Observability Is Eating Your Cores: Fine-Grained Analysis of Microservice Metrics with IPU-Hosted Sketches

Alessandro Cornacchia
KAUST

Theophilus A. Benson
Carnegie Mellon University
CMU

Muhammad Bilal
KAUST

Marco Canini
KAUST

Abstract

Observability has become mission-critical for troubleshooting cloud-native technology. However, today's observability fails to meet the demands of cloud-native environments, either resulting in crippling complexity and high costs for collecting and storing huge data volumes, or sacrificing events coverage by sampling at coarse time granularity. We present μView, which stands out from conventional cloud monitors by incorporating a lightweight observability data-plane on Infrastructure Processing Units (IPUs). Our novel architecture leverages the proximity of IPUs to the monitored services to tackle observability bloat. Crucially, μView's data-plane applies streaming data sketching techniques to continuously process and analyze microservice's metrics at fine time resolution, without hurting application performance. We show for several use cases that by anticipating SLO violations μView can help *(i)* narrow the focus on informative observability data, and *(ii)* trigger useful signals about service performance, thus enabling timely proactive actions. Our code and artifacts are available at: https://github.com/sands-lab/uview.

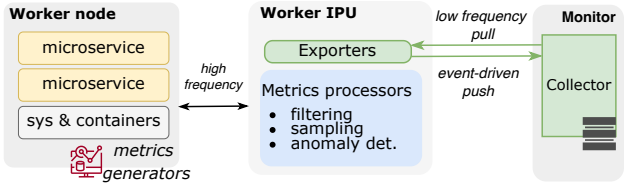


Figure 1: μView enables in-situ analysis of microservice metrics at fine time resolution. It offloads on Infrastructure Processing Units (IPUs) both metrics processing tasks and network-related endpoints for centralized metrics collection.

1 Introduction

Cloud-native applications consist of thousands of single-concern, loosely-coupled microservices running on containerized platforms with many organizations adopting this approach [3, 6, 8, 37, 72, 94]. While the shift from monolithic to distributed design introduces many benefits (e.g., flexibility, simplified maintenance, and efficient resource allocation), it also introduces new system challenges (e.g., resource orchestration and application debugging).

The sheer number of distributed components and the increased pace at which microservices are upgraded/rolled out drastically complicates managing the health of cloud-native applications. First, as the churn of code [22, 43] and the number of distributed components expands [53, 88], the likelihood of failures and performance changes also increases exponentially. Accurate and quick debugging requires collecting significantly more data than a monolithic design [54, 63, 92], which in turns requires non-negligible compute and/or network resources for the monitoring itself [105]. First, this introduces a scalability challenge for monitoring, as the application performance can degrade when sharing resources with monitoring processes. Second, the increase in the type of unique microservices and the number of layers introduced by the container ecosystem (e.g., service mesh [9], sidecars [36]) results in a significant explosion of runtime states and the number of unique types of failures. This naturally increases the range of data and analysis required for troubleshooting, thus introducing an analysis challenge. Overall, managing the health of microservice applications in an automated fashion represents an everyday hurdle for operators.

To address the *scalability challenge*, Operations (Ops) teams employ data sampling [61, 75] which reduces bandwidth and compute overheads while sacrificing data quality and accuracy (i.e., by missing events potentially relevant to later analysis). Moreover, the coarse-grained sampling intervals used by dominant metrics monitors [16, 79] (e.g., in the order of 30 seconds or 1 minute [105]) to collect system (CPU, I/O, memory, power, etc.) and application metrics (e.g., key-value stores cache hits, queue lengths in sidecar proxies, etc.) introduce a large mismatch between the timescale at which metrics are sampled and the one at which application state variations happen (in the order of *ms*). Ultimately, downstream analysis tools are limited to correlating system

events [19, 84, 85], and troubleshooting cascaded [39, 74, 116] or metastable failures [21, 28, 49], all of which are typical in microservice applications.

To address the *analysis challenge*, prior work has proposed supervised techniques [23, 39, 52, 117]. However, these require specialized labels, which are impractical at the speed at which the microservices evolve. On the other hand, recently proposed unsupervised techniques [38, 56, 107] do not address the *scalability challenge*. More generally, these techniques (supervised and unsupervised) fail to dynamically adjust to significant workload changes, which affect a microservice's performance envelope.

