane* insertion logic, and use the Tofino compiler report to quantify resource utilization.

5.5.1 Hardware Implementation. NeuFSD is designed to match the feed-forward multi-stage pipeline of PISA switches and to avoid complex recirculation or control dependencies across stages. We implement the Hot Filter and the main counter array as register arrays in SRAM. Packet processing consists of (i) hashing the flow label to locate the target bucket, (ii) performing a small number of key comparisons and conditional updates within the bucket (including the vote update and potential eviction), and (iii) updating the counter array for cold packets and for replayed evictions. These operations map naturally to a small number of pipeline stages using stateful ALUs for read-modify-write updates under simple conditions. The resulting per-packet update path remains fixed and independent of the complexity of the subsequent FSD decoding, making the design well suited to deterministic line-rate execution.

Importantly, NeuFSD keeps the switch data plane lightweight: only constant-time per-packet updates are executed on the switch, while the expensive FSD decoding is performed in the control plane via neural inference. For a fixed sketch layout, updating the decoder's learned weights does not change the register organization or the per-packet update sequence. This separation allows the decoder to adapt to recent traffic while the switch retains the same insertion logic.

5.5.2 Resource Overhead. Table 1 summarizes the hardware resource consumption of NeuFSD and representative baselines on Tofino. NeuFSD requires only 4 pipeline stages, substantially fewer than Elastic Sketch (12 stages), sampling (9 stages), and the DaVinci (16 stages), while remaining close to MRAC (1 stage). NeuFSD also consumes moderate SRAM (4.27%), reflecting the Hot Filter plus a compact counter array, and it uses no TCAM. Importantly, NeuFSD requires significantly fewer stateful ALUs (3.13%) than Elastic Sketch (9.38%) and DaVinci (12.50%), indicating that NeuFSD avoids complex multi-stage hot flow/cold flow coordination and instead relies on a compact Hot Filter and a single counter array with simple update rules. NeuFSD also reduces hash-bit consumption compared with Elastic Sketch, sampling, and DaVinci, which helps preserve hashing budget for co-located switch functions. The main additional cost over the minimal MRAC design comes from the Hot Filter, while remaining well below the deeper multi-structure baselines.

Overall, these results show that NeuFSD fits comfortably within the resource budget of a commodity programmable switch, leaving ample headroom for other packet-processing logic and measurement tasks. This confirms that moving reconstruction to learned control-plane decoding does not require shifting excessive complexity back into the switch pipeline.

5.5.3 Per-packet Data-plane Latency. Beyond resource usage, it is crucial that a measurement primitive does not introduce excessive processing latency in the switch pipeline. We therefore measure the per-packet processing latency (in nanoseconds) for each design when deployed on Tofino. The last column in Table 1 reports the latency overhead of the data-plane insertion logic.

NeuFSD incurs only 297.39 ns per packet, which is close to MRAC (260.39 ns) and notably lower than Elastic Sketch (399.33 ns), sampling (376.92 ns), and the DaVinci (458.83 ns). This result is consistent with our design goal: NeuFSD uses a small number of pipeline stages and modest

stateful operations, avoiding the deeper pipelines and heavier update logic required by Elastic Sketch and DaVinci. DaVinci's latency reflects a longer insertion path: it combines a hot flow part, multiple Tower layers, and Fermat structures, so its resource model assigns more stateful stages and hash-dependent updates. Taken together with Table 1, NeuFSD achieves a favorable hardware-efficiency profile: low resource footprint and low per-packet latency, while delegating full-distribution decoding to the control plane.

Table 1. Hardware resource consumption and per-packet processing latency on Barefoot Tofino.

Sketch	Stages	SRAM (%)	TCAM (%)	Hash Bits (%)	Stateful ALU (%)	Latency (ns)
MRAC	1	3.44	0	0.34	1.04	260.39
Elastic	12	6.77	0	2.42	9.38	399.33
Sample	9	3.54	0	2.08	9.38	376.92
DaVinci	16	8.95	0	4.84	12.50	458.83
NeuFSD	4	4.27	0	1.26	3.13	297.39

5.6 Deployment on Redis

We further evaluate whether NeuFSD can be embedded into a widely used in-memory serving system. Redis is an in-memory data-structure server that exposes operations through a command interface [34]. It also supports loadable modules, as illustrated by production modules such as RedisBloom [35]. We implement the five measurement algorithms as Redis module objects. A sketch instance is bound to a Redis key, and the module provides commands to create a sketch, insert a batch of flow keys, query items, reset state, and return metadata. The experiment measures only the online insertion path: a client sends batched `dp.insert` commands, Redis performs command parsing and object lookup, and the module calls the corresponding sketch runtime's `Insert()`.

Figure 19(a) reports the Redis module insertion throughput. Each bar shows the median of 10 repeated runs for a fixed algorithm and memory budget, and the error bar spans the minimum and maximum. Redis adds command parsing, key lookup, and module dispatch overhead, so the absolute throughput is lower than a standalone C++ microbenchmark. Nevertheless, all methods sustain hundreds of thousands of insertions per second. NeuFSD remains in the same range as MRAC and Elastic Sketch, with median throughput around 0.60–0.69 million events/s across the memory sweep. These rates characterize measurement through the Redis command interface, including the work needed to locate and update a server-side sketch object. They therefore indicate the throughput available when sketch insertion is exposed as a service operation. This confirms that NeuFSD's data-plane update is not the bottleneck in a key-value serving environment; the dominant cost is the Redis execution path shared by all module commands. Sampling is the fastest in some configurations because its retained-flow update is structurally simpler, while DaVinci is consistently slower due to its hot-flow, Tower, and Fermat update logic.

5.7 Deployment on BESS

We also deploy the same runtime interface inside BESS, a programmable software switch framework [31]. BESS organizes packet processing as a graph of C++ modules executed by worker threads. We implement a `DataplaneTracker` module that receives a packet batch, checks Ethernet/IPv4 headers, extracts the IPv4 source address as the flow key, calls `Insert()`, and forwards the batch to the next module. This deployment evaluates NeuFSD in a realistic software data-plane execution model, where measurement logic shares the packet-processing path with parsing and forwarding operations. The experiment uses a live BESS pipeline

FlowGen → DataplaneTracker → Sink,

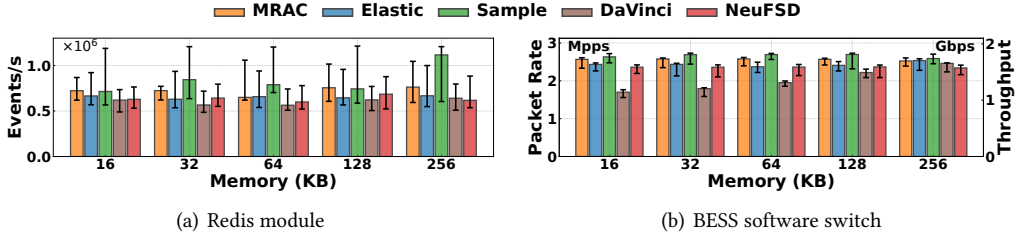


Fig. 19. Deployment throughput on Redis and BESS.

where FlowGen produces 64B IPv4/UDP packets and Sink discards packets after measurement. This setup measures the packet-processing cost inside the BESS dataplane graph without requiring a physical NIC. The right axis in Figure 19(b) converts packet rate to wire throughput using 64B packets plus 20B Ethernet L1 overhead.

Figure 19(b) shows that NeuFSD processes about 2.34–2.37 Mpps in the BESS software pipeline, corresponding to roughly 1.57–1.59 Gbps under the L1 throughput convention. The narrow throughput range across the tested memory settings suggests that NeuFSD’s insertion rate is relatively insensitive to the configured memory budget in this pipeline. Its throughput is close to Elastic Sketch and within the same regime as MRAC, while using the neural decoder only in the control plane. Sample is faster because its insertion path is the simplest. DaVinci improves with memory, from about 1.70 Mpps at 16KB to 2.46 Mpps at 256KB, but this trend reflects that larger memory reduces pressure on its expensive auxiliary structures. The two software deployments exercise different entry points to the same update interface: Redis receives batches of flow keys through commands, whereas BESS extracts a key from each generated packet. Together, they demonstrate how the sketch runtime can be integrated into both a serving system and a software-switch pipeline. Overall, these Redis and BESS deployments support the same conclusion as the Tofino evaluation: NeuFSD keeps online insertion lightweight enough for practical data-plane integration, while moving the expensive FSD reconstruction out of the packet-processing path.

6 Conclusion

In this paper, we introduced NeuFSD, a framework that fundamentally rethinks the problem of Flow Size Distribution estimation. Moving beyond the constraints of traditional statistical inference, we pioneered the use of neural decoding, treating the extraction of flow information from sketches as a learned pattern recognition task. By combining a hybrid data plane (Hot Filter + Counter Array) with a novel Vision Transformer-based decoder, NeuFSD captures complex collision patterns that traditional Expectation-Maximization algorithms struggle to resolve efficiently. Our extensive evaluation on real-world traces demonstrates that NeuFSD improves the long-standing trade-off between accuracy and speed. It reports estimation errors up to 2 orders of magnitude lower than state-of-the-art sketches under the stated scoring conventions while delivering millisecond-scale neural decoding in the control plane. This work not only provides a practical solution for real-time, high-precision traffic analysis but also suggests a promising new direction: utilizing data-driven neural computing to modernize and accelerate classic network measurement algorithms.

Acknowledgments

We thank our shepherd and anonymous SIGMETRICS reviewers for their valuable comments. The research is supported by the National Natural Science Foundation of China (Grant 62132004, 62472246, 62221003), the National Key R&D Program of China (No. 2025YFE0201000). Yuanyuan Zhang, Yangyang Wang, and Mingwei Xu are the corresponding authors.

