Dynamic SLA-aware Network Slice Monitoring

NILOY SAHA, University of Waterloo, Canada
MINA TAHMASBI ARASHLOO, University of Waterloo, Canada
NASHID SHAHRIAR, University of Regina, Canada
RAOUF BOUTABA, University of Waterloo, Canada

Next-generation networks increasingly rely on *network slices* — logical networks tailored to specific application requirements, each with distinct Service-Level Agreements (SLAs). Ensuring compliance with these SLAs requires continuous, real-time monitoring of end-to-end performance metrics for each slice, within a limited telemetry budget. However, we find that existing solutions face two fundamental limitations: they either lack end-to-end visibility (e.g., sketches, probabilistic sampling) or provide visibility but lack the control mechanisms to dynamically allocate monitoring resources according to slice SLAs.

We address this through a formal framework that reframes slice monitoring as a *closed-loop control problem*, and defines the minimal data plane requirements for SLA-aware slice monitoring via a *telemetry primitive contract*. We then present *SliceScope*, a realization of this framework that combines: (1) a control strategy that dynamically allocates the monitoring resources across diverse slices according to their SLA criticality, and (2) a data-plane based on change-triggered INT that provides per-packet end-to-end visibility with tunable accuracy-overhead trade-offs, satisfying the telemetry contract. Our evaluation results on programmable switches and in large-scale simulations with a mixture of different slice types, demonstrate that SliceScope *tracks critical slices up to* 4× *more accurately* compared to static baselines, while showing that *change-triggered INT outperforms alternative primitives* for realizing the telemetry primitive contract.

1 Introduction

A network slice is an isolated logical network that provides specific capabilities and performance characteristics to an application or service over a shared physical network. These guarantees are typically expressed through Service Level Agreements (SLAs) over metrics such as throughput, latency, jitter, and loss. Modern networks can have dozens to hundreds of concurrently active slices [4, 5, 8], each with distinct SLAs. For example, in the context of 5G networks, Ultra Reliable Low Latency Communication (URLLC) slices are expected to provide very low loss rates (e.g., $< 0.001\%$) and end-to-end latency (e.g., 1–5 ms) but have relatively flexible bandwidth requirements (e.g., 1–10 Mbps) [1–3]. Enhanced Mobile BroadBand (eMBB) slices, by contrast, have higher bandwidth requirements (e.g., up to 1000 Mbps) but tolerate higher packet loss (e.g., $< 1\%$) and end-to-end latency (e.g., 10–50 ms) [1–3].

To ensure compliance with these SLAs, network operators need to *continuously* monitor, in *real time*, the relevant SLA metrics for *every active slice*. This is essential for timely detection of SLA violations – operators must know immediately if the per-packet latency of a latency-critical slice exceeds the acceptable threshold to trigger appropriate mitigation actions. To see why this is challenging, consider a 5G network that can support 500 active slices. While slices can be broadly categorized into the URLLC and eMBB types, each instance comes with its own unique SLA that specifies its concrete requirements in terms of end-to-end latency, loss rate, and throughput. Moreover, the set of active slices *dynamically changes over time* – service providers using the 5G network as their substrate can request new slices to be created or existing ones to be removed as needed. The operators of the 5G network need to monitor each slice to ensure that its traffic, in aggregate, meets the unique performance requirements specified in that slice's SLA.

Authors' Contact Information: Niloy Saha, University of Waterloo, Canada, n6saha@uwaterloo.ca; Mina Tahmasbi Arashloo, University of Waterloo, Canada, mina.arashloo@uwaterloo.ca; Nashid Shahriar, University of Regina, Canada, nashid.shahriar@uregina.ca; Raouf Boutaba, University of Waterloo, Canada, rboutaba@uwaterloo.ca.

For instance, if 10 out of the 500 slices have very strict SLAs for end-to-end (e2e) latency, the operator could decide, for example, to use a packet-level telemetry mechanism such as In-band Network Telemetry (INT) [31], which embeds measurement data within packet headers, to monitor end-to-end latency. They could then collect INT information on every packet belonging to those 10 slices but less frequently for the rest, keeping the total overhead within budget.

Now, suppose the set of active slices changes such that 100 out of the 500 slices have very strict SLAs for e2e latency, and collecting telemetry information for every packet of all these 100 slices would exceed the total monitoring budget. In this case, the best strategy to allocate this limited budget could be to collect telemetry information from critical slices at a "base" frequency, but adjust it per slice if e2e latency fluctuations in a particular slice require more frequent INT collection. That is, in the presence of a dynamically changing set of slices, there is no "one-size-fits-all" monitoring strategy – as the set of active slices, their SLAs, and their traffic patterns change, the monitoring system needs to continuously revisit how it allocates its monitoring budget among the active slices in the network to effectively detect SLA violations in real time.

In this paper, we ask: *How can one design a real-time SLA monitoring system that dynamically allocates monitoring resources across diverse network slices according to their SLAs?* We model the monitoring system as a *closed control loop* that continuously observes the set of active slices, their SLAs, and relevant traffic metrics, and *tunes* the data plane to collect the "right" amount of telemetry for each slice. The first key challenge is to formalize the underlying resource-allocation problem in a way that is *independent* of the specific data-plane mechanism used for telemetry collection. This abstraction allows us to reason systematically about control-plane design and identify the minimal properties that a data-plane primitive must satisfy to enable slice- and SLA-aware closed-loop control of monitoring resources.

At a high level, our formulation introduces a "tuning" variable $k_{s,m}$, defined per slice s and SLA metric m, along with two functions, $E(k_{s,m})$ and $\Gamma(k_{s,m})$, that capture the monitoring error and overhead, respectively, for a given choice of $k_{s,m}$. To make effective allocation decisions, the control plane must accurately estimate E and Γ, which depend on both the monitoring primitive and the characteristics of the active traffic in the network.

This leads to the second challenge: estimating these functions accurately is difficult since their behavior varies with both the primitive and traffic dynamics. For example, the bandwidth overhead of change-triggered selective INT – which collects telemetry information only when metric fluctuations exceed a threshold [10, 27] – depends on the variability of the monitored metric, which in turn depends on the characteristics of interacting traffic streams in a network (e.g., extent and timing of bursts). Similarly, in sketch-based monitoring primitives, the collision probability that affects measurement accuracy depends on the active traffic streams [11].

As such, in practice, E and Γ must be continuously learned from the data-plane observations. This, however, is challenging since, at any moment, only one configuration of $k_{s,m}$ values is deployed in the data plane, while the control loop needs estimates of E and Γ across a *range* of configurations to choose the most effective operating point.

Finally, the third challenge concerns the design of the data-plane primitive itself. Because E and Γ define a trade-off space between accuracy and overhead that the controller navigates per slice, the primitive must (1) expose a sufficiently wide range of operating points within this space, (2) support configurability at slice granularity, and (3) allow transitions between these operating points at runtime. As an extreme example, a primitive that always samples one out of ten packets would be unsuitable, as E and Γ would have the same value regardless of $k_{s,m}$, slice s, and SLA metric m. We discuss these properties in detail in §3.3.

To demonstrate the feasibility of addressing these challenges and the benefits of slice-aware and SLA-aware closed-loop control, we design and implement an INT-based monitoring system, called

SliceScope, that realizes this abstraction in practice. We choose an INT-based data plane because slice monitoring in modern networks requires *continuous, real-time* measurement of per-packet performance metrics, rather than aggregate statistics such as percentiles or averages. Moreover, SLA metrics like latency, jitter, and loss are increasingly defined *end-to-end*, rather than per-hop [1–3]. INT enables network switches to add and update performance metrics to individual packets as they traverse the network [31], exposing per-packet end-to-end performance metrics. As such, it is well-suited for satisfying these requirements. INT, however, is known to incur non-negligible bandwidth overhead [6]. As such, *SliceScope* opts for a change-triggered selective INT approach that enables trading off telemetry collection frequency (i.e., bandwidth overhead) against monitoring accuracy, while providing the flexibility and granularity required for slice-aware and SLA-aware closed-loop control. While we focus on an INT-based implementation in this paper, the control framework is general and could be extended to other data-plane monitoring mechanisms, such as probabilistic INT or sketches, which we leave for future exploration.

Summary of contributions. This paper makes the following contributions:

- We formalize SLA-aware slice monitoring as a closed-loop control problem (§3), by introducing an abstract optimization framework parameterized by tuning knobs $k_{s,m}$ and trade-off functions $E(k_{s,m})$ and $\Gamma(k_{s,m})$. This formulation is independent of the specific telemetry primitive and identifies the minimal requirements a data-plane primitive must satisfy, which we term the Telemetry Primtive Contract (TPC).
- We present *SliceScope* (§4), a system that realizes our framework by implementing the TPC with change-triggered INT. *SliceScope* introduces: (1) per-slice state tracking for slice-level control, (2) composable end-to-end semantics for bounded error guarantees, and (3) mathematical models that enable the control plane to estimate the $E(k_{s,m})$-$\Gamma(k_{s,m})$ trade-off for different knob configurations in the design space.
- We demonstrate the feasibility and benefits of slice-aware and SLA-aware closed-loop control by implementing *SliceScope* on Intel Tofino switches and evaluating it on a testbed with emulated 5G traffic and through large-scale simulations with 300 slices exhibiting diverse SLA targets and traffic patterns (§5).

Evaluation highlights. We demonstrate that *SliceScope*'s adaptive control is essential: while change-triggered INT provides the foundation, static thresholds alone cause frequent SLA violations. Our control-plane reduces violations by up to 4× for critical slices under comparable overhead. We also validate that change-triggered INT best realizes the TPC, outperforming alternatives like PINT [6] and LightGuardian [33] for end-to-end tracking of slice SLAs, and perform microbenchmarks and test-bed evaluations. *This paper does not raise ethical issues.*

Prior work. To the best of our knowledge, we are the first to formulate slice monitoring as a closed-loop control problem and define the *Telemetry Primitive Contract (TPC)* that specifies the minimal data-plane requirements for such control. In §4.1, we revisit prior telemetry mechanisms, and explain why change-triggered INT provides the best basis to realize the TPC in practice.

2 Context: SLA Diversity in 5G and Beyond

SLA diversity in the real world: 5G slicing. While our work applies broadly to any network supporting slicing, we ground our discussion in 5G networks, where slicing is a core architectural primitive. 5G networks are expected to support dozens to hundreds of active slices [4, 5, 8], each with distinct SLAs. There are three broad slice categories: Ultra Reliable Low Latency Communications (URLLC), Enhanced Mobile Broadband (eMBB), and Massive Machine Type Communications (mMTC) (see Table 4). However, real-world deployments require more granularity. For example, within the broad URLLC category, discrete automation needs 10ms latency and 10Mbps throughput, while process automation needs 60ms latency and just 1Mbps throughput [3]. Therefore, operators typically provision many slices, as opposed to just one for each broad category, each with distinct, fine-grained SLAs that must be continuously monitored in real-time for effective service assurance [5, 20]. Other examples of slicing in modern networks include cloud providers providing service-specific connections with distinct performance guarantees (e.g., Azure ExpressRoute [23], Google Dedicated Interconnect [14]), and wide-area networks providing differentiated services [12].