To close this gap, this paper proposes μView, a system to enable better observability by preserving monitoring timeliness and accuracy, without sacrificing scalability or increasing overhead. The goal of μView is to provide a lightweight but general mechanism to locally analyze each service's metrics (e.g., application or system) at a fine temporal granularity, and generate useful signals about service performance. Fig. 1 shows a high-level overview of μView. Our architecture enables monitoring of anomalous behaviors locally and deploying data hygiene algorithms before data ingestion and storage. With the local metrics processing service, Ops teams can deploy strategies to filter out irrelevant data and gain online insights about relevant information, without exporting huge data volumes.

In short, μView's design builds on three observations:

Value of local data. First, a broad class of service performance issues can be detected using local metrics at each node. Note that while previous work highlights the need for global knowledge to attribute a problem to a specific service within an application, the detection is often based on locally available metrics. For example, Hindsight [114] showed how local data can assist distributed tracing to capture relevant requests.

Fine-grained analysis. Second, whilst a service's local metrics are already often generated and/or updated at a fine-granularity (e.g., counter values updated at every request), they are often analyzed at a coarse granularity because they are exported to a centralized observability datastore (discussed in § 2). Yet, analysis of this fine-grained data can yield richer (faster, higher accuracy) insights.

IPUs adoption. Third, the recent rise of IPUs[1] [1, 7, 34] in the data center, provides a unique opportunity to process and analyze richer fine-granularity data without imposing CPU overheads, interference, or bloat on the services. By offloading system and application metrics processing to the IPU cores, providers can save profitable server's CPU cycles to execute tenants' workloads.

We found two pivotal challenges towards realizing this vision. (1) Although IPUs have been demonstrated effective to offload several tasks in distributed systems [55, 66, 102], they are typically equipped with a small number of low-performance cores. This limits the complexity of the tasks that can be executed on them. (2) The high code velocity of the microservices ecosystem implies that local analysis techniques must evolve quickly in real-time with minimal manual reconfiguration effort and overheads on the host CPU.

We address both challenges by embracing data sketching as a general but lightweight metrics analysis technique to efficiently perform local analysis on the IPU. Sketch-based analysis operates over multidimensional vectors of metrics in a single pass without requiring local storage of historical data. Additionally, while processing on the IPU eliminates CPU processing overheads, it may introduce communication overheads as metrics must be taken off the host boundaries. If this is not addressed, data transfer can impose significant CPU overheads on its own [105]. To address this, μView introduces an efficient design to transfer data between the services and the IPU, leveraging RDMA over the host PCIe bus.

In summary, we contribute the following:

- we systematically analyze overheads of today's production microservice monitoring systems and motivate why fine-grained metric analysis is a viable choice (§ 2);
- we propose an efficient framework and accompanying design choices for coordinating metrics exchange across various layers of a microservice deployment stack with a IPU for localized processing (§ 3 and § 5);
- we apply sketch-based streaming analysis [65] to the domain of microservice debugging, as a lightweight real-time metrics processing engine. We share our experience and methodological insights (§ 4), and show the sketch ability to determine the critical metrics for each microservice and detect anomalies, while adjusting to workload changes in an online fashion (§ 6.2);
- we present a prototype implementation of μView (§ 5) with several use cases (§ 4.3) and evaluate its performance on diverse benchmark applications, such as DeathStarBench [32] and OnlineBoutique [13], under a heterogeneous set of injected system and application failures (§ 6.2).

Source		Metric	Timescale
App	KV store [86]	db_keys	RPC
	NoSQL DB [76]	mongodb_connections	
Service mesh	Envoy [36]	upstream_rq_active	RPC
Containers	cAdvisor [42]	cpu_usage_seconds	secondly

Table 1: Representative examples of metrics generated at different layers of the microservice's stack. Timescale is the frequency at which a metric is generated and/or updated by the source.