References

- [1] Ran Ben-Basat, Gil Einziger, Roy Friedman, and Yaron Kassner. 2016. Heavy hitters in streams and sliding windows. in *IEEE INFOCOM* (2016).
- [2] Theophilus Benson. 2010. Data Set for IMC 2010 Data Center Measurement. https://pages.cs.wisc.edu/~tbenson/IMC10_Data.html.
- [3] Theophilus Benson, Aditya Akella, and David A Maltz. 2010. Network traffic characteristics of data centers in the wild. in *ACM IMC* (2010).
- [4] CAIDA. 2024. The CAIDA Traces. <http://www.caida.org/data/overview/>.
- [5] Moses Charikar, Kevin Chen, and Martin Farach-Colton. 2002. Finding frequent items in data streams. in *Springer ICALP* (2002).
- [6] Peiqing Chen, Yuhan Wu, Tong Yang, Junchen Jiang, and Zaoxing Liu. 2021. Precise error estimation for sketch-based flow measurement. In *Proceedings of the 21st ACM Internet Measurement Conference*. 113–121.
- [7] Kimberly C Claffy. 1995. Internet traffic characterization. (1995).
- [8] Graham Cormode. 2011. Sketch techniques for approximate query processing. *Foundations and Trends in Databases. NOW publishers* (2011).
- [9] Graham Cormode, Minos Garofalakis, Peter J Haas, and Chris Jermaine. 2012. Synopses for massive data: Samples, histograms, wavelets, sketches. *Foundations and Trends in Databases* 4, 1–3 (2012), 1–294.
- [10] Graham Cormode and Marios Hadjieleftheriou. 2008. Finding frequent items in data streams. *Proceedings of the VLDB Endowment* 1, 2 (2008), 1530–1541.
- [11] Graham Cormode and Shan Muthukrishnan. 2005. An improved data stream summary: the count-min sketch and its applications. *Journal of Algorithms* 55, 1 (2005), 58–75.
- [12] Erik D Demaine, Alejandro López-Ortiz, and J Ian Munro. 2002. Frequency estimation of internet packet streams with limited space. in *Springer ESA* (2002).
- [13] Allen B. Downey. 2001. Evidence for long-tailed distributions in the Internet. In *Proceedings of the 1st ACM SIGCOMM Workshop on Internet Measurement*. 229–241. <https://doi.org/10.1145/505202.505230>
- [14] Michael T Goodrich and Michael Mitzenmacher. 2011. Invertible bloom lookup tables. in *IEEE Allerton* (2011).
- [15] Qun Huang, Xin Jin, Patrick PC Lee, Runhui Li, Lu Tang, Yi-Chao Chen, and Gong Zhang. 2017. Sketchvisor: Robust network measurement for software packet processing. in *ACM SIGCOMM* (2017).
- [16] Qun Huang and Patrick PC Lee. 2014. Ld-sketch: A distributed sketching design for accurate and scalable anomaly detection in network data streams. *IEEE INFOCOM* (2014).
- [17] Qun Huang, Patrick PC Lee, and Yungang Bao. 2018. Sketchlearn: relieving user burdens in approximate measurement with automated statistical inference. in *ACM SIGCOMM* (2018).
- [18] Abhishek Kumar, Minh Sung, Jun Xu, and Jia Wang. 2004. Data Streaming Algorithms for Efficient and Accurate Estimation of Flow Size Distribution. in *ACM SIGMETRICS* (2004).
- [19] Haoyu Li, Qizhi Chen, Yixin Zhang, Tong Yang, and Bin Cui. 2022. Stingy sketch: a sketch framework for accurate and fast frequency estimation. in *VLDB* 15, 7 (2022).
- [20] Jizhou Li, Zikun Li, Yifei Xu, Shiqi Jiang, Tong Yang, Bin Cui, Yafei Dai, and Gong Zhang. 2020. WavingSketch: An Unbiased and Generic Sketch for Finding Top-k Items in Data Streams. in *ACM SIGKDD* (2020).
- [21] Tao Li, Shigang Chen, and Yibei Ling. 2012. Per-flow traffic measurement through randomized counter sharing. *IEEE/ACM Transactions on Networking* 20, 5 (2012), 1622–1634.
- [22] Weihe Li and Paul Patras. 2024. Stable-Sketch: A Versatile Sketch for Accurate, Fast, Web-Scale Data Stream Processing. in *ACM WWW* (2024).
- [23] Yuliang Li, Rui Miao, Changhoon Kim, and Minlan Yu. 2016. Flowradar: A better netflow for data centers. in *USENIX NSDI* (2016).
- [24] Zaoxing Liu, Ran Ben-Basat, Gil Einziger, Yaron Kassner, Vladimir Braverman, Roy Friedman, and Vyas Sekar. 2019. Nitrosketch: Robust and general sketch-based monitoring in software switches. in *ACM SIGCOMM* (2019).
- [25] Zaoxing Liu, Antonis Manousis, Gregory Vorsanger, Vyas Sekar, and Vladimir Braverman. 2016. One sketch to rule them all: Rethinking network flow monitoring with univmon. in *ACM SIGCOMM* (2016).
- [26] Patrick Loiseau, Paulo Gonçalves, Stéphane Girard, Florence Forbes, and Pascale Vicat-Blanc Primet. 2009. Maximum likelihood estimation of the flow size distribution tail index from sampled packet data. In *Proceedings of ACM SIGMETRICS*. 263–274. <https://doi.org/10.1145/1555349.1555380>
- [27] Yi Lu, Andrea Montanari, Balaji Prabhakar, Sarang Dharmapurikar, and Abdul Kabbani. 2008. Counter braids: A novel counter architecture for per-flow measurement. *ACM SIGMETRICS Performance Evaluation Review* 36, 1 (2008), 121–132.
- [28] Gurmeet Singh Manku and Rajeev Motwani. 2002. Approximate frequency counts over data streams. in *ACM VLDB* (2002).
- [29] MAWI Working Group. 2024. MAWI Working Group Traffic Archive. <https://mawi.wide.ad.jp/mawi/>.

- [30] Ahmed Metwally, Divyakant Agrawal, and Amr El Abbadi. 2005. Efficient computation of frequent and top-k elements in data streams. in *Springer ICDT* (2005).
- [31] NetSys Lab. 2022. BESS Website. Accessed: 2022. [Online]. Available: <https://github.com/NetSys/bess>.
- [32] NeuFSD Authors. 2026. Our Open Source GitHub. <https://github.com/stallone0000/NeuFSD>.
- [33] Jiuhua Qi, Wenjun Li, Tong Yang, Dagang Li, and Hui Li. 2019. Cuckoo counter: A novel framework for accurate per-flow frequency estimation in network measurement. in *ACM/IEEE ANCS* (2019).
- [34] Redis. 2022. Redis Website. Accessed: 2022. [Online]. Available: <https://redis.io>.
- [35] RedisBloom. 2022. RedisBloom. Accessed: 2022. [Online]. Available: <https://github.com/RedisBloom/RedisBloom>.
- [36] Pratanu Roy, Arijit Khan, and Gustavo Alonso. 2016. Augmented sketch: Faster and more accurate stream processing. in *ACM SIGMOD* (2016).
- [37] Robert Schweller, Zhichun Li, Yan Chen, Yan Gao, Ashish Gupta, Yin Zhang, Peter A Dinda, Ming-Yang Kao, and Gokhan Memik. 2007. Reversible sketches: enabling monitoring and analysis over high-speed data streams. *IEEE/ACM Transactions on Networking* 15, 5 (2007), 1059–1072.
- [38] Qilong Shi, Xirui Li, Hanyue Zheng, Tong Yang, Yangyang Wang, and Mingwei Xu. 2025. HeavyLocker: Lock Heavy Hitters in Distributed Data Streams. in *ACM SIGKDD* (2025).
- [39] Qilong Shi, Yuchen Xu, Jiuhua Qi, Wenjun Li, Tong Yang, Yang Xu, and Yi Wang. 2023. Cuckoo Counter: Adaptive Structure of Counters for Accurate Frequency and Top-k Estimation. *IEEE/ACM Transactions on Networking* (2023).
- [40] Vibhaalakshmi Sivaraman, Srinivas Narayana, Ori Rottenstreich, Shan Muthukrishnan, and Jennifer Rexford. 2017. Heavy-hitter detection entirely in the data plane. in *ACM SOSR* (2017).
- [41] Cha Hwan Song, Pravein Govindan Kannan, Bryan Kian Hsiang Low, and Mun Choon Chan. 2020. FCM-Sketch: Generic Network Measurements with Data Plane Support. (2020), 78–92.
- [42] Lu Tang, Qun Huang, and Patrick PC Lee. 2019. Mv-sketch: A fast and compact invertible sketch for heavy flow detection in network data streams. *IEEE INFOCOM* (2019).
- [43] Lu Tang, Qun Huang, and Patrick PC Lee. 2020. Spreadsketch: Toward invertible and network-wide detection of superspreaders. *IEEE INFOCOM* (2020).
- [44] Yanshu Wang, Jianan Ji, Chao-Hsuan Liu, Hengyang Zhou, and Tong Yang. 2025. DaVinci Sketch: A Versatile Sketch for Efficient and Comprehensive Set Measurements. (2025), 210–223.
- [45] Yuhan Wu, Shiqi Jiang, Yifei Xu, Siyuan Dong, Kaicheng Yang, Peiqing Chen, and Tong Yang. 2023. Unbiased Real-time Traffic Sketching. *IEEE Transactions on Network Science and Engineering* (2023), 1–13. <https://doi.org/10.1109/TNSE.2023.3284004>
- [46] Yuhan Wu, Aomufei Yuan, Zhouan Shi, Yuanpeng Li, Yikai Zhao, Peiqing Chen, Tong Yang, and Bin Cui. 2024. Online Detection of Outstanding Quantiles with QuantileFilter. In *2024 IEEE 40th International Conference on Data Engineering (ICDE)*. IEEE, 831–844.
- [47] Kaicheng Yang, Sheng Long, Qilong Shi, Yuanpeng Li, Zirui Liu, Yuhan Wu, Tong Yang, and Zhengyi Jia. 2023. SketchINT: Empowering INT with TowerSketch for Per-Flow Per-Switch Measurement. *IEEE Transactions on Parallel and Distributed Systems* (2023).
- [48] Tong Yang, Junzhi Gong, Haowei Zhang, Lei Zou, Lei Shi, and Xiaoming Li. 2018. HeavyGuardian: Separate and guard hot items in data streams. in *ACM SIGKDD* (2018).
- [49] Tong Yang, Jie Jiang, Peng Liu, Qun Huang, Junzhi Gong, Yang Zhou, Rui Miao, Xiaoming Li, and Steve Uhlig. 2018. Elastic Sketch: Adaptive and Fast Network-Wide Measurements. in *ACM SIGCOMM* (2018).
- [50] Tong Yang, Lingtong Liu, Yibo Yan, Muhammad Shahzad, Yulong Shen, Xiaoming Li, Bin Cui, and Gaogang Xie. 2017. Sf-sketch: A fast, accurate, and memory efficient data structure to store frequencies of data items. in *IEEE ICDE* (2017).
- [51] Tong Yang, Haowei Zhang, Jinyang Li, Junzhi Gong, Steve Uhlig, Shigang Chen, and Xiaoming Li. 2019. HeavyKeeper: An Accurate Algorithm for Finding Top-k Elephant Flows. *IEEE/ACM Transactions on Networking* 27, 5 (2019), 1845–1858.
- [52] Tong Yang, Yang Zhou, Hao Jin, Shigang Chen, and Xiaoming Li. 2017. Pyramid sketch: A sketch framework for frequency estimation of data streams. *Proceedings of the VLDB Endowment* 10, 11 (2017), 1442–1453.
- [53] Da Yu, Yibo Zhu, Behnaz Arzani, Rodrigo Fonseca, Tianrong Zhang, Karl Deng, and Lihua Yuan. 2019. dShark: A general, easy to program and scalable framework for analyzing in-network packet traces. in *USENIX NSDI* (2019).
- [54] Fuheng Zhao, Punnaal Ismail Khan, Divyakant Agrawal, Amr El Abbadi, Arpit Gupta, and Zaoxing Liu. 2023. Panakos: Chasing the Tails for Multidimensional Data Streams. in *ACM VLDB* (2023).
- [55] Guangxiang Zhao, Wenkai Yang, Xuancheng Ren, Lei Li, Yunfang Wu, and Xu Sun. 2022. Well-classified examples are underestimated in classification with deep neural networks. in *AAAI* (2022).
- [56] Yang Zhou, Peng Liu, Hao Jin, Tong Yang, Shoujiang Dang, and Xiaoming Li. 2017. One memory access sketch: a more accurate and faster sketch for per-flow measurement. in *IEEE GLOBECOM* (2017).
- [57] Yang Zhou, Tong Yang, Jie Jiang, Bin Cui, Minlan Yu, Xiaoming Li, and Steve Uhlig. 2018. Cold filter: A meta-framework for faster and more accurate stream processing. in *ACM SIGMOD* (2018).