Why monitor per slice (as opposed to per flow)? From a network operator's perspective, the natural monitoring unit is a *slice* – the level at which resources are provisioned and SLAs are defined. Slice-level monitoring offers two key advantages.

- *Alignment with SLAs*: Slices are the natural unit of SLA definition and resource provisioning. Operators provision resources at slice granularity and commit to slice-level QoS guarantees, making slice-level monitoring the natural level for SLA verification and control decisions. The responsibility of managing *individual flows within a slice* typically rests with the service provider using the slice.
- *Tractability*: Although a network may have millions of flows, the number of active slices is typically in the hundreds [4, 5, 8]. This makes dedicated and accurate (as opposed to approximate) per-slice state manageable in programmable switches. This also makes closed-loop optimization of resource allocation computationally tractable – our evaluations (§5.4) show that joint optimization of monitoring resources over 300 slices completes in < 1s, which enables real-time adaptation.

3 Formalizing closed-loop SLA-aware slice monitoring

As shown in figure 1, we model the slice monitoring system as a closed control loop that continuously takes in the set of active slices, their SLAs, and relevant information about their traffic patterns and observed SLA metrics, and tunes the network data plane to collect the "right" amount of telemetry information for each slice. In this section, we describe how we formalize the resource allocation problem that the control loop must solve independent of the particular data-plane monitoring mechanism used to collect telemetry information (§3.1), how it can adapt to changing network conditions (§3.2), and its implications on the properties needed from the data-plane monitoring primitives (§3.3).

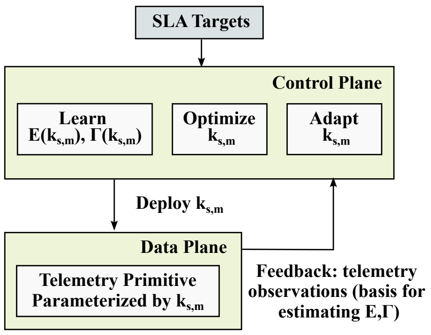
SLA Targets
Control Plane
Learn E(k_{s,m}), Γ(k_{s,m})
Optimize k_{s,m}
Adapt k_{s,m}
Deploy k_{s,m}
Data Plane
Telemetry Primitive Parameterized by k_{s,m}
Feedback: telemetry observations (basis for estimating E,Γ)

Fig. 1. *Closed-loop formulation of SLA-aware slice monitoring.* The control plane continuously learns from telemetry observations that reflect slice-level performance, estimates monitoring error $E(k_{s,m})$ and overhead $\Gamma(k_{s,m})$, and tunes per-slice knobs $k_{s,m}$ of the data plane telemetry primitive to satisfy SLA objectives under resource constraints.

3.1 Joint Multi-Slice Resource Allocation

SLA model and monitoring error tolerance. We model a slice SLA for a metric m by its target value α (e.g., max latency of 10ms). Critical slices (e.g., URLLC) have stringent targets (e.g., α=5ms) while non-critical slices (e.g., mMTC) have more relaxed targets (e.g., α=40ms). Slice SLAs specify targets for end-to-end metrics but not acceptable *monitoring* error, despite its acknowledged impact [12].

Slice monitoring as a joint resource allocation problem. Each slice–metric pair operates within its own accuracy–overhead trade-off space: allocating more monitoring resources (e.g., computational resources in network devices or telemetry bandwidth) yields higher measurement

Objective	$\min_K \sum_{s,m} \lambda E(k_{s,m}) + (1-\lambda)\Gamma(k_{s,m})$
Constraints	
C1	$\forall s, m : E(k_{s,m}) \leq \epsilon_{s,m}$
C2	$\forall s, m : \sum_c z_{s,m,c} = 1$
C3	$\forall s, m, c : z_{s,m,c} \in \{0, 1\}$
C4	$\forall s, m : K_{s,m} = \sum_k z_{s,m,c} \cdot K_{s,m,k}$

(a) Optimization formulation

Symbol	Description
$E(k_{s,m})$	Expected monitoring error for slice s and metric m given $k_{s,m}$ as tuning parameter
$\Gamma(k_{s,m})$	Expected telemetry overhead for slice s and metric m given $k_{s,m}$ as tuning parameter
$\epsilon_{s,m}$	SLA-specific monitoring error tolerance for slice s and metric m
λ	Trade-off between accuracy and overhead
$K_{s,m,c}$	Candidate "knob" values
$z_{s,m,c}$	Binary variable selecting $K_{s,m,c}$

(b) Variable definitions

Table 1. SLA-aware multi-slice joint resource allocation

accuracy for that metric in that slice. However, as we show in our evaluation (§5.2), independent per-slice optimization is suboptimal; *joint optimization* enables coordinated trade-offs where slices with error-budget slack can accept reduced fidelity while near-constrained slices receive tighter monitoring. As such, for a network with N active slices and M SLA metrics, the monitoring system must manage $N \times M$ "instances" of this trade-off space while balancing per-SLA monitoring accuracy with network-wide monitoring cost.

We model this as a constrained resource allocation problem, summarized in Table 1. Specifically, our formulation captures the accuracy-overhead trade-off space in a unified manner with two functions, $E(x)$ and $\Gamma(x)$, that are connected through *the "tuning" parameter x*. This provides our formulation with a unified way to explore the $N \times M$ instances of the trade-off space using dedicated tuning variables for each slice and metric. Concretely, we define a "tuning" variable $k_{s,m}$ for each slice s and SLA metric m, along with two functions, where $E(k_{s,m})$ and $\Gamma(k_{s,m})$ capture the monitoring error and overhead, respectively, for a given choice of $k_{s,m}$. The goal is to select configurations $\{k_{s,m}\}$ that minimize system-wide cost – sum of $\Gamma(k_{s,m})$ across all slices and metrics – while respecting all constraints with respect to SLA monitoring accuracy, i.e., $E(k_{s,m}) \leq \epsilon_{s,m}$.

Objective: balancing accuracy and overhead among acceptable joint solutions. The objective is to minimize both expected monitoring error and telemetry overhead across all slices and metrics. Note that, as we discuss below, our formulation enforces that for each slice s and metric m, the monitoring error $E(k_{s,m})$ should not exceed the acceptable threshold $\epsilon_{s,m}$ through dedicated constraints. The objective includes a tunable weight λ that operators can adjust to express policy preferences. Specifically, λ allows operators to balance accuracy and overhead among all valid solutions: larger λ favors more accurate solutions, whereas smaller λ favors those with lower network-wide overhead. Moreover, including error in the objective (in addition to respecting per-slice tolerances) can prevent undesirable solutions, e.g., highly unbalanced ones that would otherwise just barely satisfy the monitoring requirements of critical slices.

Accuracy and $k_{s,m}$ selection constraints. Each slice s has an error tolerance $\epsilon_{s,m}$ for each metric m, with critical slices having lower tolerances than non-critical ones. Constraint C1 enforces that chosen $k_{s,m}$ values respect these tolerances. Moreover, our formulation selects $k_{s,m}$ from a discrete set of predefined candidate values. Each $k_{s,m}$ represents a distinct operating point in the accuracy–overhead trade-off of the data-plane monitoring primitive. While some primitives may conceptually support a continuous range of $k_{s,m}$ values (§4), real-world programmable switches (e.g., Intel Tofino [17]) typically implement them using fixed-point arithmetic with a configurable step size, yielding a finite set of realizable values. For example, with an 8-bit fixed-point representation and step size 0.05, $k_{s,m}$ can take on 256 possible values between 0 and 12.75.

To generalize across different data-plane primitives and reflect these hardware constraints, we model the candidate values as $K_{s,m,c}$ and use binary variables $z_{s,m,c}$ to indicate which one is selected. Constraints C2–C4 ensure that exactly one candidate value is chosen per slice–metric pair (s, m).

The challenge of learning trade-off functions. To make effective allocation decisions, the control plane must accurately estimate the "trade-off functions" E and Γ, which depend on both the monitoring primitive and the characteristics of the active traffic in the network. For example, the bandwidth overhead of change-triggered selective INT – which collects telemetry information only when metric fluctuations exceed a threshold [10, 27] – depends on the variability of the monitored metric, which in turn depends on the characteristics of interacting traffic streams in a network. Similarly, in sketch-based monitoring primitives, the collision probability that affects measurement accuracy depends on the active traffic streams [11]. As such, E and Γ must be continuously learned from the data-plane observations.

What makes this challenging is that the control plane needs to construct models for predicting how *arbitrary candidate values* of $k_{s,m}$ affect monitoring accuracy E and overhead Γ from *limited observations* of a data plane that is configured with *one set of concrete $k_{s,m}$ values* at that point in time. We cannot simply try out different $k_{s,m}$ configurations. Trying suboptimal settings risks SLA violations during exploration. Moreover, with hundreds of slices, various metrics, and large parameter spaces, exhaustive exploration is impractical. As such, the control plane must generalize from limited observations without requiring deployment of every candidate in the data plane. As mentioned above, E and Γ also depend on the particular data-plane monitoring primitive used in the control loop. We describe how *SliceScope*, our realization of the closed-loop control with change-triggered INT as its data-plane primitive, addresses this challenge and estimates these functions in §4.2.

3.2 Adapting to Network Dynamics

The formulation in §3.1 provides optimal $k_{s,m}$ values for each slice and metric. However, its effectiveness depends on E and Γ, which can very over time with changing network conditions. As such, we divide time into *epochs of τ seconds*, and poll the data plane at the beginning of each epoch to update the learned estimates of E and Γ, and subsequently re-run the joint optimization (§3.1) to recompute $k_{s,m}$ values. The choice of epoch length τ is a trade-off: it must be short enough to capture network dynamics, yet long enough for the control plane to compute feasible thresholds. We evaluate the impact of τ in §5.4.

Update strategies. We need to update the $k_{s,m}$ values within bounded per-epoch computation time. What makes this challenging is that optimization may take a long time to converge, requiring approximate methods or fallbacks to ensure timely decisions (§3.2). Our formulation is an Integer Linear Program (ILP), which are NP-complete in the general case. As such, exact optimal solutions may take exponential time in the worst case. To ensure robustness in the face of uncertain optimization times and enable timely adaptation, we use a two-tiered approach:

1. ILP with early stopping. We limit solver runtime to a fraction of the epoch length τ, to ensure that a solution is found in bounded time. Even if the global optimum is not reached, the best current solution is validated for feasibility before being deployed. Our evaluation shows that early stopping provides feasible solutions for τ as small as 5 seconds (§5.4).