2 Motivation

Our work is motivated by the scalability limitations of traditional monitoring systems [14, 16, 42] for cloud-native applications. We quantify the overheads of observability, breaking

[1] IPUs [1, 7] are a class of programmable NIC devices equipped with onboard processing units that can be programmed to execute offloaded tasks, decoupling datacenter infrastructure from business applications.

down the metrics data *generation* and *ingestion* costs. *We define a metric as a stream of data points that expose application state and resource usage over time, with variations often signaling anomalous behaviors.* We then highlight a set of observability tasks which, despite their simplicity and proven usefulness, are hindered by the data sampling strategies adopted. Based on our cost analysis, we posit the opportunity for in-situ metrics analysis.

Observability at scale has high costs and impacts SLOs. Observability in cloud-native applications is challenged by the thousands of generated metrics; e.g., Uber aggregates 500M metrics/s and stores 20M/s globally [11], which translates to over $21M/month [18] when stored on AWS with 150-day retention costs.

High-volume metric export can also degrade performance, e.g., Alibaba Cloud showed up to 2× tail latency increase due to monitoring cycles [105]. Additionally, since even small improvements in CPU efficiency can save millions of dollars [101], production clouds typically have low CPU headroom to accommodate monitoring collection cycles, as data center operators try to maintain a high water level for user workloads to save costs [20, 27, 115].

Despite this, Ops teams often over-collect metrics under the common-sense assumption that more data increases future utility. This leads to noisy datasets that impede insight [62, 109]. These trends highlight the need for cost-effective observability approaches and to narrow the focus of metrics collection to informative and insightful data.

Existing remedies sacrifice data quality and hinder downstream analysis tools. A microservice's metrics are generated and updated independently by each layer of the microservice's stack: application, service mesh proxies, and system (i.e., OS kernel). Table 1 reports a few representative examples for each category. The metrics at each layer are updated using distinct methods. System-level metrics are updated periodically at a tunable frequency, e.g., for Linux containers, cAdvisor [42] reads from Linux `procfs`. On the contrary, most application-level and service mesh metrics are only updated when RPC requests are processed. For example, Redis [86] updates counters, such as the number of keys it stores, the number of requests for each command (e.g., `HGET`, `HSET`), etc. These differences imply that for many metrics, the update timescale is fine-grained and follows the request arrival rate. Whereas, for other metrics, tuning generation granularity is easy and can be done arbitrarily.

To effectively assist downstream analysis tools [38, 39, 46, 111] in localizing failures and SLO violations, the observability data should ideally enable visibility of system events at the finest granularity across all layers. However, traditional monitoring systems are limited in satisfying this goal. While new metrics monitors like Rezolus [96] pursue high-resolution generation, metrics collection systems such as Prometheus [16] are configured with coarse-grained polling intervals (e.g., no smaller than 30s), which are far from the RPC timescale at which events happen, and not adequate for capturing subsecond-scale variations or spikes. As a result, engineers are often given an insufficient picture of system events to precisely correlate spatially and temporally every-day hurdles such as gray and cascaded failures [50, 74, 85, 116], hotspots propagation [38, 39, 84] and metastable behaviors [21, 28, 49], that are ordinary in microservice applications.

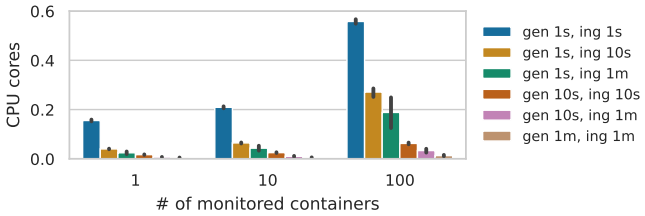


Figure 2: Resource consumption of the cAdvisor exporter for different granularities of metrics generation and ingestion. cAdvisor reads Linux procfs every *gen* seconds to expose the metrics of the monitored containers. The metrics are periodically ingested (*ing*) by a remote Prometheus collector.

System	CPU	Latency	RPS	Analysis
Pull over HTTP [16, 79]	high ×	high ×	high ×	offline
ZERO [105]	zero ✓	net RTT ×	same ✓	offline
μView	offloaded ✓	PCIe ✓	same ✓	online

Table 2: Comparison among existing metrics collectors. CPU: cycles for metrics ingestion on worker node, RPS: impact on requests/s on a cloud application; μView supports timely in-situ analysis of metrics.