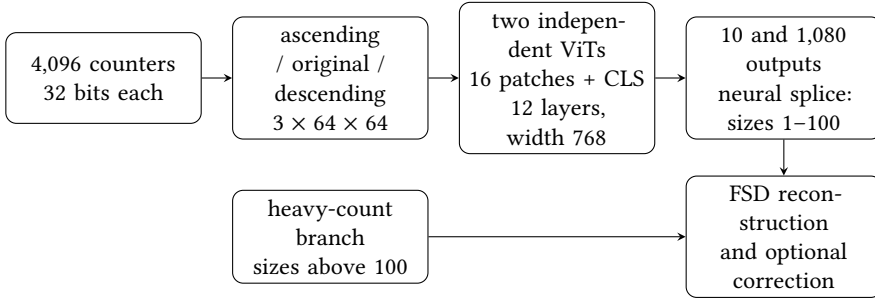


Fig. 20. Detailed decoder configuration: two independent ViTs process the same three counter views, with the larger-size contribution supplied by the heavy-flow branch.

A Concrete Decoder Configuration

For the 64×64 configuration, the released default converts 4,096 counters to raw float32 values and stacks ascending, original, and descending views into a $3 \times 64 \times 64$ tensor. There is no input normalization in this default; optional normalization modes are documented separately. A learned 16×16 patch projection produces 16 tokens of width 768. A learned classification token and absolute positional embeddings give 17 tokens in total. Each of the 12 pre-LayerNorm encoder blocks has 12 attention heads and a feed-forward layer of width 3,072 with GELU activation. The final normalized classification token feeds a regression head: LayerNorm, a $768 \rightarrow 1000$ linear layer, GELU, dropout 0.1, and a final linear layer. Encoder dropout is zero, and the outputs have no activation. Figure 20 summarizes the complete decoder architecture and the interaction between the two neural branches.

B Supplementary Results for Parameter Sensitivity and Decoder Ablation

Replacement threshold λ . Figure 21 reports the threshold sweep over the 50 1-minute windows indexed 5–54. The sweep reruns the heavy-flow processor for each λ while reusing the recorded neural predictions; it does not reconstruct and retrain the complete bounded-filter pipeline for each setting. Across CAIDA 2016, CAIDA 2018, IMC, and MAWI, both WMRD and MRD are relatively flat over this range. In the Hot Filter design, small λ values permit earlier replacement, while large values increase the sufficient survival margin and can delay admission. We select $\lambda = 8$ within the observed plateau rather than relying on a trace-specific minimum.

Hot-flow splice threshold ϕ . Figure 22 varies the boundary between the supplied heavy-flow counts and neural residual predictions over windows 5–54. This sweep changes the splice threshold of cached predictions and fixed heavy-flow inputs; it does not retrain the decoder or vary the insertion structure. For IMC and MAWI, the heavy-flow inputs contain exact counts only for flows of size at least 100, so the $\phi = 50$ point does not supply heavy-flow entries for sizes 51–99. Larger ϕ assigns a longer portion of the tail to neural decoding. The four-trace average favors $\phi = 100$, whose residual support is covered directly by the decoder output coordinates.

Decoder architecture. Figure 23 compares decoder configurations over windows 5–54. Final includes calibration; No Hot reuses Final’s recorded neural predictions and removes the heavy-flow splice. Origin+ViT, Origin+CNN, and Origin+MLP use the unsorted input alone and are evaluated without Final’s calibration. The ViT variants use prior initialization, whereas the CNN and MLP do not load that prior; parameter counts also differ. These are therefore configuration comparisons. No Hot’s higher MRD shows the contribution of the heavy-flow splice in these records. Origin+ViT

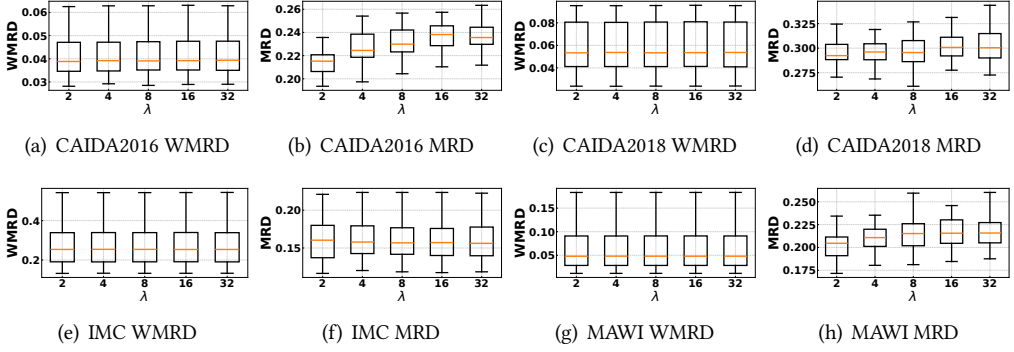


Fig. 21. Full λ sensitivity results over windows 5–54, with neural predictions reused across heavy-flow processor settings.

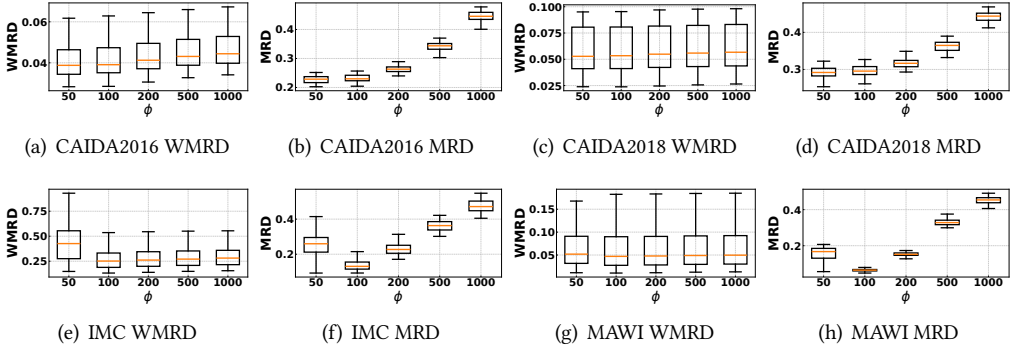


Fig. 22. Full ϕ sensitivity results over windows 5–54, varying the splice of cached predictions and fixed heavy-flow inputs.

retains attention but lacks the sorted counter-value views, and Origin+CNN and Origin+MLP have higher recorded error. The weak spatial semantics of hashed counters motivate global attention, but these comparisons do not isolate its effect from the other configuration differences.

Training and decoding cost. Figure 24 reports the corresponding computation costs. The complete decoder has slightly higher SFT time than smaller variants. The figures report time per update and inference time amortized over 400 sketch instances; observation windows each contain one million packets. These inference times exclude metric evaluation and do not measure single-request latency. Appendix E separately reports single-sketch inference and physical observation durations. The decoding cost is much smaller than the SFT cost for every neural variant because inference does not update the model weights. This separation is useful for deployment: adaptation can be scheduled independently of packet insertion, and decoding operates on the exported sketch snapshot; meeting a window deadline additionally depends on preparation and adaptation costs.

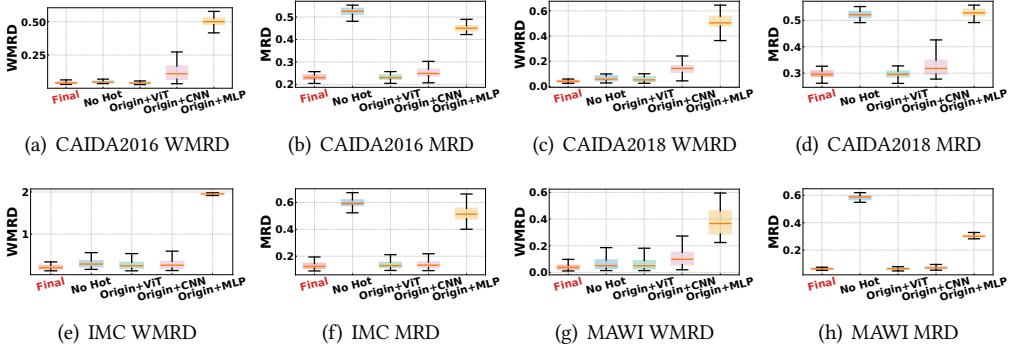


Fig. 23. Full decoder configuration comparison over windows 5–54. Input views, initialization, parameter counts, and calibration differ as described above.

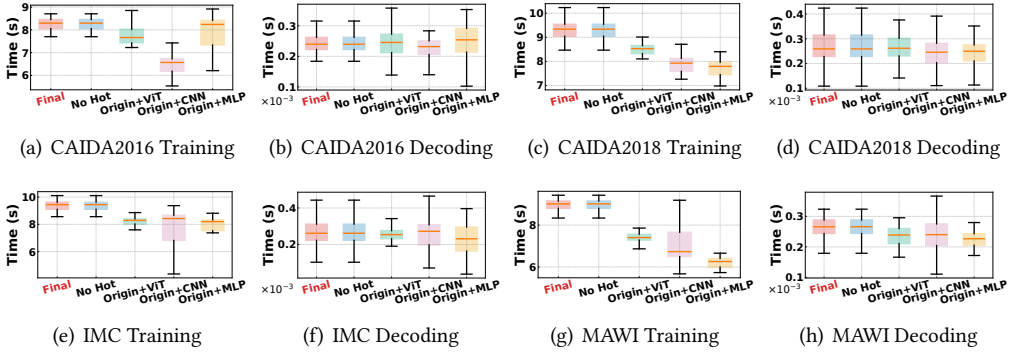


Fig. 24. Full decoder cost comparison over windows 5–54. Decoding time is amortized over 400 sketch instances.

C Proofs and Details for Section 4

C.1 Notation Summary

Table 2. Main notation used in NeuFSD.