2. Greedy heuristic fallback. If the ILP solver fails to find a feasible solution within the allotted time, we switch to a lightweight greedy heuristic. The heuristic iterates over slices in order of criticality (e.g., URLLC first). For each slice-metric pair, it selects the set of candidate $k_{s,m}$ values (out of the discrete set of possible values) that satisfy that pair's monitoring error tolerance (based on E). If multiple candidates meet this criterion, the one with the lowest expected overhead is chosen. When no candidate satisfies the tolerance, the heuristic defaults to the most conservative $k_{s,m}$ value (i.e., smallest error) to ensure SLA safety. This greedy approach prioritizes constraint satisfaction over the error-overhead trade-off: it ensures all $\epsilon_{s,m}$ bounds are met, then minimizes overhead among valid solutions, rather than jointly balancing the two objectives via λ. This preserves SLA guarantees while enabling timely decisions when full optimization is infeasible.

3.3 Requirements from the data-plane telemetry primitive

For the joint multi-slice optimization to be effective, we need a data plane primitive with particular capabilities in the closed control loop. Inspired by our general formulation of the problem in §3.1 and §3.2, we derive a minimal set of such capabilities, which we call the *Telemetry Primitive Contract (TPC)* and describe in this section.

(R1) Per-slice per-metric tunability. The control-plane optimization relies on navigating the accuracy-overhead trade-off space – through E and Γ – *for each slice*. As such, the data-plane telemetry primitive should expose *per-slice, per-metric runtime knobs* that the control plane can adjust each epoch, enabling differentiation across slices with heterogeneous SLAs. Moreover, these knobs should offer a wide range of operating points in this trade-off space, i.e., with a wide variety of values for E and Γ. A primitive that, say, always samples one out of ten packets would be unsuitable, as E and Γ would have the same value regardless of $k_{s,m}$, slice s, and SLA metric m.

(R2) Run-time tunability. Not only should the data plane primitive offer a wide range of operating points in the accuracy-overhead trade-off space, it should allow transitions between them *at runtime*. That is, to be able to do closed-loop control, the control plane should be able to adjust data-plane knobs, and in turn, monitoring and accuracy for each slice-metric pair without having to recompile the data plane.

(R3) Robust and composable E2E Semantics: To allow the control plane to reason about slice-level SLA monitoring fidelity, we need a data-plane primitive with E2E telemetry semantics whose local measurements combine into a predictable end-to-end accuracy (E) and overhead (Γ) view (through strict analytical bounds or calibrated proxies), allowing the control plane to reason about slice-level SLA monitoring fidelity. This is because modern SLAs are *real-time* and increasingly defined *end-to-end*, as opposed to per-hop [1–3]. Tracking SLA metrics such as latency, jitter, and loss per-hop (as statistics or quantiles) is not effective for real-time compliance: without per-packet correlation across hops, we cannot tell whether observations at different hops affect the same or different packets. For example, suppose a discrete automation application with an end-to-end latency SLA of 15ms runs in a URLLC slice instance, and we find out 50% of packets exceed 10ms at switch A, and 50% exceed 5ms at switch B. Without per-packet correlation, we cannot tell whether these are the *same* packets accumulating delay across hops, or distinct packets encountering independent issues. Only the former indicates an SLA violation. As such, fine-grained (e.g., per-packet) end-to-end visibility is key to providing robust and composable E2E semantics that can be used as part of continuous closed-loop control.

Summary. The TPC captures the minimal high-level requirements for a data plane telemetry primitive for SLA-aware network slice monitoring, without prescribing *how* a primitive satisfies these requirements. In *SliceScope* (§4), we present one realization of the TPC using change-triggered selective INT; however, alternative primitives could satisfy it through different means.

4 *SliceScope*: Realizing the Closed Loop with Change-Triggered INT

Building on §3, *SliceScope* realizes the real-time SLA-aware closed-loop control using a change-triggered INT-based telemetry in the data-plane. We first explain why change-triggered INT is uniquely well-suited for this framework (§4.1), then develop the mathematical models for estimating $E(k_{s,m})$ and $\Gamma(k_{s,m})$ (§4.2), describe the data plane implementation (§4.3), and discuss additional mechanisms needed for practical deployment (§4.4).

	Data Plane Requirements (§3.3)		SLA-aware Closed-Loop Control (§3.1 & §3.2)		
	Slice-Level Run-Time Tunability (R1-R2)	E2E Semantics (R3)	Trade-off Learning	Multi-Slice Optimization	Runtime Adaptation
No e2e per-packet visibility					
Active probing [7, 16]	✗	✗	✗	✗	✓
Sketches [19, 21, 33]	✗	✗	✗	✗	✗
Probabilistic [6, 29]	✓	✗	✗	✗	✓
E2e visibility, but lacking control					
Path-aggregated [22]	✗	✓	✗	✗	✗
Change-triggered [10, 27]	~	~	✗	✗	✗
SliceScope (this work)	✓	✓	✓	✓	✓

Table 2. Comparison of representative telemetry approaches against data-plane requirements (§3.3) and closed-loop control challenges (§3.1 and §3.2). Only *SliceScope* satisfies all criteria. ✓ supported, ✗ not supported, ~ partial support.

Generalizability. Although we instantiate the TPC abstraction using change-triggered selective INT, the abstraction itself is not tied to any specific primitive. Other telemetry mechanisms, such as future probabilistic or hybrid designs could satisfy the same contract if extended with equivalent slice-level control and e2e semantics. Together, our closed-loop control formulation (control-plane) and TPC (data-plane) provide a blueprint for any dynamic, SLA-aware monitoring system.

4.1 Why change-triggered In-Band-Network Telemetry (INT)?

Existing telemetry mechanisms fall short. Section §3.3 defined the abstract Telemetry Primitive Contract in the form of three requirements R1-R3. These requirements rule out several classes of monitoring approaches. *Active probing* uses synthetic probes to collect telemetry for e2e paths. However, the measurements are for the probes themselves, not actual slice traffic. Probes can experience different treatment (e.g., ECMP path, queueing/scheduling, policing), so their telemetry is not representative of per-slice behavior; consequently, it fails to meet the TPC (R1–R3). *Sketch-based* approaches (e.g., [21, 33]) allow recording traffic summaries and memory-efficient estimates of statistics such as top-k heavy hitters or latency quantiles at a single switch. However, they cannot correlate observations across packets to reconstruct end-to-end behavior (R3); our evaluation (§5.3) shows this yields insufficient accuracy for SLA tracking. Moreover, they are typically not sufficiently tunable at run-time (R2). *Probabilistic sampling* [6] aims at reducing overhead by instrumenting a subset of packets according to a configured sampling probability. While it is possible to adapt the sampling rate per-slice, probabilistically spreading telemetry across packets breaks the packet-path continuity needed for strict e2e semantics (R3). As we show in §5.3, this loss of continuity leads to insufficient accuracy for SLA tracking.

INT-based approaches provide the necessary per-packet, end-to-end correlation. *Path-aggregated* INT [22] carries complete e2e metrics in fixed size packet headers, ensuring full visibility but at fixed overhead with no run-time per-slice control (R1 and R2). *Change-triggered* INT [10, 27] is more promising: it inserts telemetry only when metrics deviate by more than threshold Δ, naturally exposing a control knob. However, existing designs apply Δ globally and statically, and for one concrete metric, providing no concrete pathway to per-slice run-time adaptation and for a range of end-to-end common SLA metrics. Thus our data plane requirements are only partially satisfied (~R1–R3). Moreover, the existing designs all lack any form of SLA-aware closed-loop control.

Table 2 summarizes these limitations against our data-plane requirements (R1–R3), as well as various components of our closed-loop control addressed by *SliceScope* and not by prior work.

SliceScope extends change-triggered INT to address these gaps. We make Δ a per-slice, per-metric knob $\Delta_{s,m}$, add mechanisms for composable end-to-end semantics, and enable the control plane to learn trade-offs, allocate thresholds across slices, and adapt them at runtime. This provides the missing link: a controllable primitive that enables the closed loop of learning, resource allocation, and adaptation required for real-time, SLA-aware monitoring.

4.2 Mathematical Framework for Modeling E and Γ

Section §3.1 identified that the control plane must estimate $E(k_{s,m})$ and $\Gamma(k_{s,m})$ from limited observations, a challenge whose solution depends on the specific telemetry primitive. 5 In *SliceScope*, our telemetry primitive is changed-triggered INT. Specifically, we define per-slice, per-metric knob $\Delta_{s,m}$, and only add INT information to the packet only when metric m for slice s deviate by more than threshold $\Delta_{s,m}$ from previously collected value. We set $k_{s,m} \equiv \Delta_{s,m}$, and develop the mathematical models that enable this estimation for change-triggered INT. The core challenge is that only one threshold configuration, $\Delta_{s,m}$, is deployed in the data plane at any given time, yet the optimizer needs to evaluate many candidates to select the best one (§3.1). We address this by modeling the relationship between $\Delta_{s,m}$ and telemetry insertion decisions, which in turn, determines both monitoring accuracy and overhead.

Data plane operation (preview). Before developing the models, we briefly preview how change-triggered INT works (full description in §4.3): Each switch maintains per-slice state, including the last reported metric value for each metric. When a packet arrives, the switch computes the accumulated difference $\delta_{s,m}$ between the current metric and the last reported value. If $\delta_{s,m} > \Delta_{s,m}$, the switch inserts telemetry and resets $\delta_{s,m}$; otherwise, it skips insertion. This mechanism creates the knob: smaller $\Delta_{s,m}$ causes more insertions (higher accuracy, higher overhead), while larger $\Delta_{s,m}$ suppresses insertions, as shown in Figure 2.

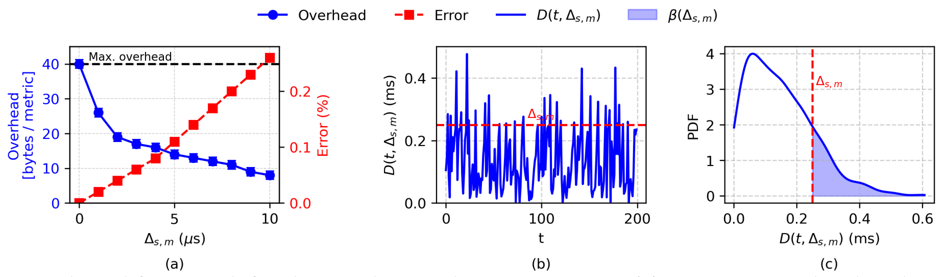

Fig. 2. Analytical framework for change–driven telemetry insertion. **(a)** Accuracy–overhead trade-off as a function of the telemetry threshold $\Delta_{s,m}$: increasing $\Delta_{s,m}$ reduces telemetry overhead (blue) but increases monitoring error (red). **(b)** Example trace of the simulated cumulative metric difference $D(t, \Delta_{s,m})$, which mimics how $\delta_{s,m}$ accumulates in the data plane, and triggers telemetry insertion whenever it exceeds $\Delta_{s,m}$. **(c)** Probability density of $D(t, \Delta_{s,m})$, where the shaded region corresponds to the insertion probability $\beta(\Delta_{s,m})$.