Dissecting the observability overheads. Observability compute overheads can come from data generation or ingestion. Anecdotal evidence suggests that regardless of the generation method, ingesting (i.e., collecting or uploading) the metrics into an observability datastore incurs significant performance overheads and costs. Our experiments, in Fig. 2, show that the ingestion interval is a bigger contributor to the CPU overhead than the generation interval. Thus, while it is feasible to generate metrics at a fine granularity, Ops teams use coarse-grained sampling because of ingestion overheads and costs. More details about these experiments are discussed in Appendix B.

Takeaway: an opportunity for *in-situ* monitoring. This implies that observability frameworks can leverage fine-grained metrics, with marginal costs, for generating richer insights if they can avoid the ingestion overheads – for example, by performing processing locally on the node. Compared to state-of-the-art (Table 2), μView's key advantage is that it performs filtering and anomaly detection directly on the IPU, enabling high-resolution in-situ metric analysis, without overwhelming backend systems or network resources. Compared with traditional data stores, e.g., Prometheus, this fundamentally reduces network overheads. Relative to recent advances in OTel [24, 97], μView's use of IPU allows for complex filtering with zero to minimal impact on the microservice. Finally, our

approach eliminates the need for a specialized flow control mechanism, e.g., ZERO [105], which streams all raw metrics over the network at high frequency.

2.1 Use-cases in the observability landscape

We now present use cases enabled by in-situ processing.

UC#1. Accurate spike detection. Engineers spend significant effort to triage and resolve tail latency. However, such a diagnosis usually requires correlating information across different sources, both spatially and temporally [19, 39, 74] – e.g., "*is the CPU hotspot propagating from a sidecar proxy to the service, or from the service to the sidecar proxy?*" Unfortunately, the coarse-grained interval at which system and application metrics are observed makes posterior time-correlation methods inaccurate and error-prone. Coarse-grained intervals lack a sufficient resolution to provide a precise view of metrics evolution, and are unable to distinguish between a smooth increase or a sharp jump, which usually offers a crucial means to pinpoint interesting events in time. A notable example is metastable failures [28, 49], where a sharp and localized variation of system state acts as a trigger [21] for a prolonged anomalous service state.

Observability frameworks require techniques to detect or cross-analyze the data for these kinds of issues.

UC#2. Coordinated cross-container tracing. Even when metrics do indicate an uncommon application state, Ops teams would benefit from coordinating traces [10, 17, 93] sampled across the different microservices processing the request. However, head-based sampling methods (common in production systems [93, 114]) sample user requests uniformly at random, thus relying on luck to capture traces associated with uncommon states – especially if the anomalous conditions last for a short interval. In contrast, biased-sampling of traces using local knowledge of runtime state (i.e., metrics) was shown to help capture informative traces [46, 114].

To realize bias-sampling, observability frameworks need designs that allow them to efficiently observe metrics locally, at a time scale close to the request rate, so that they can anticipate the realization of representative traces and timely mechanisms to trigger their collection.

UC#3. Dynamic metrics sampling. Current monitoring systems use fixed sampling intervals, leading to inefficiencies. Metrics vary in behavior (e.g., CPU usage fluctuates more than CPU limits), suggesting that slowly changing metrics should be sampled on update. Moreover, all metrics benefit from higher sampling rates during anomalies. Cloud provider monitoring tools (e.g., AWS-managed Prometheus) charge tenants for observability data ingestion and storage, creating strong incentives to optimize data collection. Although Prometheus allows per-metric scraping intervals, these settings are static, require manual tuning, and cannot adapt to dynamic workload changes.