Symbol	Meaning
S, N	Ordered packet sequence and packet count in a window
$p, e, \mathcal{E}$	Packet, flow key, and set of distinct flows
f_e, n_i	Flow size of e ; number of flows with size i
A, m, h_A	Counter array, number of counters, and array hash
B, w, E, h_B	Hot Filter, number of buckets, associativity, and bucket hash
λ	Hot Filter replacement threshold
ϕ	Hot-flow splice threshold at decoding time
G_e, J_e	Colliding mass for e and its Hot Filter bucket
M_{hot}	Packet mass directly explained by hot-spliced entries
$R_A, \bar{R}$	All array residual mass; mass within decoded support
$R, \hat{R}$	True/estimated residual packet mass
$s_{\min}, s_{\max}$	Residual-calibration clipping interval
$\pi, \hat{\pi}$	Packet-mass-normalized residual FSD shapes

C.2 Formal Hot Filter Process

For bucket j after processing the first t packets, write its state as

$$B_t^j = (K_t^j, c_t^j, v_t^j),$$

where K_t^j is the set of stored flow keys, $|K_t^j| \leq E$, $c_t^j(x)$ is the stored frequency of key $x \in K_t^j$, and v_t^j is the bucket vote. Initially $K_0^j = \emptyset$ and $v_0^j = 0$ for all j .

For packet $e_t = x$ with $j = h_B(x)$, the update is:

- (1) If $x \in K_{t-1}^j$, then $c_t^j(x) = c_{t-1}^j(x) + 1$ and the vote is unchanged.
- (2) If $x \notin K_{t-1}^j$ and $|K_{t-1}^j| < E$, then x is inserted with count 1 and the vote is unchanged.
- (3) If $x \notin K_{t-1}^j$ and $|K_{t-1}^j| = E$, then the vote is first increased to $v_{t-1}^j + 1$. Let

$$y_t \in \arg \min_{y \in K_{t-1}^j} c_{t-1}^j(y)$$

with ties resolved by a fixed deterministic rule. If

$$\frac{v_{t-1}^j + 1}{c_{t-1}^j(y_t)} < \lambda,$$

the packet is inserted into the counter array and the Hot Filter entries are unchanged. If

$$\frac{v_{t-1}^j + 1}{c_{t-1}^j(y_t)} \geq \lambda,$$

then y_t is evicted, x is inserted with count 1, and $v_t^j = 0$. The evicted mass $c_{t-1}^j(y_t)$ is replayed into the counter array.

The counter-array replay does not affect future Hot Filter states, but it is essential for residual-mass accounting.

Residual array state. Let $A_t[j]$ be the counter-array state after the t -th packet and any replay triggered by that packet. In the non-replacement branch of Case 3, $A_t[h_A(x)] = A_{t-1}[h_A(x)] + 1$. In its replacement branch, $A_t[h_A(y_t)] = A_{t-1}[h_A(y_t)] + c_{t-1}^j(y_t)$ for the evicted key y_t ; the triggering packet x starts a Hot Filter entry and is not inserted into A_t . At query time, each final Hot Filter entry $\langle x, c_N^j(x) \rangle$ with $c_N^j(x) \leq \phi$ is folded into A by adding $c_N^j(x)$ to $A[h_A(x)]$. Entries with $c_N^j(x) > \phi$ are hot-spliced directly into the output FSD and are not folded into A . This accounting assumes exact counters without overflow or saturation and a single query-time fold of the snapshot.

For a target flow e , define

$$T_e = \{t : e_t = e\}, \quad f_e = |T_e|, \quad J_e = h_B(e).$$

Let S_{J_e} denote the subsequence of packets whose flow keys hash to J_e .

Definition C.1 (Capture, complete capture, and miss). Flow e is captured at the final snapshot if $e \in K_N^{J_e}$. It is hot-spliced at threshold ϕ if $e \in K_N^{J_e}$ and $c_N^{J_e}(e) > \phi$. It is completely captured if $e \in K_N^{J_e}$ and $c_N^{J_e}(e) = f_e$. The final Hot Filter miss event is

$$\text{Miss}_B(e) = \{e \notin K_N^{J_e}\}.$$

C.3 Order-Dependence Counterexamples

The final colliding-mass condition $\{G_e \geq \lambda f_e\}$ does not characterize misses in the Hot Filter process.

Not necessary. Use $E = 1$, $\lambda = 2$, $f_e = 3$, and one colliding flow a with three packets. In the sequence

$$S_2 = (a, a, a, e, e, e),$$

the flow a first enters the only entry and reaches count 3. The three packets of e raise the vote only to 3, which is smaller than $\lambda \cdot 3 = 6$. Therefore e is never promoted and is missed. However

$$G_e = 3 < 6 = \lambda f_e.$$

Thus $\text{Miss}_B(e)$ does not imply $G_e \geq \lambda f_e$.

Not sufficient. Again use $E = 1$, $\lambda = 2$, and $f_e = 3$, but consider

$$S_3 = (a, b, b, c, c, d, e, e, e).$$

The colliding mass is $G_e = 1 + 2 + 2 + 1 = 6 = \lambda f_e$. The sequence causes churn among colliding flows before the target arrives. When the first packet of e arrives, the accumulated vote is sufficient to replace the current colliding entry with e ; the next two packets increment e to count 3. Hence e is captured even though $G_e \geq \lambda f_e$.

These examples show that total colliding mass is not a sufficient state variable for the deterministic Hot Filter algorithm. A deliberately simplified toy model may use such a mass-only condition, but Algorithm 1 requires the order-aware prefix-pressure argument.

C.4 Proof of the Prefix-Pressure Lemma

PROOF OF LEMMA 4.1. Suppose e is admitted at time τ . Define $C_e(u) = c_\tau(e) + |\{t : \tau < t \leq u, e_t = e\}|$ as a virtual cumulative count. It agrees with the stored count until the first subsequent eviction and remains defined at the eviction instant. At the moment of admission, the bucket vote is zero: admission by replacement resets the vote, while admission into an empty entry occurs before the bucket has become full and hence before any vote has accumulated for that admission path.

Assume e is first subsequently evicted at time $\sigma > \tau$. The packet at time σ must be a non- e packet hashing to J_e and not currently stored in the bucket. The eviction decision is made from the

pre-update state $B_{\sigma-1}^{J_e}$ after incrementing the vote by one. Since e is the entry evicted, it is one of the minimum-frequency entries selected by the deterministic tie-breaking rule, and

$$v_{\sigma-1}^{J_e} + 1 \geq \lambda c_{\sigma-1}^{J_e}(e).$$

Because the triggering packet is not from e ,

$$C_e(\sigma) = C_e(\sigma - 1) = c_{\sigma-1}^{J_e}(e).$$

Moreover, while e remains stored, the actual vote can increase only on non- e packets in bucket J_e that are not resident at their arrival time. Therefore the incremented vote at time σ is at most the total number of non- e bucket packets after τ up to σ :

$$v_{\sigma-1}^{J_e} + 1 \leq M_e(\sigma).$$

Combining these inequalities gives

$$M_e(\sigma) \geq \lambda C_e(\sigma).$$

Thus eviction implies a prefix-pressure violation at $u = \sigma$. The survival statement follows by contraposition. $\square$

C.5 Random-Permutation Admission-Survival Bound

Fix the hash function and condition on $G_e = g$ and $f_e = f$. Under the random permutation model, the $f + g$ packets in bucket J_e appear in uniformly random order.

Define the sufficient early-admission event

$$\mathcal{A}_E(e) = \{e \text{ appears among the first } \min\{E, f + g\} \text{ packets of } S_{J_e}\}.$$

If $\mathcal{A}_E(e)$ holds, then e is admitted before the bucket can be full, because before the E -th bucket packet there can be at most $E - 1$ distinct previously observed keys in that bucket. Therefore the first packet of e is inserted into an empty entry.

LEMMA C.2 (EARLY-ADMISSION PROBABILITY). *Conditional on $G_e = g$ and $f_e = f$,*

$$\Pr(\mathcal{A}_E(e)^c \mid G_e = g) = \begin{cases} \frac{\binom{g}{E}}{\binom{f+g}{E}}, & g \geq E, \\ 0, & g < E. \end{cases}$$

Moreover,

$$\Pr(\mathcal{A}_E(e)^c \mid G_e = g) \leq \left(\frac{g}{f+g}\right)^E.$$

PROOF. The event $\mathcal{A}_E(e)^c$ means that the first E positions of the bucket subsequence are all occupied by colliding packets. Sampling the first E positions without replacement from f target packets and g colliding packets gives the exact hypergeometric probability. The product form

$$\frac{\binom{g}{E}}{\binom{f+g}{E}} = \prod_{r=0}^{E-1} \frac{g-r}{f+g-r}$$

is at most $(g/(f+g))^E$ when $g \geq E$; when $g < E$, the exact probability is zero. $\square$

We next bound survival after early admission. Conditional on the first admission time of e , the real suffix contains exactly $f - 1$ remaining packets of e and at most g remaining colliding packets. The probability of a prefix-pressure violation is monotone nondecreasing in the number of colliding packets: starting from any fixed suffix, inserting additional colliding packets cannot remove an already existing violating prefix, because it can only increase the non- e count before the

corresponding endpoint. Therefore, replacing the actual remaining number of colliding packets by g gives a conservative dominating process.

In this dominating process, consider a uniformly random permutation of $f - 1$ target packets and g colliding packets. For a prefix of length ℓ , let

$$X_\ell \sim \text{Hypergeom}(f - 1, g, \ell)$$

be the number of target packets in that prefix. Since the target has initial stored count 1 after early admission, a prefix-pressure violation requires

$$\ell - X_\ell \geq \lambda(1 + X_\ell),$$

or equivalently

$$X_\ell \leq \frac{\ell - \lambda}{\lambda + 1}.$$

Define

$$p_{\text{hg}}(f, g, \lambda) = \min \left\{ 1, \sum_{\ell=1}^{f+g-1} \Pr \left[\text{Hypergeom}(f - 1, g, \ell) \leq \frac{\ell - \lambda}{\lambda + 1} \right] \right\}.$$

THEOREM C.3 (FULL RANDOM-PERMUTATION BOUND). For $f \geq 2$,

$$\Pr(\text{Miss}_B(e) \mid G_e = g) \leq \min \left\{ 1, p_{\text{adm}}^c(f, g, E) + p_{\text{hg}}(f, g, \lambda) \right\}.$$

Furthermore, if

$$\Delta(f, g, \lambda) = \frac{f - 1}{f + g - 1} - \frac{1}{\lambda + 1} > 0,$$

then

$$p_{\text{hg}}(f, g, \lambda) \leq \min \left\{ 1, \sum_{\ell=1}^{f+g-1} \exp(-2\ell \Delta(f, g, \lambda)^2) \right\}.$$

PROOF. If $\mathcal{A}_E(e)$ holds and no later eviction occurs, then all subsequent packets of e hit the stored entry and increment its count. Hence e is completely captured. Therefore

$$\text{Miss}_B(e) \subseteq \mathcal{A}_E(e)^c \cup \{e \text{ is evicted after early admission}\}.$$

A union bound gives the admission term plus the survival term.

For the survival term, Lemma 4.1 implies that eviction requires at least one prefix-pressure violation. The dominating suffix process described above gives the hypergeometric union bound $p_{\text{hg}}(f, g, \lambda)$.