Modeling telemetry insertion probability. Given this mechanism, the key is to quantify the relationship between $\Delta_{s,m}$ and insertion likelihood through the telemetry insertion probability: $\beta(\Delta_{s,m}) = \mathbb{P}(\delta_{s,m} > \Delta_{s,m})$. However, directly modeling $\delta_{s,m}$ creates a circular dependency: circular dependency: the accumulated difference $\delta_{s,m}$ depends on the last reported state, which itself depends on when telemetry was last inserted (i.e., when $\delta_{s,m}$ previously exceeded the threshold currently deployed). This circular dependency makes direct modeling of $\beta(\Delta_{s,m})$ impractical.

Learning underlying packet differences. Our key insight is that while we cannot directly model $\delta_{s,m}$, we can instead work with the underlying packet-to-packet differences that generate these deviations. Specifically, we learn the distribution of successive metric differences $d(t) = E^t_{curr} - E^t_{last}$ from the switch data plane, which capture how the metric changes between consecutive packets at time t. Unlike $\delta_{s,m}$, these packet differences are independent of the chosen threshold.

Using this learned distribution, we can simulate, in the control plane, how different threshold values would affect telemetry insertion. We do this by computing the cumulative sum $D(t, \Delta_{s,m}) = |\sum d(t)|$, resetting it whenever it reaches $\Delta_{s,m}$. This mimics exactly how $\delta_{s,m}$ accumulates and resets in the data plane. Thus, the cumulative sum D effectively serves as a proxy for $\delta_{s,m}$. This simulation enables us to construct the probability function for arbitrary thresholds. Figure 2 illustrates this process, showing how the telemetry insertion probability $\beta(\Delta_{s,m})$ is computed as area under the probability density of $D(t, \Delta_{s,m})$.

Having established how different thresholds determine telemetry insertion probabilities, we next examine how these probabilities affect monitoring accuracy along a path. Skipped insertions reduce overhead, but also allow errors to accumulate across switches. Quantifying and bounding this accumulation is critical to ensure SLA error tolerances are respected.

Modeling error accumulation. Consider packets from slice s traversing a path $\mathcal{P}$. Let η_i denote the absolute error between the true and measured value of metric m at switch i. At each hop $i-1$, telemetry is inserted with probability $\beta_{i-1}(\Delta_{s,m})$. The expected error therefore evolves as:

$$\mathbb{E}[\eta_i] = \mathbb{E}[\eta_{i-1}] + \left(1 - \beta_{i-1}(\Delta_{s,m})\right) \cdot \Delta_{s,m}.$$

Intuitively, when telemetry is inserted (probability β_{i-1}), the current switch simply inherits the previous error η_{i-1}. On the other hand, when it is skipped (probability $1 - \beta_{i-1}$), the error grows by up to $\Delta_{s,m}$ in addition to η_{i-1}.

Bounding error growth. A naive worst-case argument assumes every hop skips insertion, yielding an upper bound of $(|\mathcal{P}| - 1) \cdot \Delta_{s,m}$ (one potential $\Delta_{s,m}$ contribution per hop). However, insertions occur with probability $\beta_j(\Delta_{s,m})$ and reset the error. Unrolling the recurrence above (by induction on path length) gives:

$$E(k_{s,m}) = E(\Delta_{s,m}) = \mathbb{E}[\eta]_{UB} = \left((|\mathcal{P}| - 1) - \sum_{j=1}^{|\mathcal{P}|-1} \beta_j(\Delta_{s,m})\right) \Delta_{s,m}$$

This is a *β-discounted* (tighter) upper bound: it matches the naive bound exactly when $\beta_j = 0$ for all hops (no insertions), and is strictly smaller whenever any $\beta_j > 0$. Hence it avoids the overprovisioning implied by the naive bound. This is what we use for the monitoring error $E(k_{s,m})$ in the closed-loop control's optimization problem in §3.1.

Estimating per-packet monitoring overhead. Having established how errors accumulate, we next quantify the cost of telemetry in terms of expected header overhead. Each monitored packet consists of: (i) a per-packet shim and bitmap (b_0 bits), (ii) per-hop metadata that is always present (b_h bits per hop), and (iii) metric payloads (b bits each) included only when $\delta_{s,m} > \Delta_{s,m}$, i.e., with probability $\beta_j(\Delta_{s,m})$ at hop j (see §4.3 for details). The expected per-packet overhead for metric m on slice s is therefore

$$\Gamma(k_{s,m}) = \Gamma(\Delta_{s,m}) = \mathbb{E}[\gamma] = (b_0 + b_h) \cdot |\mathcal{P}| \; + \; b \sum_{j=1}^{|\mathcal{P}|} \beta_j(\Delta_{s,m}).$$

The first term reflects the fixed cost that scales linearly with path length, while the second term captures the conditional metric payloads and grows in proportion to the insertion probabilities. We use this as the monitoring overhead $\Gamma(k_{s,m})$ in the closed-loop optimization (§3.1).

4.3 Telemetry Data Structures and Operations

Having developed models for $E(\Delta_{s,m})$ and $\Gamma(\Delta_{s,m})$ (§4.2), we now describe how the data plane realizes change-triggered INT and extends it to satisfy the Telemetry Primitive Contract from §3.3.

Existing change-triggered INT designs apply thresholds globally and lack mechanisms for composable end-to-end semantics. *SliceScope* addresses these gaps through: (1) per-slice state tracking that enables slice-specific thresholds $\Delta_{s,m}$ (R1, R2), and (2) a two-part telemetry header with conditional insertion that provides bounded end-to-end error and overhead guarantees (R3).

We first describe the data structures, then walk through the four-step processing algorithm with examples for latency, jitter, and loss. Deployment mechanics (e.g., MTU headroom, multipath disambiguation, recovery) are addressed in §4.4.

Per-slice state tracking. Each switch maintains d bucket arrays, each with w buckets, to track telemetry state for active slices (Figure 3). On packet arrival, a key x including the slice ID is extracted from the packet headers. Next, for each bucket array A_i, an independent hash function $h_i(x)$ is used to map this packet to a bucket A_{ij} (Figure 3). Each bucket stores: (1) the key x, (2) three end-to-end values for a given metric — E_{prev} (estimate from previous hops), E_{rep} (last reported), and E_{last} (value from most recent packet), (3) auxiliary metrics V_{aux} needed for computation of some metrics (e.g., estimated flow size for computing packet loss), and (4) a 1-bit table miss flag F_{tm} to help recovery when downstream telemetry state is lost due to hash collisions. This per-slice state tracking enables slice-level customization of telemetry behavior (R1), as each slice can use its own threshold $\Delta_{s,m}$ to determine whether telemetry is inserted or skipped, at runtime (R2).

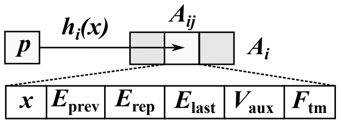

Fig. 3. Per-slice telemetry state stored in each bucket A_{ij} of the bucket arrays. Each bucket maintains the slice key x, e2e estimates ($E_{prev}, E_{rep}, E_{last}$), auxiliary state V_{aux} (e.g., counters for packet loss), and a miss flag for recovery.

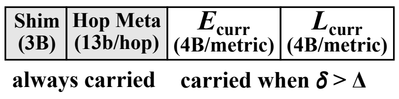

Fig. 4. Telemetry header format. The fixed part (always present) includes a shim and per-hop metadata. The conditional part (inserted only when $\delta_{s,m} > \Delta_{s,m}$) carries updated e2e metric values E_{curr} and auxiliary state V_{aux} for computing metrics such as jitter or packet loss.

Telemetry header. The telemetry header (Figure 4) has two parts: a *fixed part* which is always carried, and a *conditional part* only carried when significant changes occur (only when $\delta > \Delta_{s,m}$). The fixed part consists of (1) a telemetry shim with a bitmap indicating which selective telemetry fields are present, and (2) per-hop metadata containing a 10-bit node ID and three 1-bit anomaly flags. The conditional part consists of (1) current end-to-end values E_{curr} for the slice metric, and (2) any auxiliary metrics V_{aux} needed to compute that slice metric. For example, to compute end-to-end (e2e) packet loss, each switch appends the number of packets it forwarded for that slice (V_{aux}), allowing the next hop to estimate loss by comparing it with its own packet count.

Next, we show how the telemetry header and per-slice state are used to track per-slice SLAs in an end-to-end per-packet manner with bounded error (R3). *SliceScope*'s data plane algorithm runs in the switch egress pipeline, and processes incoming packets in the following steps:

Step 1 (Bucket array lookup). The switch computes the key x based on the packet headers, and looks it up in the bucket arrays. If a match is found, the switch retrieves the relevant telemetry state from previous packets, including E_{prev} (estimate from previous hop) and E_{rep} (last reported).

Step 2 (Metric computation). The metric computation step involves three key operations:

- *Compute the current per-hop value (L_{curr}):* This can be derived using either internal device state (e.g., hop latency) or auxiliary metrics (e.g., hop loss). Note that this value is different from the e2e metric values E_{curr}, which are aggregated across the slice path.
- *Update the previous hop estimate of the e2e metric (E_{prev}):* If the packet carries a new update from the previous hop, it will be used and stored as the new E_{prev}.

- *Compute the current e2e value (E_{curr}):* This is done by combining the (potentially updated) previous hop estimate E_{prev} with L_{curr} using a simple aggregation operation (e.g., sum or max).

Step 3 (Selective telemetry insertion). The switch compares the current e2e value E_{curr} with the last reported value E_{rep}. If the difference δ exceeds a given threshold $\Delta_{s,m}$ for that slice's metric, it is considered a significant change. The switch then inserts the current e2e value E_{curr} in the e2e metrics part of the packet header, and, when required, auxiliary information V_{aux} to support downstream metric computation. For example, to compute e2e packet loss across multiple hops, each switch inserts the number of packets it forwarded (V_{aux}), enabling the next hop to detect missing packets and estimate per-hop loss.
Step 4 (State update). If telemetry data is added to the packet, the switch updates the last reported e2e value E_{rep} in the corresponding bucket. We also update E_{last} to be E_{curr}.