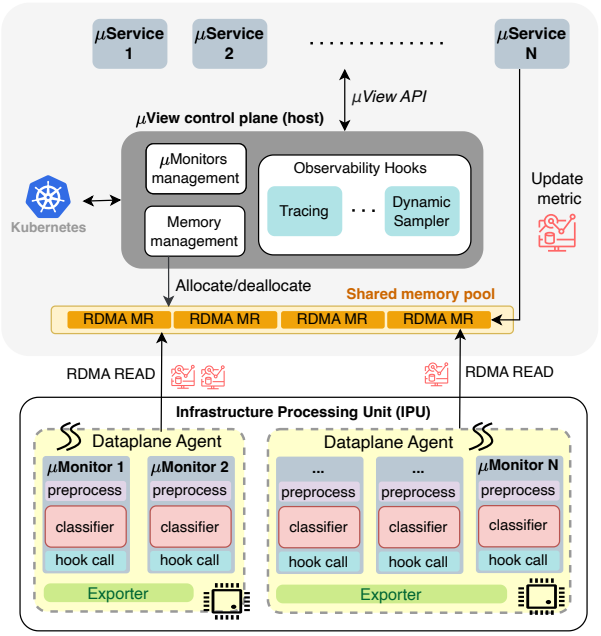


Figure 3: The architecture of μView.

Observability frameworks require quick techniques to accurately determine the appropriate sampling granularity.

3 System design

The architecture of μView (Fig. 3) logically consists of two components, i.e., data plane and control plane, which we discuss below.

Data plane agents. The data plane – offloaded to IPU – runs parallel data plane *agents* which are responsible for reading metrics from the host node and performing real-time analysis of the metrics data. Additionally, data plane agents can be instrumented with *exporters* libraries to interface with and push data to external collectors or observability systems (e.g., Prometheus, OpenTelemetry, etc.).[2]

Each agent contains for each service an array of μ-MONITORs and a control loop for metrics processing. The μ-MONITORs are the core execution units of μView and they process and analyze their microservice's metrics data.[3] An agent's control loop consists of periodically fetching a μ-MONITORs's metrics from host memory at a fine time interval (i.e., *local scraping interval*) and feeding them into the μ-MONITORs – we provide pseudo-code in Fig. 14. The local scraping interval is typically set to be smaller than the remote scraping interval to enable fine-grained analysis (discussed in § 2).

A μ-MONITOR receives as input a vector of Ops-specified metrics for the microservice it manages and produces an

[2] μView supports both event-driven and pull-based observability.
[3] We use the terminology service and microservice interchangeably.

output signal indicating whether the metrics vector contains anomalies. Our design explicitly associates a μ-MONITOR with a single service, regardless of the source of a service's metrics (i.e., apps, service mesh, or system, Table 1), because it allows for seamless co-migration of services and their corresponding μ-MONITORs without creating deployment dependencies among μ-MONITORs. Additionally, this design choice allows for a sketch-based classifier, which we discuss later (§ 4.1).

A μ-MONITOR consists of a set of optional pre-processing transformations and a (streaming) classifier, which performs anomaly detection based on these features. Our design of μView's μ-MONITORs includes a default lightweight sketch-based classifier (§ 4.1) for anomaly detection; however, our design is plug-in-based and enables Ops teams to select alternative classification algorithms. We dive more deeply into our choices for the μ-MONITOR design and specific optimizations for off-path IPUs in § 4.

Control plane. The control plane orchestrates the interaction between the sources that generate metrics and the data plane. At a glance, a control plane component executes on each compute node; this component (1) manages the lifecycle of the μ-MONITORs, (2) handles how the data plane can access memory locations where metrics reside, and (3) based on μ-MONITOR outputs, produces actionable insights for observability.

For generality, μView does not dictate how insights are used directly. Our approach is general in that various observability hooks can be configured and μView merely invokes the hooks when appropriate. This allows integration of μView with existing observability tools, such as distributed tracers, loggers, and metrics collectors. We listed several use case scenarios in § 2.1; § 4.3 illustrates a dynamic sampler hook that pushes metrics to a remote Prometheus collector.

To adopt μView, microservices need to interface with the control plane via an API. We detail the *μView API* in § 5. When the per-service metrics are specified ahead of time (i.e., they are stated in the service initialization configuration), we can generate automatic instrumentation to invoke the API for well-known metrics.[4] In other cases, the microservice application needs to be modified to insert an API invocation for each metric that it intends to export to μView.