For the exponential form, the hypergeometric variable has mean

$$\mathbb{E}[X_\ell] = \ell \frac{f - 1}{f + g - 1}.$$

Since

$$\frac{\ell - \lambda}{\lambda + 1} \leq \frac{\ell}{\lambda + 1},$$

the bad event is contained in

$$\left\{ X_\ell \leq \ell \frac{1}{\lambda + 1} \right\}.$$

When $\Delta(f, g, \lambda) > 0$, Hoeffding's inequality for sampling without replacement gives

$$\Pr \left[X_\ell \leq \ell \frac{1}{\lambda + 1} \right] \leq \exp(-2\ell \Delta(f, g, \lambda)^2).$$

Applying a union bound over ℓ proves the result. $\square$

PROOF OF COROLLARY 4.3. By the collision-uniform hash assumption,

$$\mathbb{E}[G_e] = \sum_{x \neq e} f_x \Pr[h_B(x) = h_B(e)] = \frac{F-e}{w}.$$

Thus Markov's inequality gives

$$\Pr(G_e > g_0) \leq \frac{F-e}{wg_0}.$$

Conditioned on $G_e \leq g_0$, Theorem 4.2 gives the worst-case admission-survival bound over $0 \leq g \leq g_0$. Combining the two events proves the corollary. $\square$

C.6 Poisson Race Bound After Admission

The random-permutation model is convenient for fixed packet multisets. A Poisson model gives a complementary rate-based interpretation. It is conditional on admission and does not bound the probability that a flow enters the Hot Filter.

THEOREM C.4 (POISSON RACE AFTER ADMISSION). *Fix the hash functions and $\lambda > 0$. Suppose flow e has been admitted to bucket J_e with stored count $c_0 \geq 1$ and zero initial vote. After this admission time, suppose packets of e arrive as a Poisson process of rate $\rho_e > 0$, and all non- e packets hashing to J_e arrive as an independent aggregate Poisson process of rate*

$$\mu_e = \sum_{x \neq e: h_B(x) = J_e} \rho_x.$$

Let $\kappa = \mu_e / \rho_e$. If $\kappa = 0$, the eviction probability is zero. If $0 < \kappa < \lambda$, let $r_{\kappa, \lambda} \in (0, 1)$ be the nontrivial root of

$$r^\lambda + \kappa r^{-1} = 1 + \kappa.$$

Then the probability that e is ever evicted after this admission time is at most

$$r_{\kappa, \lambda}^{\lambda c_0}.$$

PROOF. By Lemma 4.1, eviction is possible only if the prefix-pressure process hits zero or below. Define the dominating embedded random walk

$$X_n = \lambda C_n - M_n,$$

where C_n is c_0 plus the number of target arrivals and M_n is the number of non- e bucket arrivals in the embedded sequence after admission. The virtual C_n agrees with the stored count until eviction. When $\kappa = 0$ no non- e packet can trigger eviction, so consider $0 < \kappa < \lambda$. Initially,

$$X_0 = \lambda c_0.$$

At each bucket arrival, X_n increases by λ with probability

$$\alpha_e = \frac{\rho_e}{\rho_e + \mu_e} = \frac{1}{1 + \kappa}$$

and decreases by 1 with probability

$$\beta_e = \frac{\mu_e}{\rho_e + \mu_e} = \frac{\kappa}{1 + \kappa}.$$

The drift is positive exactly when $\lambda \alpha_e - \beta_e > 0$, i.e., $\kappa < \lambda$.

Choose $r = r_{\kappa, \lambda} \in (0, 1)$ such that

$$\alpha_e r^\lambda + \beta_e r^{-1} = 1,$$

which is equivalent to $r^\lambda + \kappa r^{-1} = 1 + \kappa$. To establish the root for every $\lambda > 0$, set $r = e^{-\theta}$. The function $H(\theta) = e^{-\lambda\theta} + \kappa e^\theta$ is strictly convex, satisfies $H(0) = 1 + \kappa$ and $H'(0) = \kappa - \lambda < 0$, and diverges as $\theta \rightarrow \infty$. Hence it has a unique positive solution of $H(\theta) = 1 + \kappa$, giving the required root $r \in (0, 1)$.

Then r^{X_n} is a martingale for the embedded chain. Let

$$\tau = \inf\{n : X_n \leq 0\}.$$

For $\tau \wedge n$, optional stopping gives

$$\mathbb{E}[r^{X_{\tau \wedge n}}] = r^{X_0} = r^{\lambda c_0}.$$

On $\{\tau \leq n\}$, we have $X_\tau \leq 0$, so $r^{X_\tau} \geq 1$ because $0 < r < 1$. Therefore

$$\Pr(\tau \leq n) \leq r^{\lambda c_0}.$$

Letting $n \rightarrow \infty$ yields

$$\Pr(\tau < \infty) \leq r^{\lambda c_0}.$$

Since eviction is contained in the hitting event $\{\tau < \infty\}$, the same bound applies to eviction after admission. $\square$

The theorem explains the post-admission role of λ in a sufficient survival bound. For fixed rates, larger λ increases the positive drift margin of the dominating pressure process. The resulting bound describes protection after admission; final capture also depends on replacement-based admission and on the vote resets induced by other entries.

C.7 Counter Array Proof

PROOF OF THEOREM 4.4. All expectations below are conditional on the packet sequence and Hot Filter state. The array hash h_A is independent of the Hot Filter hash h_B and the hash-oblivious packet sequence, and remains uniform and pairwise independent across keys under this conditioning. For each distinct residual key $x \in \mathcal{U}$, define

$$X_{x,j} = \mathbf{1}\{h_A(x) = j\}.$$

Then

$$C_j = \sum_{x \in \mathcal{U}} a_x X_{x,j}.$$

Uniform hashing gives $\mathbb{E}[X_{x,j}] = 1/m$, and by linearity of expectation,

$$\mathbb{E}[C_j] = \sum_{x \in \mathcal{U}} a_x \mathbb{E}[X_{x,j}] = \frac{1}{m} \sum_{x \in \mathcal{U}} a_x = \frac{R_A}{m}.$$

For the variance, pairwise independence across distinct keys makes all covariance terms vanish:

$$\text{Var}(C_j) = \sum_{x \in \mathcal{U}} a_x^2 \text{Var}(X_{x,j}).$$

Since $X_{x,j}$ is Bernoulli with probability $1/m$,

$$\text{Var}(X_{x,j}) = \frac{1}{m} \left(1 - \frac{1}{m}\right) = \frac{m-1}{m^2}.$$

Therefore

$$\text{Var}(C_j) = \frac{m-1}{m^2} \sum_{x \in \mathcal{U}} a_x^2 = \frac{m-1}{m^2} F_{2,A}.$$

$\square$

The aggregation by flow key is essential. If the same flow contributes multiple chunks to A , those chunks share the same value $h_A(x)$ and must not be treated as independently hashed records.

C.8 End-to-End Error Decomposition Proof

Let A^{fold} be the folded counter array. After all query-time folding, aggregate the array-represented residual mass by flow key and denote it by a_x . Define the decoded-support residual FSD

$$q_{0,i} = |\{x : a_x = i\}|, \quad 1 \leq i \leq \phi.$$

Keys with residual mass $a_x > \phi$ are outside the decoder output support and are therefore charged to the splice error. Let n_B^{hot} be the FSD vector contributed by Hot Filter entries with stored count $> \phi$, and let

$$n_B^{\text{splice}} = n_B^{\text{hot}} + q_0,$$

where q_0 is zero-padded outside $1, \dots, \phi$. Thus

$$\epsilon_{\text{splice}} = \|n - n_B^{\text{splice}}\|_1$$

captures missed hot flows, partially captured hot-flow spillover, and any residual array key whose mass exceeds ϕ .

Let

$$\begin{aligned} q_1 &= g_t^*(A^{\text{fold}}), & q_2 &= g_{\mathcal{F},t}(A^{\text{fold}}), \\ q_3 &= g_{\mathcal{F},\text{hist}}(A^{\text{fold}}), & q_4 &= g_{\mathcal{F},\text{hist}}^{\text{samp}}(A^{\text{fold}}), & q_5 &= \hat{g}_t(A^{\text{fold}}). \end{aligned}$$

Here g_t^* is an ideal residual decoder for the current window distribution; $g_{\mathcal{F},t}$ is the best decoder in the chosen neural function class for the current distribution; $g_{\mathcal{F},\text{hist}}$ is the best decoder for the recent SFT-window distribution; $g_{\mathcal{F},\text{hist}}^{\text{samp}}$ is the corresponding decoder trained against sampled labels; and $\hat{g}_t$ is the actually learned decoder. All six vectors are taken in $\mathbb{R}_{\geq 0}^\phi$, after any required nonnegative output projection.

Let $r = (1, \dots, \phi)^\top$ and $\bar{R} = r^\top q_0$. The algorithm instead observes the total folded-array mass $R_A = N - M_{\text{hot}} = \sum_x a_x$. In particular,

$$R_A - \bar{R} = \sum_{x: a_x > \phi} a_x \geq 0,$$

so the two masses need not agree. For any target mass $T \geq 0$ and $q \in \mathbb{R}_{\geq 0}^\phi$, define

$$C_T(q) = \begin{cases} \text{clip}\left(\frac{T}{r^\top q}, s_{\min}, s_{\max}\right) q, & r^\top q > 0, \\ q, & r^\top q = 0. \end{cases}$$

Assume $0 \leq s_{\min} \leq 1 \leq s_{\max} < \infty$, so that calibration with the true decoded-support mass leaves its residual vector unchanged:

$$C_{\bar{R}}(q_0) = q_0.$$

Define

$$\epsilon_{\text{counter}} = \|C_{\bar{R}}(q_0) - C_{\bar{R}}(q_1)\|_1,$$

$$\epsilon_{\text{approx}} = \|C_{\bar{R}}(q_1) - C_{\bar{R}}(q_2)\|_1,$$

$$\epsilon_{\text{drift}} = \|C_{\bar{R}}(q_2) - C_{\bar{R}}(q_3)\|_1,$$

$$\epsilon_{\text{sample}} = \|C_{\bar{R}}(q_3) - C_{\bar{R}}(q_4)\|_1,$$

and

$$\epsilon_{\text{gen}} = \|C_{\bar{R}}(q_4) - C_{\bar{R}}(q_5)\|_1.$$

The additional mass-target mismatch is

$$\epsilon_{\text{mass}} = \|C_{R_A}(q_5) - C_{\bar{R}}(q_5)\|_1.$$

The actual algorithm output and the splice reference are

$$\hat{n} = n_B^{\text{hot}} + C_{R_A}(q_5), \quad n_B^{\text{splice}} = n_B^{\text{hot}} + q_0.$$

By the triangle inequality,

$$\|n - \hat{n}\|_1 \leq \|n - n_B^{\text{splice}}\|_1 + \|n_B^{\text{splice}} - \hat{n}\|_1.$$

The first term is ϵ_{splice} . For the second term, insert calibration with the decoded-support mass:

$$\|q_0 - C_{R_A}(q_5)\|_1 \leq \|C_{\bar{R}}(q_0) - C_{\bar{R}}(q_5)\|_1 + \epsilon_{\text{mass}}.$$

The first norm is at most the sum of the five intermediate differences above. This proves the seven-term bound in Proposition 4.5.