Pkt	Sw	State Before	L_{curr}	E_{curr}	$\|\delta\|$ vs Δ	Action	State After
p1	s_1	E_{prev}=0, E_{rep}=0	5	$0 + 5 = 5$	$\|5-0\| = 5 \geq 2$	Add telemetry (5)	$E_{rep} \leftarrow 5$
p1	s_2	E_{prev}=5, E_{rep}=0	6	$5 + 6 = 11$	$\|11-0\| = 11 \geq 2$	Add telemetry (11)	$E_{rep} \leftarrow 11$
p2	s_1	E_{prev}=0, E_{rep}=5	4	$0 + 4 = 4$	$\|4-5\| = 1 < 2$	Skip telemetry	No change
p2	s_2	E_{prev}=5, E_{rep}=11	8	$5 + 8 = 13$	$\|13-11\| = 2 \geq 2$	Add telemetry (13)	$E_{rep} \leftarrow 13$

Table 3. Walkthrough on a 2-hop path ($s_1 \longrightarrow s_2$): for each packet, the switch computes the local observation L_{curr}, updates the end-to-end estimate E_{curr}, compares $\delta = |E_{curr} - E_{rep}|$ to $\Delta_{s,m}$, and inserts telemetry only on threshold crossings.

Walkthrough Example. Consider the simple case of a linear two-hop path with switches s_1 and s_2, where $\Delta = 2$ for both switches. Table 3 walks through how two packets (p_1, p_2) are processed at each hop. For packet p_1: At s_1, the local observation is $L_{curr} = 5$, giving an end-to-end estimate $E_{curr} = 0 + 5 = 5$. Since $\delta = |5 - 0| = 5 \geq 2$, telemetry is added and E_{rep} updates to 5. At s_2, the local observation is 6, updating E_{curr} to $5 + 6 = 11$. Since $\delta = |11 - 0| = 11 \geq 2$, telemetry is again inserted. For packet p_2: At s_1, $L_{curr} = 4$ gives $E_{curr} = 0 + 4 = 4$, but $\delta = |4 - 5| = 1 < 2$, so no telemetry is added. At s_2, $L_{curr} = 8$ updates E_{curr} to $5 + 8 = 13$. Since $\delta = |13 - 11| = 2 \geq 2$, telemetry is inserted.

The actual end-to-end latency for packet p_2 is $4 + 8 = 12$, but the reported estimate is 13 (using the stale $E_{rep} = 5$ from s_1). This creates an error of $|12 - 13| = 1$ at switch s_1, which is bounded by $\Delta_{s,m} = 2$. This example shows how selective insertion reduces overhead (one skipped report) while maintaining per-switch error bounds. Section 4.2 describes how these per-switch bounds can be combined to derive end-to-end error guarantees.

Metric-specific instantiations. The same four-step logic generalizes to different SLA metrics by adapting the definition of the local observation L_{curr} and any auxiliary state V_{aux}. For example:

- **End-to-end latency.** Each switch computes L_{curr} as the difference between its egress and ingress timestamps ($t_{eg} - t_{ig}$). The end-to-end estimate is updated additively as $E_{curr} = E_{prev} + L_{curr}$, requiring no auxiliary state.
- **End-to-end jitter.** Here L_{curr} is the change in per-hop latency relative to the previous packet. This requires storing the prior packet's latency as V_{aux}; the switch then computes $L_{curr} = L_{curr}^{latency} - V_{aux}$ and updates the end-to-end jitter estimate additively.
- **End-to-end packet loss.** Each switch maintains a per-slice packet counter and, when telemetry is inserted, exports its current counter value V_{aux} in the packet header (cf. Fig. 4). The downstream hop computes L_{curr} as the difference between the upstream counter and its own local count, and aggregates this into the cumulative end-to-end loss estimate ($E_{curr} = E_{prev} + L_{curr}$).

In all cases, the dataplane executes the same four-step loop; only the definitions of L_{curr} and V_{aux} differ by metric. These examples show how *SliceScope*'s threshold-based logic enables *end-to-end*, per-slice monitoring of *diverse SLA metrics* (latency, jitter, loss) with bounded error. We next address deployment aspects that preserve these semantics in practice (§4.4).

4.4 Deployment Considerations: Ensuring Boundedness & Stability

The data plane design (§4.3) enables adaptive per-slice telemetry with bounded error. However, making this *practical* requires addressing additional challenges: (1) ensuring telemetry headers fit within MTU constraints, (2) maintaining correctness under multipath routing, and (3) recovering from missing telemetry state. *SliceScope* has mechanisms to address these challenges, which, together, make its data plane *deployable in practice*.

1. Bounded telemetry header. A key deployability challenge is ensuring *predictable header size*. Because sources cannot know in advance whether a packet will carry conditional telemetry, operators need a bounded worst-case header size to avoid fragmentation. *SliceScope*'s telemetry header (Figure 4) is designed with this in mind: it has (i) a fixed part that grows with the hop count H, and (ii) conditional fields that grow with the number of reported metrics M but are independent of H. The fixed part consists of a 3 B shim plus 13 bits per hop (10b node ID + 3b flags), rounded to bytes. The conditional part, carried only when $\delta_{s,m} > \Delta_{s,m}$, adds at most 8 B per metric (4 B E_{curr} and, when needed, 4 B V_{aux}). The maximum header size is therefore $3 + \lceil \frac{13H}{8} \rceil + 8M$ bytes.

Crucially, the *heavier conditional fields do not scale with path length, unlike in traditional INT.* Even for three metrics (latency, jitter, loss) and H=8 hops, the total is only $3 + 13 + 24 = 40$ B. For longer paths (H=16) this grows to just 53 B. In practice, operator 5G transport networks are typically engineered with fewer than ~15 hops [18], so provisioning a small constant headroom (e.g., 64 B via TCP MSS/MTU settings) ensures no fragmentation even in worst-case conditions.

2. Multi-path support. Slice traffic often traverses multiple paths, i.e., distinct sequences of switches between the same endpoints. Aggregating telemetry state across paths can obscure performance variations and compromise accuracy. For example, suppose packets p_1 and p_3 of slice s traverse $s_1 \longrightarrow s_2 \longrightarrow s_3$, while p_2 takes $s_1 \longrightarrow s_4 \longrightarrow s_3$. If switch s_3 maintains a single telemetry state for slice s, then p_2 updates E_{rep} based on its higher-latency path. When p_3 arrives via the shorter path, its δ will be computed against p_2's state, potentially suppressing telemetry that should have been reported.

To avoid this, *SliceScope* keys its telemetry state not only by slice ID but also by *path identifier* and output port. The path ID distinguishes between upstream routes that converge at the same switch, while the output port disambiguates diverging downstream paths. To disambiguate paths without incurring large header overheads, *SliceScope* uses a compact 16-bit path identifier, pre-assigned by the control plane based on known slice routes. The path ID is constant-size and avoids carrying an explicit list of switch IDs. In practice, a 16-bit field suffices: with ~300 slices and a few disjoint paths per slice (~1000 paths total), the probability of collision remains below 0.05%.[1]

3. Recovering from state loss. Finally, state in the bucket arrays may be lost due to collisions or selective INT suppressing upstream updates. If a packet from slice k arrives at s_3 but no valid E_{prev} is found, accurate E2E computation would be impossible. *SliceScope* recovers by using the programmable packet replication engine to send a *table miss notification* upstream. Upon receiving such a notification, the upstream switch sets a "table miss" bit for slice k, forcing the next packet to carry full telemetry regardless of its δ relative to Δ_k. If that upstream switch has also lost state, the notification propagates recursively until it reaches the ingress of the slice path, where the state can be reinitialized. This ensures that downstream visibility is restored quickly after any loss. We evaluate the additional overhead of this mechanism in §5.4.

[1]By the birthday bound with $M = 2^{16}$ identifiers and $N \approx 1000$ paths: $P_{\text{collision}} \approx 1 - e^{-N(N-1)/(2M)} \approx 0.0005$.

Summary. *SliceScope* implements the closed-loop framework from §3 using change-triggered INT with per-slice thresholds $\Delta_{s,m}$. We developed mathematical models to estimate monitoring accuracy $E(\Delta_{s,m})$ and overhead $\Gamma(\Delta_{s,m})$ (§4.2), designed a data plane that provides slice-level control and end-to-end semantics (§4.3), and developed mechanisms to enable practical deployment of *SliceScope* (§4.4). We now evaluate whether this design achieves the framework's goals of SLA-aware monitoring with controlled overhead.

5 Evaluation

We demonstrate the feasibility and benefits of slice-aware and SLA-aware closed-loop control by evaluating *SliceScope* as its concrete realization along three main aspects:

- **The benefit of per-slice closed-loop control (§5.2).** Unlike prior work (e.g., [10, 27]) that relies on a single, static threshold, *SliceScope* continuously repositions each slice within the monitoring accuracy–overhead trade-off space. We demonstrate the benefit of this dynamic, SLA-aware adaptation by comparing against two baselines: a *static slice-agnostic* control plane (same theshold for all slices) and a *static slice-aware* control plane (different thresholds for each slice type, fixed over time).
- **Choice of data-plane monitoring primitive (§5.3).** In §4.1, we discuss why we choose change-triggered selective In-band Network Telemetry (INT) over per-hop sketches or probabilistic selective INT. In this section, we further justify this choice by empirically comparing *SliceScope against LightGuardian [33] and PINT [6]* in their ability to provide *accurate end-to-end* tracking of per-slice SLA metrics.
- **Microbenchmarks (§5.4) and testbed evaluation (§5.5).** We evaluate the impact of the number of slices on the control plane's decision time, the effect of epoch length on monitoring performance, and the impact of bucket array size on reporting frequency and other resource overheads. We also demonstrate *SliceScope*'s effectiveness and hardware resource overheads over a testbed with Intel Tofino switches and emulated 5G traffic.

5.1 Experimental Setup, Workloads, and Implementation

While our work applies broadly to any network supporting slicing, we have grounded our motivating example and experiments in 5G networks, where slicing is a core architectural primitive and networks are expected to support hundreds of active slices [4, 5, 8], each with distinct SLAs (§2).

5G-Inspired topology and slices. Our simulation topology is based on Telecom Italia's regional transport network [26]: with 25 Gbps access links, 40 Gbps aggregation links, and a 100 Gbps core. We use 300 network slices of three main types: URLLC, eMBB, and mMTC (Table 4). Network switches do weighted round-robin scheduling with weights decided based on the criticality of the main slice type and 22MB of buffer per-port [32]. See §5.4 for *SliceScope*'s default parameters.

Workloads and SLAs. The impact of monitoring overhead varies with packet size – small packets are more affected by INT headers. Moreover, the "right" point in the per-slice accuracy-overhead trade-off depends on the SLAs of active slices – fewer critical slices means the control plane can use lower Δ values more freely, as less-critical slices require fewer INT reports. As such, we generate three different workloads with different compositions of slice types and packet sizes by changing the fraction of active slice instances that are of each of three main slice types.

Each main slice type has a distinct range of expected performance metrics and packet sizes in the 5G literature [1–3]. For example, URLLC and mMTC slices have much smaller packets than eMBB, and URLLC has stricter latency requirements than eMBB, which in turn is stricter than mMTC (Table 4). Accordingly, the compositions of our three workloads are: (1) *More small (and highly-critical) packets (SP)* with 60% of slices of the URLLC type and 20% for each of eMBB and

Main Slice Type	Range of performance metrics for SLA purposes and expected traffic pattern [1–3]					Composition of Evaluated Workloads (% of slice instances of each main type)		
	E2E Lat. (ms)	Loss (%)	Jitter (ms)	Pkt. Size (B)	BW (Mbps/user)	More small pkts (SP)	Balanced (BAL)	More large pkts (LP)
uRLLC	1-5	<0.001	<1	20-250	1-10	60%	33%	20%
eMBB	10-50	<1	5-30	1K-1.5K	15-50	20%	33%	60%
mMTC	50-100	1-10	50-100	20-125	0.001-0.1	20%	33%	20%
					Total # Slices in Workload	300	300	300

Table 4. Summary of slice types and workloads used in the evaluation. Each slice type reflects a class of applications with similar SLA characteristics; many instances of each slice type are generated, each with its own SLA within the specified range. Bandwidth values refer to typical per-user usage; total slice bandwidth is computed as: (App. BW) × (#users per slice), which is 3–10 (uRLLC), 10–20 (eMBB), and ~10,000 (mMTC). Representative applications include industrial automation (uRLLC), UHD video streaming (eMBB), and sensor telemetry (mMTC). Details on workload generation are discussed in **§5.1**.

mMTC, (2) *Balanced (BAL)* with equal fraction of slices from each main type, and (3) *More large (and less highly-critical) packets (LP)* with 60% eMBB, 20% URLLC, and 20% mMTC. For each workload, we generate traffic by creating the set of active slices, each with randomly assigned SLAs, flow counts, packet sizes, and per-flow rates from expected ranges of these values for the corresponding main slice type. We set the monitoring error tolerance to 5% of each metric's SLA value.

Implementation. We have implemented *SliceScope*'s data-plane pipeline (§4) using P4 [30] ($\approx 2K$ lines of code) in the egress pipeline of an Intel Tofino switch [17]. The telemetry data is carried in an INT shim header and metadata stack between the TCP/UDP headers and the payload. In our Tofino implementation, we added 3 pad bits to the 13-bit per-hop metadata header to meet byte-alignment requirements. We use the Tunnel Endpoint Identifier (TEID) from the GPRS Tunnelling Protocol (GTP) protocol used in 5G as the `slice_id`. In networks using Segment Routing IPv6 (SRv6), `slice_id` can be embedded in the SRv6 header [13]. In other networks, such as MPLS, the MPLS labels, added by the ingress or provider edge router can be used instead. *SliceScope*'s control plane is implemented in Python, using the proprietary Gurobi library [15] to solve the ILP. To estimate the telemetry insertion probabilities $\beta(\Delta_{s,m})$, we fit a Laplace distribution to packet differences $d(t)$ after learning the distribution parameters from the data plane. The control plane runs on a server equipped with an Intel i9 5.7 GHz processor and 32 GB RAM. For large-scale experiments, we developed a discrete-event simulator using SimPy [28] that replicates the behavior of our P4 implementation. We will make our implementation available with the published paper.

5.2 The Benefit of Per-Slice Closed-Loop Control

SliceScope's control plane continuously revisits the per-slice $\Delta_{s,m}$ values based on the set of active slices and network conditions to reposition each slice within the monitoring accuracy–overhead trade-off space. To show the benefit of such per-slice closed-loop control, this section compares it against (1) a *Static slice-agnostic* control plane (same threshold for all slices, similar to prior work [10, 27]), and (2) a *Static slice-aware* control plane (different but fixed thresholds for each main slice type that does not change over time). In our experiments, we set the monitoring error tolerance set to 5% of the metric's SLA value. Our goal is to show which control-plane strategy better navigates the trade-off between tolerance violations (i.e., accuracy) and bandwidth overhead.

Figure 5 depicts the results. The Y axis represents the percentage of violations of the monitoring error tolerance across all packets, and the X axis is the total bandwidth overhead (lower and to the left is better). For each control plane strategy, we plot the *Pareto frontier* representing how it navigates this trade-off space. For *Static slice-agnostic*, each point is a different Δ value, which is fixed for all slices and over time. For *Static slice-aware*, each point represents a combination of

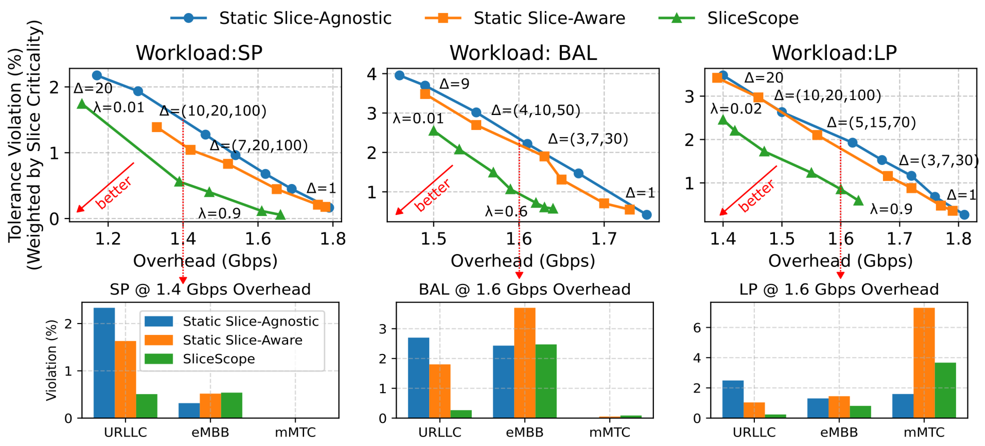

Fig. 5. Pareto frontiers comparing per-packet overhead and tolerance violations across different workload mixes (SP, BAL, and LP). *SliceScope* "moves" monitoring resources from less critical slices to more critical ones as needed. For Static Slice-Agnostic, points represent varying Δ values (1-20), for Static Slice-Aware, different combinations of Δ values for (URLLC, eMBB, mMTC) slices, and for *SliceScope*, different values of tuning parameter λ.

three specific Δ values, one for each of the three slice types. For *SliceScope*, each point represents a different value of the tuning parameter λ in the optimization objective.

The results demonstrate that *SliceScope* achieves the best balance between overhead and tolerance violations across all workloads. While *Static slice-agnostic* with higher Δ values achieves lower overhead, it does so at the cost of increased tolerance violations, which can be particularly problematic for slices with tight SLAs. *Static slice-aware* does better, showing the benefit of customizing Δ even just based on the main slice type, but fails to adjust to changing network conditions and active slices over time. Specifically, for the BAL workload, *SliceScope* with $\lambda = 0.6$ reduces tolerance violations by ~2.3× compared to *Static slice-agnostic* and *Static slice-aware*, while maintaining similar or slightly lower bandwidth overhead.

A closer look per slice type. To better understand where *SliceScope* offers improvements, we break down the tolerance violations by slice type for each workload at a fixed bandwidth overhead (Fig. 5(bottom)). *SliceScope* consistently reduces violations for latency-sensitive URLLC slices compared to both *Static slice-agnostic* and *Static slice-aware* across all workloads. For example, in the SP workload, *SliceScope* reduces the tolerance violations by 3× compared to the best performing static scheme. In the BAL workload, the reduction is up to 4.5×. This can come at the expense of less-critical slices. For example, In the LPS workload, *SliceScope* exhibits higher violation rates for mMTC slices. This behavior reflects its design: rather than uniformly allocating monitoring bandwidth to all traffic, *SliceScope* prioritizes critical slices where SLA violations are more costly, while reducing monitoring fidelity for less sensitive slices. In doing so, it allocates limited monitoring bandwidth resources where they matter the most.

Take-away. The results underscore the importance of optimizing Δ values per slice. *SliceScope* effectively tracks critical slices with high accuracy, while achieving savings in overhead by assigning higher thresholds, i.e., fewer monitoring resources, to non-critical slices or in scenarios when the measured metric exhibits less variation. This adaptive approach enables *SliceScope* to balance the efficiency of selective INT insertion with the need for fine-grained monitoring of specific slices.

5.3 Data Plane Primitives for Slice SLA Monitoring

To validate our choice of threshold-based selective insertion for the data-plane of SLA-aware closed-loop control, we empirically compare *SliceScope* against two prominent alternative data plane primitives: *probabilistic INT* (PINT [6]) and *sketch-based monitoring* (LightGuardian [33]). We evaluate their ability to provide accurate end-to-end tracking of per-slice SLA metrics, focusing on a scenario where an URLLC slice exhibits natural latency variation under our balanced (BAL) workload, as shown by the ground truth line in Figure 6(a). This comparison demonstrates why threshold-based insertion is superior for slice SLA monitoring compared to approaches that sacrifice per-packet end-to-end visibility. Similar trends are observed under the SP and LP workloads.

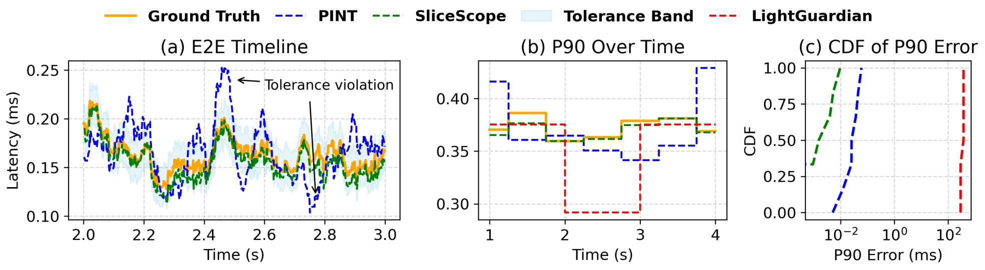

Fig. 6. Comparison of different data-plane monitoring primitives for per-packet end-to-end slice monitoring

We compare the reported end-to-end latency from *SliceScope* and PINT at the per-packet level (Figure 6(a)), and evaluate 90th percentile (P90) tracking accuracy across all three schemes – *SliceScope*, PINT, and LightGuardian – over 500ms intervals (Figure 6(b)) and using the CDF of P90 error in ms (Figure 6(c)). LightGuardian does not provide per-packet tracking and is therefore omitted from the timeline in Figure 6(a). All schemes operate under comparable or higher bandwidth overhead budgets than *SliceScope*.