For $Q = r^\top q_5 > 0$, writing $\text{clip}(z)$ for clipping to the same interval gives

$$\begin{aligned} \epsilon_{\text{mass}} &= \|q_5\|_1 |\text{clip}(R_A/Q) - \text{clip}(\bar{R}/Q)| \\ &\leq \|q_5\|_1 \min \left\{ \frac{|R_A - \bar{R}|}{Q}, s_{\max} - s_{\min} \right\} \\ &\leq |R_A - \bar{R}|. \end{aligned}$$

The first inequality uses the nonexpansiveness and bounded range of scalar clipping; the second uses $r_i \geq 1$ and hence $\|q_5\|_1 \leq Q$. If $Q = 0$, nonnegativity and $r_i > 0$ imply $q_5 = 0$, so both calibrated outputs and ϵ_{mass} are zero.

For WMRD, divide the seven-term bound by the actual output denominator $(\|n\|_1 + \|\hat{n}\|_1)/2$, provided it is positive, with both vectors evaluated over the same bins. The denominator uses $\hat{n} = n_B^{\text{hot}} + C_{R_A}(q_5)$, not the hypothetical prediction calibrated to $\bar{R}$.

C.9 Residual-Mass Calibration Proofs

Throughout this subsection,

$$\begin{aligned} r &= (1, 2, \dots, \phi)^\top, & D &= \text{diag}(1, 2, \dots, \phi), \\ R &= r^\top n, & \hat{R} &= r^\top \hat{n}. \end{aligned}$$

Here n denotes the true residual vector on the decoded support, rather than the full FSD from the preceding subsection. We assume $n, \hat{n} \in \mathbb{R}_{\geq 0}^\phi$ and $R > 0, \hat{R} > 0$; the case $R = 0$ is degenerate for the normalized shapes $\pi, \hat{\pi}$ and is handled by the calibration operator's clipping/skip rule rather than by this shape theorem. The theorem uses the true support mass $R = r^\top n$. The available mass $R_A = N - M_{\text{hot}}$ can include residual keys outside that support and cannot automatically replace R in an improvement guarantee.

PROOF OF PROPOSITION 4.6. Without clipping,

$$r^\top \tilde{n} = r^\top \left(\frac{R}{\hat{R}} \hat{n} \right) = \frac{R}{\hat{R}} r^\top \hat{n} = R.$$

With clipping, $s = \text{clip}(s_0, s_{\min}, s_{\max})$ and $s_0 = R/\hat{R}$. Hence

$$|r^\top (s\hat{n}) - R| = |s\hat{R} - s_0\hat{R}| = \hat{R}|s - s_0|.$$

□

PROOF OF THEOREM 4.7. Define

$$\pi = \frac{Dn}{R}, \quad \hat{\pi} = \frac{D\hat{n}}{\hat{R}}.$$

Both are probability vectors because their entries are nonnegative and sum to one. Without clipping,

$$D\tilde{n} = D\left(\frac{R}{\hat{R}}\hat{n}\right) = R\frac{D\hat{n}}{\hat{R}} = R\hat{\pi}.$$

Also $Dn = R\pi$. Therefore

$$\|D(\tilde{n} - n)\|_1 = \|R\hat{\pi} - R\pi\|_1 = R\|\hat{\pi} - \pi\|_1.$$

For the uncalibrated prediction,

$$\|D(\hat{n} - n)\|_1 = \|\hat{R}\hat{\pi} - R\pi\|_1 \geq |\mathbf{1}^\top (\hat{R}\hat{\pi} - R\pi)| = |\hat{R} - R|,$$

where the inequality is the reverse triangle inequality. Thus calibration strictly improves the weighted L_1 error whenever

$$R\|\hat{\pi} - \pi\|_1 < |\hat{R} - R|.$$

□

If the mass target differs from the true support mass, an additive error term is necessary. Suppose calibration uses $T \geq 0$ instead of R , with $|T - R| \leq \delta R$. Without clipping, its weighted error satisfies

$$\left\| D\left(\frac{T}{\hat{R}}\hat{n} - n\right) \right\|_1 = \|T\hat{\pi} - R\pi\|_1 \leq R\|\hat{\pi} - \pi\|_1 + \delta R.$$

For clipping to $s_T = \text{clip}(T/\hat{R}, s_{\min}, s_{\max})$, an additional term $\hat{R}|s_T - T/\hat{R}|$ bounds the effect of clipping. Thus neither the full-array target nor clipping inherits the exact-target improvement condition without accounting for these terms. A weighted- L_1 improvement also does not by itself prove a WMRD improvement: WMRD has an unweighted count numerator and a prediction-dependent denominator.

THEOREM C.5 (MULTIPLICATIVE-BIAS CORRECTION). *Suppose*

$$n = \alpha q + \epsilon, \quad \hat{n} = \beta q + \eta,$$

where $q \in \mathbb{R}_{\geq 0}^\phi$ is a common residual FSD shape, $\alpha, \beta > 0$, and ϵ, η are shape errors. Let

$$Q = r^\top q, \quad E_R = r^\top \epsilon, \quad H_R = r^\top \eta.$$

Assume $Q > 0$ and $\hat{R} = \beta Q + H_R > 0$. For the un-clipped scale $s_0 = R/\hat{R}$,

$$\|s_0\hat{n} - n\|_1 \leq s_0\|\eta\|_1 + \|\epsilon\|_1 + \|q\|_1 \frac{\beta|E_R| + \alpha|H_R|}{\hat{R}}.$$

Meanwhile,

$$\|\hat{n} - n\|_1 \geq |\beta - \alpha| \|q\|_1 - \|\eta\|_1 - \|\epsilon\|_1.$$

Therefore calibration improves the residual L_1 numerator whenever the first upper bound is smaller than the second lower bound. In the exact multiplicative-bias case $\epsilon = \eta = 0$, calibration recovers n exactly.

PROOF. Since

$$R = r^\top n = \alpha Q + E_R, \quad \hat{R} = r^\top \hat{n} = \beta Q + H_R,$$

we have

$$s_0\beta - \alpha = \frac{\beta(\alpha Q + E_R)}{\beta Q + H_R} - \alpha = \frac{\beta E_R - \alpha H_R}{\hat{R}}.$$

Thus

$$s_0 \hat{n} - n = (s_0 \beta - \alpha)q + s_0 \eta - \epsilon.$$

Taking norms and applying the triangle inequality gives

$$\|s_0 \hat{n} - n\|_1 \leq |s_0 \beta - \alpha| \|q\|_1 + s_0 \|\eta\|_1 + \|\epsilon\|_1$$

and

$$|s_0 \beta - \alpha| \leq \frac{\beta |E_R| + \alpha |H_R|}{\hat{R}}.$$

This proves the upper bound.

For the uncalibrated vector,

$$\hat{n} - n = (\beta - \alpha)q + \eta - \epsilon.$$

The reverse triangle inequality gives

$$\|\hat{n} - n\|_1 \geq |\beta - \alpha| \|q\|_1 - \|\eta\|_1 - \|\epsilon\|_1.$$

If $\epsilon = \eta = 0$, then $R = \alpha Q$, $\hat{R} = \beta Q$, and $s_0 = \alpha/\beta$, so $s_0 \hat{n} = n$. □

PROPOSITION C.6 (PROJECTION AND CLIPPING). *The un-clipped scale $s_0 = R/\hat{R}$ is the unique minimizer of the scalar mass objective*

$$J_{\text{mass}}(s) = |r^\top (s\hat{n} - n)|^2.$$

The clipped scale

$$s = \text{clip}(s_0, s_{\min}, s_{\max})$$

is the minimizer of $J_{\text{mass}}(s)$ over $[s_{\min}, s_{\max}]$.

For the weighted least-squares objective

$$J_2(s) = \|D(s\hat{n} - n)\|_2^2,$$

the unconstrained optimum is

$$s_2^\star = \frac{\langle D\hat{n}, Dn \rangle}{\|D\hat{n}\|_2^2}.$$

Writing

$$\pi = \frac{Dn}{R}, \quad \hat{\pi} = \frac{D\hat{n}}{\hat{R}},$$

we have

$$s_2^\star = s_0 \frac{\langle \hat{\pi}, \pi \rangle}{\|\hat{\pi}\|_2^2}$$

and therefore

$$\left| \frac{s_2^\star}{s_0} - 1 \right| \leq \frac{\|\pi - \hat{\pi}\|_2}{\|\hat{\pi}\|_2}.$$

Thus the residual-mass scale is close to the weighted least-squares optimal scale when the predicted packet-mass shape is close to the true packet-mass shape.

PROOF. Since

$$J_{\text{mass}}(s) = |s\hat{R} - R|^2,$$

its unconstrained minimizer is $s_0 = R/\hat{R}$. The constrained minimizer over an interval is the Euclidean projection of this scalar onto the interval, which is exactly clipping.

For J_2 , differentiate:

$$J_2(s) = \langle D(s\hat{n} - n), D(s\hat{n} - n) \rangle.$$

Setting the derivative to zero yields

$$s_2^\star = \frac{\langle D\hat{n}, Dn \rangle}{\|D\hat{n}\|_2^2}.$$

Substituting $Dn = R\pi$ and $D\hat{n} = \hat{R}\hat{\pi}$ gives

$$s_2^\star = \frac{\hat{R}R\langle \hat{\pi}, \pi \rangle}{\hat{R}^2\|\hat{\pi}\|_2^2} = s_0 \frac{\langle \hat{\pi}, \pi \rangle}{\|\hat{\pi}\|_2^2}.$$

Therefore

$$\left| \frac{s_2^\star}{s_0} - 1 \right| = \frac{|\langle \hat{\pi}, \pi - \hat{\pi} \rangle|}{\|\hat{\pi}\|_2^2} \leq \frac{\|\hat{\pi}\|_2 \|\pi - \hat{\pi}\|_2}{\|\hat{\pi}\|_2^2} = \frac{\|\pi - \hat{\pi}\|_2}{\|\hat{\pi}\|_2}.$$

□

For the weighted L_1 objective

$$J_1(s) = \|D(s\hat{n} - n)\|_1 = \sum_{x=1}^{\phi} x |s\hat{n}_x - n_x|,$$

an unconstrained minimizer is a weighted median of the ratios $n_x/\hat{n}_x$ over bins with $\hat{n}_x > 0$, with weights $x\hat{n}_x$; bins with $\hat{n}_x = 0$ contribute terms independent of s . If $\text{supp}(n) \subseteq \text{supp}(\hat{n})$, then $s_0 = R/\hat{R}$ is the corresponding weighted mean of these ratios. Hence s_0 is L_1 -optimal when the ratios are equal and is close to an L_1 optimum when the ratios are concentrated. Without such shape-alignment conditions, no unconditional L_1 non-increase is possible.