We observe that *SliceScope* consistently provides accurate tracking of both per-packet and aggregate (P90) latency, closely matching the ground truth. In contrast, PINT and LightGuardian show larger deviations: PINT has spikes where it violates the allowable error tolerances, and LightGuardian exhibits noticeable P90 error in end-to-end latency.

Comparison with PINT [6]. PINT implements selective INT for latency by a distributed sampling process where each packet probabilistically carries latency from one or more hops randomly chosen along its path. This allows lightweight tracking of per-hop latency quantiles under a given per-packet overhead budget. However, unlike *SliceScope*, it does not natively support per-slice end-to-end latency monitoring (§4.1). To enable such comparison, we adapted PINT with a slice-aware reconstruction process at the collector: for each slice, the collector maintains the most recently observed latency for every hop, and reconstructs end-to-end latency by summing these values across the slice's path. We compare the reported end-to-end latency from *SliceScope* and PINT at the per-packet level (Figure 6(a)). PINT only reports per-hop latency quantiles. Due to the lack of per-packet alignment in PINT's reconstruction of end-to-end latency (i.e., it may combine hop measurements from different packets in the same slice), its reported end-to-end latencies deviate significantly from the ground truth, as indicated by the two spikes crossing the allowable monitoring error tolerance.

Comparison with LightGuardian [33]. Sketch-based approaches such as LightGuardian are designed to track per-hop distributions using compact, approximate data structures. LightGuardian uses the SuMax sketch — a variant of the count-min sketch (CMS), combined with binning, to track the distribution of per-hop latencies. It segments the latency range into multiple bins and uses

CMS to count the number of packets in each bin. The sketches are then split into "sketchlets" and embedded into packets, allowing the collector to gradually reconstruct per-hop latency distributions.

To enable a fair comparison in our setting, we simulate LightGuardian with idealized conditions: we assume accurate per-hop sketches are received at the collector every 500ms, allowing the collector to reconstruct per-hop latency distributions for each slice. Next, to compute end-to-end P90 latency for a slice, the collector convolves the per-hop latency distributions along the slice path to obtain the combined distribution, then extracts the 90th percentile. This convolution is necessary because LightGuardian does not track end-to-end metrics. We also increase the number of bins from 4 to 10, compared to LightGuardian's original setup, to improve accuracy.

Despite these favorable assumptions, LightGuardian's reconstructed slice-level end-to-end latency still deviates significantly from the ground truth (Figure 6(b)), due to both the coarseness of its update interval and the inherent inaccuracy of deriving end-to-end metrics from per-hop metrics. In contrast, *SliceScope* leverages the fact that the number of active slices is much smaller than the number of active flows to use a data-plane primitive that stores exact telemetry information (as opposed to approximate statistics) in the data plane and report that when there is a significant enough change, and it can directly provide accurate per-packet end-to-end latency.

Take away. These experiments highlight the limitations of primitives such as per-hop sketches or probabilistic sampling approaches that do not natively support per-packet, end-to-end monitoring for SLA-aware closed-loop control of slice monitoring resources. Whether due to unaligned per-hop sampling or sketch-based summaries, these approaches struggle to provide accurate visibility into slice-level performance metrics. *SliceScope* provides a better alternative in navigating the accuracy-overhead trade-off space in the presence of network slices with well-defined SLAs where real-time and accurate tracking of end-to-end metrics is required.

5.4 Microbenchmarks

Control plane scalability (optimizer and heuristic). We evaluate the runtime of our control plane as the number of slices increases from 50 to 300. Recall from §3.2 that *SliceScope* employs an ILP optimizer for threshold selection, with a greedy heuristic as a bounded-time fallback. Figure 7(a) shows how both strategies scale as the slice count increases. Each control epoch involves two stages: (1) computing expected monitoring overhead and error across candidate knobs $\Delta_{s,m}$ (§4.2), and (2) selecting the final configuration via the optimizer or heuristic (§3.1).

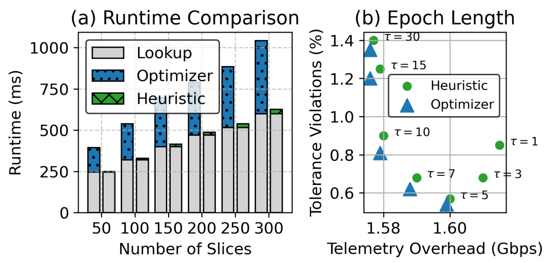

Fig. 7. Control-plane scalability. (a) Runtime scaling with slice count for the ILP optimizer and heuristic fallback. (b) Monitoring quality across epoch lengths τ; the heuristic achieves comparable overhead–violation trade-offs while running ∼12× faster.

To efficiently evaluate overhead and error across many candidate thresholds $\Delta_{s,m}$ (§3.1), *SliceScope* pre-computes and caches them in lookup tables. This avoids repeatedly estimating $\beta(\Delta_{s,m})$ and $E(\Delta_{s,m})$ inside the optimizer: each candidate $\Delta_{s,m}$ would otherwise require a full recomputation of reporting probability and expected error, which the ILP solver (and even our heuristic) would invoke many times per iteration. This lookup phase accounts for a significant portion of the total runtime and grows linearly with slice count, taking ∼250 ms at 100 slices and ∼500 ms at 300 slices on a 16-core machine. Although lookup cost scales linearly, it is inherently parallelizable since lookups for slice–metric pairs are independent and easily distributed across CPU cores.

Once lookup results are available, we select thresholds using either the ILP optimizer with early stopping or our greedy heuristic (§3.2). The ILP solver's timeout is set proportional to the epoch length τ to ensure the control plane adapts to network changes in a bounded time. For $\tau > 5$s, the optimizer consistently converges to high-quality solutions, while shorter epochs ($\tau < 5$s) more frequently trigger fallback to the heuristic.

As shown in Fig. 7(a), the optimizer adds ~600 ms at 300 slices, while the heuristic incurs only ~50 ms of additional latency. *Despite its ~12× lower runtime, the heuristic achieves comparable overhead–violation trade-offs to the ILP* in regimes where both complete ($\tau \geq 5$s) (Fig. 7(b)). This is because enforcing all monitoring-error constraints and minimizing overhead closely approximates the ILP's joint objective when constraints are tight, making the heuristic a practical, SLA-safe fallback under tighter time budgets.

Epoch length. We also evaluate the effect of epoch length τ on bandwidth overhead and SLA tolerance violations (Fig. 7(b)). We observe that $\tau = 5$s achieves the best overall trade-off, with $\tau = 7$s a close second. Shorter intervals (e.g., 1–3s) provide insufficient time to learn stable metric variation patterns, leading to noisier $\Delta_{s,m}$ selection and higher overhead and violation rates. Longer intervals (e.g., 10–15s) reduce overhead by choosing more informed thresholds, but react too slowly to network changes, causing more frequent SLA violations. A 5–7s control interval offers a sweet spot: it is short enough to adapt to network dynamics, yet long enough to support accurate learning.

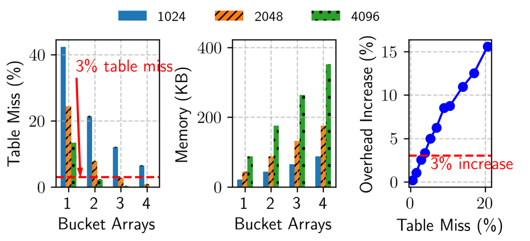

Fig. 8. Bucket array sizing. Table-miss rate, memory usage, and overhead for different (d, w) configurations. A configuration of d=2, w=4096 achieves $<$ 3% miss rate with minimal extra bandwidth overhead.

Bucket array sizing (Fig. 8.) *SliceScope* uses bucket arrays in the data plane to track per-slice metrics. When a lookup fails to find a match, i.e., a table miss, the upstream switch always emits the full telemetry in the next packet. This fallback ensures correctness but can increase overhead (§4.4). We vary the number of arrays (d) from 1 to 4 and number of buckets (w) from 1024 to 2048, and report, in Figure 8, the percentage of table miss (left), memory usage (middle), and associated increase in bandwidth overhead (right) with different configurations of d and w. We observe diminishing returns as d increases: from $d = 1$ to $d = 2$ table misses reduce significantly, but further increases yield only marginal gains while incurring higher memory cost. A configuration of $d = 2$, $w = 4096$ strikes a good balance, achieving <3% table miss rate and only ~3% additional bandwidth overhead. Fewer arrays are preferable, as increasing d consumes more data plane resources (Table 10).

Summary. These microbenchmarks inform our system parameter choices: we set $\tau = 5s$ for optimal overhead-violation trade-off and use the heuristic for scalability, and configure bucket arrays in the data plane with 4096 buckets and 2 arrays to meet our $<$ 3% miss rate target.

5.5 Testbed Evaluation

We evaluate our P4 implementation of *SliceScope* on a 5G hardware testbed comprised of: radio access network based on OpenAirInterface [25] and software-defined radios (SDRs), 5G core based on Open5GS [24] and Intel Tofino switch (UfiSpace S9180-32X) as transport. A Google Pixel 7 Pro smartphone serves as the UE, enabling us to replicate real-world 5G traffic patterns. To further ensure realism, we replay commercial 5G traffic traces [9] that include applications such as cloud gaming and live streaming. User-plane traffic is carried over GTP-U encapsulated flows,

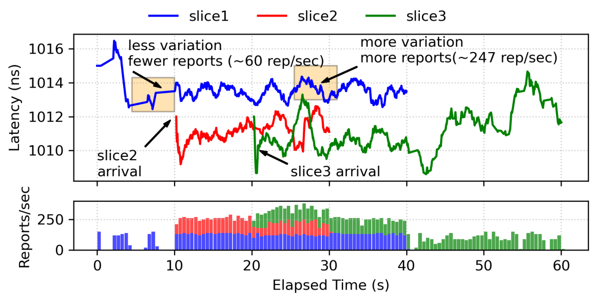

Fig. 9. Tracking of slice metrics on Tofino.

Bucket Array (d, w)	**Hash**	**SALU**	**SRAM**
(1, 2048)	7.2%	27.1%	9.2%
(1, 4096)	8.4%	27.1%	9.2%
(2, 2048)	11.1%	27.1%	9.4%
(2, 4096)	12.5%	27.1%	9.4%

Fig. 10. Tofino resource usage with varying configurations. For the selected setting $(d = 2, w = 4096)$, *SliceScope* consumes 12.5% of hash units, 27.1% of SALUs, and 9.4% of SRAM.

representative of slice traffic. To emulate multi-hop topologies within hardware limits, we apply external loopbacks on the Tofino, following prior work [6].