For example, with $\phi = 2$,

$$n = (100, 0), \quad \hat{n} = (0, 1).$$

Then $R = 100$, $\hat{R} = 2$, and $s_0 = 50$. The uncalibrated count L_1 error is

$$\|n - \hat{n}\|_1 = 101,$$

whereas the calibrated prediction is $(0, 50)$ and has count L_1 error

$$\|n - s_0\hat{n}\|_1 = 150.$$

Thus scalar rescaling can hurt when the predicted shape is wrong; the calibration theorem is necessarily conditional.

C.10 Illustrative Checks for the Analysis

The following checks illustrate the assumptions behind the analysis. Complementary verification uses exact arithmetic for 1,500 calibration cases and enumerates 118,080 prefix-pressure cases and 240 random-permutation settings. Numerical checks cover 15 Poisson-root settings. These check the algebra and finite examples; the remaining descriptions specify broader simulation designs.

Order-dependence sanity check. Use one bucket with $E = 1$ and $\lambda = 2$. Compare

$$(e, e, e, a, a, a) \quad \text{and} \quad (a, a, a, e, e, e).$$

Both have $f_e = 3$ and the same colliding mass, but only the first captures e . This checks that final colliding mass alone is not a sufficient state variable.

Random-permutation capture. Fix f_e , g , E , and λ . Generate many random permutations of f_e target packets and g colliding packets in one bucket. Measure final capture and complete-capture probabilities. Also record

$$\max_u \frac{M_e(u)}{C_e(u)}$$

after admission. Misses should correlate with prefix-pressure violations rather than with total mass alone.

Hash scaling. One could fix a background flow multiset and draw fresh h_B seeds, then vary w and record G_e for selected target hot flows. The exact expectation is F_{-e}/w under collision-uniform hashing. Empirical capture can be compared separately; the mean-collision identity does not determine it without the order assumptions.

Associativity. One could repeat the random-permutation experiment for $E = 1, 2, 4, 8$ at fixed w . The early-admission failure probability is given by Lemma C.2 and is bounded above by $(g/(f+g))^E$. A simulation could compare the measured miss probability with the conservative admission-survival bound.

Threshold trade-off. At a fixed admitted state, increasing λ relaxes the sufficient prefix-survival condition. A simulation could separately measure replacement-based admission and subsequent eviction; the bound does not imply that final capture improves monotonically as λ increases.

Residual-mass calibration. Generate residual FSDs of the form

$$n_t = \alpha_t q + \epsilon_t, \quad \hat{n}_t = \beta_t q + \eta_t.$$

One could sweep the scalar bias $|\alpha_t - \beta_t|$ and shape noise $\|\epsilon_t\|_1 + \|\eta_t\|_1$, distinguishing weighted L_1 , count L_1 , and WMRD. Theorems 4.7 and C.5 give sufficient conditions for the first two errors; predictions outside those conditions need not improve. Tests using R_A should also vary $R_A - R$ to expose the mass-target mismatch.

D Model, Training, and Resource Configuration

This appendix specifies the evaluated decoder and the configurations of the original and supplementary experiments. Complete model and training configuration tables, split definitions, and reproduction commands are available with the implementation at <https://github.com/stallone0000/NeuFSD>; see docs/MODEL.md and docs/REPRODUCING.md. The repository provides source code and configuration instructions; users obtain the traffic datasets separately and generate experiment outputs locally.

Decoder configuration. The implementation uses two independently parameterized ViTs, each with its own encoder backbone. Table 3 specifies both networks. Counters are cast from uint32 to float32, arranged as ascending/original/descending channels, and reshaped in row-major order. The reference input is raw; optional environment settings apply $\log(1+c)$ or $\log(1+c)/\max\{\log(1+\sum_j c_j), 1\}$. Origin-only input is a separately identified ablation. The patch grid provides fixed tokenization and positional indices for the counters. At $\phi = 100$, the final neural estimate uses all 10 outputs of the first network and the first 90 outputs of the second; the heavy-flow branch supplies larger sizes, using linear interpolation and endpoint extrapolation in the evaluator. A supplementary construction and forward check of the 128×128 variant verified 65 tokens, 174,026,722 parameters, and outputs of dimensions 10 and 1,080 using freshly initialized weights.

Table 3. Reference model configuration. Parameter counts include both complete networks, all biases, normalization parameters, positions, and CLS tokens.

Component	Configuration
Input / patches	Three 64×64 channels; Conv2D $3 \rightarrow 768$, 16×16 kernel and stride, bias
Tokens / position	16 patch tokens plus learned CLS token; learned 17×768 position embedding
Encoder	12 pre-LayerNorm blocks; width 768; 12 attention heads of width 64; QKV/projection biases; attention scale $64^{-1/2}$
Feed-forward	Linear $768 \rightarrow 3072$, GELU, Linear $3072 \rightarrow 768$, biases; residual branches for attention and MLP
Normalization / dropout	Backbone LayerNorm $\epsilon = 10^{-6}$; backbone dropout 0; final LayerNorm and CLS selection
Regression	LayerNorm($768, \epsilon = 10^{-5}$), Linear($768, 1000$), GELU, dropout 0.1, Linear($1000, o$); no output activation
Outputs	$o = 10$: sizes 1–10; $o = 1080$: sizes 11–1000 and 1001, 1101, $\dots$, 9901
Parameters / FP32 bytes	$86,440,962 + 87,512,032 = 173,952,994$; 695,811,976 bytes of parameter tensors
128 $\times$ 128 variant	64 patches + CLS; $86,477,826 + 87,548,896$ parameters

Loss and optimization. Each network minimizes mean SmoothL1 on target flow counts, with scalar loss $\ell(e) = e^2/2$ for $|e| < 1$ and $|e| - 1/2$ otherwise. The reference trainers use Adam with $\beta = (0.9, 0.999)$, $\epsilon = 10^{-8}$, zero weight decay, and amsgrad=False; learning rate is constant, with no scheduler, gradient clipping, or automatic mixed precision. Full-model SFT updates all parameters. Pretraining selects the lowest validation WMRD checkpoint. Adaptation retains the final epoch and records holdout loss. Table 4 gives per-network workloads; evaluation metric columns and training checkpoint selection are distinct choices. The reference Python/NumPy/Torch seed is 42.

Data generation and split. The two synthetic roots each contain 30 datasets combining 80,000 or 100,000 flows, 10^6 packets, and Zipf exponents $\{0, 0.2, 1.0, 1.1, \dots, 2.0, 2.5, 3.0\}$. Every flow receives one packet, then largest-remainder rounding allocates the remaining packets by Zipf weights. The reference ordering seed is 20260617. Synthetic targets are exact counts on the output grid, including zeros. Each synthetic root contains 30 datasets; training uses 27 and validation uses another three. With 400 counter seeds per dataset, the split is $(2 \cdot 30 - 2 \cdot 3)400 = 21,600$ training and $2 \cdot 3 \cdot 400 = 2,400$ validation examples. Each example records its seed, so the counter array can be regenerated from the stored configuration. Stored synthetic targets are retained; the preparation records explicit zero counts at unoccupied output coordinates.

The supplementary synthetic prior uses only the 60 synthetic datasets, with the workload in Table 4 and Python/NumPy/Torch seed 42. It holds out the final 12 lexically ordered datasets after concatenating the roots. The holdout is from the second format, whose distribution shapes overlap those of the first. Its observed settings are raw three-channel inputs, four Torch threads, matmul TF32 off, cuDNN TF32 on, and cuDNN benchmarking off; it uses Python 3.10.12, PyTorch 2.6.0+cu124/CUDA 12.4, and NumPy 2.2.6 on RTX 4090. The supplementary update profiles in Appendix E use FP32 with both matmul and cuDNN TF32 allowed, cuDNN benchmarking enabled, and no autocast. Historical training uses the H800 setting.

Adaptation details. Whole-flow sampling uses the 64-bit BLAKE2b hash of the key, uint32 little-endian window index, and uint32 little-endian sample seed. Selected flows retain their full block size; size-bin counts are divided by q . The reference sampler first builds a complete host dictionary, with no fixed-capacity eviction. Sampled labels interpolate/extrapolate occupied bins and clip negatives;

Table 4. Training configuration per network. Superscript R denotes a reference trainer setting where the historical run record does not specify that value. Supplementary adaptation uses the matched Avg5 settings with the synthetic-only prior.

Setting	Initial prior	synthetic	Matched Avg5 adaptation	IMC snapshot coder configuration	de-configuration	Supplementary synthetic-only prior
Training information	60 synthetic datasets		Average of preceding 5 sampled FSDs	Fixed output grid for regression		60 synthetic datasets
Sampling probability	Full-FSD sources		$q = 0.1$	$q = 0.5$		Exact labels
Examples / split	21,600 train / 2,400 validation		2,000/update; 1,800 train / 200 holdout	2,000/window; 1,800 train / 200 holdout		24,000; 19,200 train / 4,800 validation
Split unit	Last 3 datasets in each of 2 roots		Random 10% of synthesized counters	Random 10% of synthesized counters		Last 12 datasets in concatenated list
Epochs / batch	20 / 64		5 / 256	20 / 256 ^R		10 / 64
Learning rate	10^{-2}		10^{-2}	10^{-2R}		10^{-2}
Counter seeds	0–399 per dataset		0–399 for each of 5 units	2,000 seeds; template 0–1999		0–399 per dataset
Sampling Torch seed	/ — / 42 ^R		42 / 42	42 ^R / 42 ^R		— / 42
Initialization	Random		Initial synthetic prior	Initial synthetic prior		Random
Weight policy	Train for 20 epochs		Carry weights across windows	Keep weights per snapshot ^R		Train for 10 epochs
Optimizer policy	Adam per network		Reset Adam each update	Reset Adam each window ^R		Adam per network
Order	Pretrain, then evaluate		History labels, SFT, decode current window	Read counters, infer, merge output vectors		Pretrain, then adapt

an empty FSD gives zeros and a single occupied bin gives a constant target. Rounded nonnegative bin populations generate fresh synthetic keys. In the matched 2,000-example group, Window5 uses five historical FSDs, Avg5 five key replicas of their binwise average, and Last five replicas of the latest FSD; each unit uses 400 counter seeds. Holdout seeds are $42 + 17u$ and $42 + 17u + 100000$ for the two networks, with u including replica offsets. The configured IMC tensors preserve channel dimensions and consistent counter ordering. Training loaders shuffle; validation/holdout loaders retain order with batches 128/512. Adaptation uses zero workers; the pretraining reference worker default is zero.

Historical sliding indices are 5–303 for CAIDA2016, 5–728 for CAIDA2018, 5–97 for IMC, and 5–113 for MAWI. Indices denote packet blocks. Trace keys are 8 bytes at offset 8 of a 16-byte CAIDA2016 record, 13 bytes at offset 0 of a 21-byte CAIDA2018 record, and 13-byte IMC/MAWI records. The supplementary IMC conversion uses the directed IPv4 source-address/source-port/destination-address/destination-port/protocol tuple, without a timeout or bidirectional aggregation. The trace-based neural evaluation uses full-stream counters and separate heavy-flow inputs; IMC/MAWI use exact heavy-flow counts. The configured IMC decoder additionally uses fixed regression coordinates, independently of the representation of its input counters. For each counter seed, scoring splices predictions, rounds, clips negatives, and evaluates on the loaded truth support before averaging metrics across seeds. Neural prediction includes size 100; final heavy prediction starts above 100, while the heavy-mass loader includes size 100 in calibration bookkeeping.