Real-time tracking. Figure 9 presents real-time monitoring of slice E2E latency (top) and telemetry reporting rate (bottom) across three distinct slices. Traffic from slice 2 and 3 are started at second 10 and 20, respectively. On slice arrival, *SliceScope* immediately starts reporting metrics for the new slice, demonstrating real-time per-slice tracking. The figure also shows that *SliceScope* adapts its reporting frequency in response to variation in the observed metric. While only slice 1 is active and latency is stable, the system reports at a low rate (~60 reports/sec), conserving bandwidth. After slice 2 and 3 join the network, latency variations increase, and *SliceScope* correspondingly raises its telemetry rate to ~247 reports/sec. This behavior exemplifies the core design goal of *SliceScope*: providing real-time, selective telemetry that adjust telemetry bandwidth overhead according to SLA-relevant dynamics.

Switch hardware overhead. Our prototype leverages three key hardware resources on the Tofino switch. *Hash units* compute bucket indices for each row of the hash table, *SRAM* stores the per-bucket telemetry state, and *stateful ALUs (SALUs)* support one read/write per register array, with each row in the bucket array requiring one SALU. *SliceScope* occupies 11 pipeline stages, introducing only ≈ 110 ns of additional processing delay per switch compared to simple forwarding. Resource usage for different bucket array sizes (d, w) is summarized in Table 10.

6 Conclusions and Future Work

Modern networks increasingly rely on network slices with stringent real-time SLA requirements. Continuous monitoring is therefore essential, yet monitoring resources are limited. This work demonstrates that effective SLA-aware slice monitoring requires both adaptive control-plane mechanisms and suitable data-plane primitives. We formalized SLA-aware slice monitoring as a closed-loop control problem and introduced the Telemetry Primitive Contract (TPC) – the minimal set of requirements a data-plane primitive must satisfy to enable such control. We then present *SliceScope*, a practical realization of this framework using change-triggered INT combined with adaptive, slice-aware control, which achieves up to 4× better SLA tracking for critical slices compared to existing approaches.

More broadly, this work suggests viewing telemetry for network slices as a dynamically allocatable resource, managed through coordinated data and control plane mechanisms. This perspective opens new directions for adaptive, SLA-aware slice monitoring in programmable networks.

Future work. SliceScope lays the groundwork for future SLA-driven telemetry systems. We are exploring more advanced control strategies, including learning-based and predictive mechanisms. We also plan to integrate *SliceScope* with real-time SLA enforcement and explore incorporating end-points that could be part of a slice (e.g., VNFs) using SmartNICs and/or eBPF.

References

[1] 3GPP. 2024. *Service requirements for the 5G system.* Technical Specification (TS) 22.261. 3GPP. Version 18.14.0.

[2] 5G Americas. 2017. *5G Services and Use Cases.* White Paper. 5G Americas.

[3] 5G Americas. 2021. *Vehicular Connectivity: CV2X and 5G.* White Paper. 5G Americas.

[4] Alliance for Telecommunications Industry Solutions (ATIS). 2019. IOT Categorization: Exploring the Need for Standardizing Additional Network Slices. https://access.atis.org/higherlogic/ws/public/download/51129/ATIS-I-0000075.pdf.

[5] Arjun Balasingam, Manikanta Kotaru, and Paramvir Bahl. 2024. Application-Level Service Assurance with 5G RAN Slicing. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI 24).* USENIX Association, Santa Clara, CA, 841–857. https://www.usenix.org/conference/nsdi24/presentation/balasingam

[6] Ran Ben Basat, Sivaramakrishnan Ramanathan, Yuliang Li, Gianni Antichi, Minian Yu, and Michael Mitzenmacher. 2020. PINT: Probabilistic In-band Network Telemetry. In *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication* (Virtual Event, USA) *(SIGCOMM '20).* Association for Computing Machinery, New York, NY, USA, 662–680. doi:10.1145/3387514.3405894

[7] Deval Bhamare, Andreas Kassler, Jonathan Vestin, Mohammad Ali Khoshkholghi, Javid Taheri, Toktam Mahmoodi, Peter Öhlén, and Calin Curescu. 2022. IntOpt: In-band Network Telemetry Optimization Framework to Monitor Network Slices Using P4. *Computer Networks* 216 (Oct. 2022), 109214. doi:10.1016/j.comnet.2022.109214

[8] Yongzhou Chen, Ruihao Yao, Haitham Hassanieh, and Radhika Mittal. 2023. Channel-Aware 5G RAN Slicing with Customizable Schedulers. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI 23).* USENIX Association, Boston, MA, 1767–1782. https://www.usenix.org/conference/nsdi23/presentation/chen-yongzhou

[9] Yong-Hoon Choi, Daegyeom Kim, and Myeongjin Ko. 2023. *5G Traffic Datasets.* doi:10.21227/ewhk-n061

[10] Shihabur Rahman Chowdhury, Raouf Boutaba, and Jérôme François. 2021. LINT: Accuracy-adaptive and Lightweight In-band Network Telemetry. In *Proceedings of the IFIP/IEEE International Symposium on Integrated Network Management (IM).* 349–357.

[11] Graham Cormode and S. Muthukrishnan. 2005. An improved data stream summary: the count-min sketch and its applications. *Journal of Algorithms* 55, 1 (2005), 58–75. doi:10.1016/j.jalgor.2003.12.001

[12] Adrian Farrel, John Drake, Reza Rokui, Shunsuke Homma, Kiran Makhijani, Luis M. Contreras, and Jeff Tantsura. 2024. A Framework for Network Slices in Networks Built from IETF Technologies. RFC 9543. doi:10.17487/RFC9543

[13] Xuesong Geng, Luis M. Contreras, Reza Rokui, Jie Dong, and Ivan Bykov. 2024. *IETF Network Slice Application in 3GPP 5G End-to-End Network Slice.* Internet-Draft draft-ietf-teas-5g-network-slice-application-03. Internet Engineering Task Force. https://datatracker.ietf.org/doc/draft-ietf-teas-5g-network-slice-application/03/ Work in Progress.

[14] Google Cloud. 2024. Dedicated Interconnect overview. https://cloud.google.com/network-connectivity/docs/interconnect/concepts/dedicated-overview Accessed: 2025-05-30.

[15] Gurobi Optimizer. [n. d.]. The World's Fastest Solver. https://www.gurobi.com/solutions/gurobi-optimizer/.

[16] Rumenigue Hohemberger, Ariel G. Castro, Francisco G. Vogt, Rodrigo B. Mansilha, Arthur F. Lorenzon, Fabio D. Rossi, and Marcelo C. Luizelli. 2019. Orchestrating In-Band Data Plane Telemetry With Machine Learning. *IEEE Communications Letters* 23, 12 (Dec. 2019), 2247–2251. doi:10.1109/LCOMM.2019.2946562

[17] Intel. [n. d.]. P4-programmable Ethernet switch ASICs. https://www.intel.com/content/www/us/en/products/details/network-io/intelligent-fabric-processors/tofino.html.

[18] Han Li. 2017. 5G Transport Network Requirements, Architecture and Key Technologies. In *ITU-T Workshop on IMT-2020 Network Requirements and Standardization.* Geneva, Switzerland. https://www.itu.int/en/ITU-T/Workshops-and-Seminars/20171016/Documents/2.%20Han%20Li.pdf China Mobile.

[19] Kate Ching-Ju Lin and Wei-Lun Lai. 2022. MC-Sketch: Enabling Heterogeneous Network Monitoring Resolutions with Multi-Class Sketch. In *Proceedings of the IEEE Conference on Computer Communications (INFOCOM).* 220–229. doi:10.1109/INFOCOM48880.2022.9796955

[20] Qiang Liu, Nakjung Choi, and Tao Han. 2022. Atlas: automate online service configuration in network slicing. In *Proceedings of the 18th International Conference on Emerging Networking EXperiments and Technologies (CoNEXT)* (Roma, Italy) *(CoNEXT '22).* Association for Computing Machinery, New York, NY, USA, 140–155. doi:10.1145/3555050.3569115

[21] 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 *Proceedings of the ACM Special Interest Group on Data Communication* (Beijing, China) *(SIGCOMM '19).* Association for Computing Machinery, New York, NY, USA, 334–350. doi:10.1145/3341302.3342076

[22] Jonatas Marques, Kirill Levchenko, and Luciano Gaspary. 2020. IntSight: Diagnosing SLO Violations with in-Band Network Telemetry. In *Proceedings of the 16th ACM International Conference on Emerging Networking EXperiments and Technologies.* 421–434. doi:10.1145/3386367.3431306

[23] Microsoft Azure. 2024. What is Azure ExpressRoute? https://learn.microsoft.com/en-us/azure/expressroute/expressroute-introduction Accessed: 2025-05-30.

[24] Open5GS. 2024. *Open5GS GitHub*. https://github.com/open5gs/open5gs

[25] OpenAirInterface. 2024. *OpenAirInterface 5G RAN Project*. https://openairinterface.org/oai-5g-ran-project/

[26] Emilio Riccardi. 2018. Selection of metro node architectures and optical technology options. METRO-HAUL project, Deliverable D3.1. https://zenodo.org/records/2586698.

[27] Siyuan Sheng, Qun Huang, and Patrick P. C. Lee. 2021. DeltaINT: Toward General In-band Network Telemetry with Extremely Low Bandwidth Overhead. In *Proceedings of the 29th IEEE International Conference on Network Protocols (ICNP)*. 1–11. doi:10.1109/ICNP52444.2021.9651963

[28] SimPy. [n. d.]. Discrete event simulation for Python. https://simpy.readthedocs.io/en/latest/.

[29] Shaofei Tang, Sicheng Zhao, Xiaoqin Pan, and Zuqing Zhu. 2022. How to Use In-Band Network Telemetry Wisely: Network-Wise Orchestration of Sel-INT. *IEEE/ACM Transactions on Networking* (2022), 1–15. doi:10.1109/TNET.2022.3194086

[30] The P4 Language Consortium. [n. d.]. P4 16 Language Specification. https://p4.org/p4-spec/docs/P4-16-v1.0.0-spec.html.

[31] The P4.org Application Working Group. 2024. In-band Network Telemetry (INT) Dataplane Specification. https://p4.org/p4-spec/docs/INT_v2_1.pdf.

[32] Jim Warner. [n. d.]. Switch Buffers. https://people.ucsc.edu/~warner/buffer.html.

[33] Yikai Zhao, Kaicheng Yang, Zirui Liu, Tong Yang, Li Chen, Shiyi Liu, Naiqian Zheng, Ruixin Wang, Hanbo Wu, Yi Wang, and Nicholas Zhang. 2021. LightGuardian: A Full-Visibility, Lightweight, In-band Telemetry System Using Sketchlets. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. USENIX Association, 21.