Supplementary allocation details. The original 32-bit-fingerprint C++ prototype at $w = 1000$, $E = 8$, $m = 4096$ requests 152,456 bytes per instance: 128,000 entry bytes, 16,384 counter bytes, 8,024 pointer-array bytes, 8 hash-object bytes, and a 40-byte object; shared hash constants add 5,116 bytes per process. The separate 64-bit-key prototype requests 176,464 bytes per instance: 128,000 for entries, 32,000 for bucket/vector metadata, 16,384 for counters, and 80 for the top-level object. These implementations use distinct vote/update policies. Both measurements count object and backing-storage payload, excluding allocator headers, fragmentation, and shared code. Logical entry widths, C++ layout, and compiled switch resources are separate accounting scopes.

At nominal 16 KiB on IMC block 5, query-time tracked heap peaks for Elastic, MRAC, Array Sample, Hash Sample, and DaVinci are 1,606,268, 7,123,824, 115,672, 98,824, and 4,011,814 bytes, respectively, including live insertion allocations. Elastic also has a 63,488-byte local stack counter copy; DaVinci has a 368-byte top-level stack object. Stack and heap maxima need not coincide. Hash occupancy is 74/390 keys, and the allocation ledger includes tracking-container growth. Allocation instrumentation is used for memory accounting, with decoder timing measured separately.

For whole-flow sampling at $q = 0.1$ and seed 42 on block 5, the full dictionaries contain 7,100, 225,501, and 91,939 flows for IMC, MAWI, and CAIDA2018; 692, 22,818, and 9,170 flows are selected. Their reachable full-dictionary payloads are 0.604, 19.912, and 9.053 MiB. These object inventories are separate from Python allocation peaks and process RSS. Three repetitions without the memory profiler give median parsing, label-construction, and synthetic-artifact preparation times of 0.437, 0.931, and 0.646 s, respectively, on one pinned CPU core with warmed input pages. Imports and input validation are excluded; complete scope and allocation receipts accompany the reproduction parameters.

E Supplementary Reproduction and Cost Measurements

We complement the original H800 experiments with a separately configured reproduction on an NVIDIA RTX 4090 and Intel Xeon Gold 6230R host. These runs use the synthetic-only prior and configuration in Appendix D. The repository documents the model, training settings, and reproduction workflow. Each accuracy configuration is one training execution; its 400 counter seeds are sketch instances, rather than independent training trials.

Adaptation, initialization, and sampled history. Table 5 reports all eight completed configurations. Unless specified otherwise, the label sampling probability is $q = 0.1$, and an update uses 2,000 examples per model, five epochs, batch size 256, and Adam with learning rate 0.01. Predictions use full-stream counters and exact-heavy auxiliary input. The table reports raw, uncorrected output, scored on occupied true bins through size 10,000. The code scores each sketch seed after splicing, rounding, and nonnegative projection, and then averages scores over seeds and windows. The evaluation scripts generate per-window results with separately named correction variants.

The matched continuous and fixed-once IMC runs start with identical first-window predictions. Continuous adaptation has lower WMRD in 62 windows, equal WMRD in one, and higher WMRD in 30. Its native SFT time totals 463.328 seconds for 93 updates; the fixed-once variant spends 5.544 seconds on its one update. These intervals exclude pretraining and input preparation. On the same first ten evaluation windows (indices 5–14), $q = 0.1$ gives mean WMRD 0.220665, compared with 0.365902 at $q = 0.01$. The two initialization rows report both outcomes under the same Avg-5 protocol.

For MAWI and CAIDA2018, the mean counter/root construction plus SFT costs are 11.21 and 11.07 seconds. These measurements separate observation duration from control-plane work, allowing an implementation to choose its update frequency and buffering according to the available compute

Table 5. Supplementary reproduction with the stated synthetic-only prior, except the scratch row. Each row includes every predefined evaluation window.

Configuration	Windows	Mean WMRD	Mean MRD
IMC, continuous Avg-5	93	0.280415	0.136977
IMC, first update only	93	0.390152	0.140251
IMC, Last-1	93	0.350218	0.194567
IMC, Window-5	93	0.340112	0.184549
IMC, scratch Avg-5	93	0.279199	0.136969
IMC, Avg-5, $q = 0.01$	10	0.365902	0.202657
MAWI, Avg-5	109	0.073509	0.069253
CAIDA2018, Avg-5	101	0.036813	0.034976

budget. Scheduling at capture rate additionally includes sample-cache preparation, inference, and export.

Common-support diagnostic. We additionally evaluate all four combinations of occupied or union support and signed or nonnegative scores for the supplementary neural run and the five baselines. On the complete 93-window, occupied-support, clip-to-zero diagnostic, NeuFSD and Elastic have mean WMRD 0.280415 and 0.177637, and mean MRD 0.136977 and 1.172420, respectively. This illustrates the distinct emphases of the two metrics. The other baseline values and the full-union/undefined cases are retained. Flow-key adapters, heavy auxiliary state, and sketch-seed replication are specified separately when interpreting this diagnostic.

Update cost and memory. Four fixed-window GPU profiles cover Avg5 adaptation at block 5 of IMC, MAWI, and CAIDA2018, and a separate IMC decoder-memory resource probe. The Avg5 profiles use $q = 0.1$, 2,000 examples per network split 1,800/200, five epochs, batch 256, and learning rate 0.01. The decoder-memory probe uses $q = 0.5$, one replica with counter seeds 0–1999, the same split/batch/learning rate, and 20 epochs; it initializes from the supplementary synthetic-only prior. The software and precision settings are given in Appendix D; CPU affinity covers four cores, with four Torch threads.

Both networks remain resident while their updates run sequentially. Adam moments exist only for the active network and are released after its training-and-holdout call; final gradients remain attached to both models through inference. The pair’s FP32 parameters occupy 695,811,976 bytes (663.578 MiB), with an equal-sized full gradient set. The active network’s first- and second-moment tensors occupy 691,527,696 or 700,096,256 bytes in total. Across the four profiles, maximum phase-level PyTorch allocated and reserved memory are 4.25094 and 4.81445 GiB. Peaks are measured per phase, include temporary tensors, and are not summed. An independently tracked 12 GiB device reservation is excluded, so these values describe the workload’s allocator footprint. The instrumented sequential-model training phases take 11.6906 and 11.6260 s, including cached training-data reads; holdout and initialization are separate phases.

For ordinary update latency, the fixed IMC Avg5 configuration is repeated without the phase observer: two warmups followed by seven retained updates, each starting with the same pretrained pair and random state. Median two-model update time is 5.0115 s (range 4.9672–5.0233 s). This interval includes cached CPU data reads, both updates, and holdout, and excludes sample-cache/counter construction, model loading, and inference. Instrumented and ordinary updates yield identical final weights in this check.

Single-sketch and batch decoding. Both complete models from the supplementary IMC scratch run are loaded before timing. We use ten warmups and fifty retained repetitions, synchronize GPU calls around host-wall measurements, and retain all samples. Parameters are stored in FP32; TF32 matmul/cuDNN and cuDNN benchmarking are enabled. CPU inference uses four PyTorch threads on the shared host, without a fixed CPU affinity.

For a single input already resident on the target device, median two-model forward time is 10.498 ms on RTX 4090 and 82.939 ms on CPU. Starting from prepared host images, the public inference helpers take 12.326 ms and 85.771 ms, including their loader, transfer, and output-copy operations. Single-input preparation from in-memory counters separately takes 0.044 ms and 0.036 ms. On GPU, the resident two-model batch-400 call takes 51.713 ms, and the helper takes 68.403 ms. Each value describes its measured interval; separately measured medians are not summed into an end-to-end latency. Model loading, packet export, insertion, adaptation, and heavy-tail merging/calibration lie outside the forward measurement.

Paired forward measurements and reference configuration. A separate batch-one measurement uses the 10/1,080-output model pair initialized from the synthetic-only prior, with one common GPU-resident input, evaluation mode, no gradients or autocast, ten warmups, and fifty retained observations per setting. Synchronized two-model wall time, including the synchronization between model calls, has median 10.280 ms with TF32 allowed and 10.179 ms with TF32 disabled on RTX 4090. Preprocessing, transfer, heavy merging, calibration, and adaptation are outside this interval.

The original reference forward microbenchmark uses a separate 10/9,990-output pair: 86,439,426 and 96,429,406 parameters, respectively. It uses PyTorch's 12-layer TransformerEncoder with width 768, 12 heads, feed-forward width 3,072, pre-LayerNorm with $\epsilon = 10^{-5}$, regression dropout zero, and CLS selection without a final backbone LayerNorm. Its patch projection and 17-token input have the same dimensions as Table 3. The recorded mean pair costs are 4.044 ms on H800 and 23.835 ms on CPU with 112 Torch threads, using random weights and a resident random input; GPU timing averages 120 repetitions after 20 warmups, and CPU timing averages 20 after five warmups. These reference-model measurements and the evaluated 10/1,080-output decoder measurements identify distinct workloads.

CPU query costs. On fixed IMC block 97, all methods receive the same four-core allowance on the Xeon host; PyTorch uses four intra-op threads and one inter-op thread, while the native Elastic and MRAC implementations are serial. After two warmups, seven retained queries give median times of 86.758 ms for the raw neural decoder, 2,651.458 ms for Elastic (one EM iteration), and 42,689.442 ms for MRAC (ten EM iterations). Each baseline is inserted once at nominal 16 KiB; repeated queries run in fresh children forked from its unchanged sketch snapshot. Fork/setup, insertion, input loading, and scoring are excluded; native allocation and logging inside the query remain timed.

The neural interval uses the final scratch-run model pair and counter seed zero. It includes conversion of the in-memory counters to three views, both public inference helpers, heavy interpolation and splicing, rounding, and clipping. Its exact-heavy histogram and occupied-size support through 10,000 are already available; their construction, model loading, sampling, adaptation, and calibration are excluded. Resident model forward alone takes 84.139 ms under this protocol. Baselines return their native distributions and supports, including sizes above 10,000. The neural path includes full-stream counters, exact-heavy assistance, and both resident networks, while baseline auxiliary state follows its implementation. These component timings therefore retain different resource and output scopes. All repetitions preserve identical results within each fixed model/sketch state; they measure repeated query cost on one window.

Insertion and mass accounting. The separate 64-bit-key deployment prototype with $w = 1000$, $E = 8$, and $m = 4096$ has median CPU insertion throughput 59.16 Mpps over seven repetitions on IMC window 5, including its key adapter. Snapshot checks on windows 5, 14, and 97 independently reconstruct every residual counter and verify that retained-hot plus residual packet mass equals the window's one million packets. Residual keys above 100 and partially retained hot flows are accounted for in the splice and target-mass terms of Section 4. These checks concern the insertion prototype and complement the full-counter neural replays.

Received 10 July 2026; revised 5 October 2026; accepted 9 October 2026