4 Metrics streaming analysis: μ-MONITORs

Due to its proximity to the monitored services, μView's data plane supports fine-grained *local* scraping intervals,which allows it to detect short-term variations of monitored metrics without incurring CPU overheads. In brief, a μ-MONITOR must quickly perform anomaly detection in real-time while being sufficiently flexible and efficient to support an arbitrary set of operator-defined metrics streaming at an arbitrarily configured frequency. To flexibly and efficiently support real-time processing, the μ-MONITOR requires a metrics processing pipeline to sanitize the data (§ 4.2) and a general classification technique to detect/extract anomalies/insights from arbitrary metrics (§ 4.1). We describe the challenges underlying the design of a μ-MONITOR and the components to address them.

4.1 Sketch-based Anomaly Detection (AD)

The high volume of continuous metrics collection imposes significant scaling challenges for our design of μ-MONITORs, our metrics classifier module, regarding processing time, memory, and storage availability.

To meet the requirements of lightweight execution, adaptability, and timely detection of the anomalies defined in § 2, we employ a streaming, subspace-based anomaly detection approach based on the Frequent Directions (FD) sketch algorithm [65]. We first motivate subspace-based anomaly detection in relation to microservices metrics, then elaborate on the use of FD-sketch, an instance of dimensionality reduction that operates in a streaming setting.

4.1.1 Metrics AD via dimensionality reduction

In the μ-MONITORs, our goal is to analyze metrics to detect anomalous behavior—anomalies on metrics are frequently used as a sign of problematic application behavior. μView targets multivariate anomalies that break the normal correlations between metrics. It *does not* target anomalies related to seasonality, long-term trends, or recurring patterns. *Subspace analysis (SA)* techniques [47, 48, 58, 71, 91] are a class of unsupervised anomaly detection techniques based on dimensionality reduction (e.g., via PCA/SVD). Recently, they have been proven competitive with state-of-the-art ML-based methods for multivariate AD time-series on a number of benchmarks [89], yet they are simpler to train.

The use of dimensionality reduction is fueled by the observation that microservice metrics often exhibit strong correlations [52], meaning they lie in a lower-dimensional manifold despite being measured in high-dimensional space. At their core, SA algorithms learn a low-dimensional subspace from anomaly-free data, i.e., by computing a low-rank matrix approximation of a training dataset that contains only non-anomalous data points. Anomalies (e.g., memory leaks, CPU spikes) disrupt these normal correlations, causing deviation from the subspace learned during normal operations. SA techniques can effectively detect anomalies by seeing how well a test point can be reconstructed from the learned subspace.

More formally, assume a low-rank matrix U_k can *well represent* any data point in a training dataset M containing only non-anomalous samples. Here k represents the rank of the matrix U. The matrix U_k captures the principal information

[4] The instrumentation would override access for well-known metrics at service initialization time via a LD_PRELOAD mechanism.

of the dataset M in a compact form, and it can be used to detect anomalies in new data points as follows. For any new data point $\boldsymbol{x}' \notin M$, one can project $\boldsymbol{x}'$ onto U_k, obtaining an embedding $U_k^\top \boldsymbol{x}'$ — i.e., a vector in the subspace. Then, one can reconstruct $\boldsymbol{x}'$ starting from its embedding $U_k^\top \boldsymbol{x}'$ by applying the inverse transformation $U_k U_k^\top \boldsymbol{x}'$, and evaluate the reconstruction error $\|\boldsymbol{x}' - U_k U_k^\top \boldsymbol{x}'\|$ (the input data point $\boldsymbol{x}'$ is known). Under the assumption that U_k was learned exclusively from non-anomalous data points, the reconstruction error provides a measure of the deviation of the data point $\boldsymbol{x}'$ from normality. If $\boldsymbol{x}'$ is not anomalous, its reconstruction error is expected to fall within the distribution of errors observed on the training data. Conversely, anomalous points yield higher errors, as they do not align well with the learned subspace. Thus, a point $\boldsymbol{x}'$ is flagged as anomalous when its reconstruction error exceeds a threshold (γ) that can be derived from the training error distribution.

In μView, the matrix M corresponds to a dataset derived from a microservice's metrics, which is collected offline and used to train the initial low-rank matrix U_k (see Appendix C.2 for the detailed workflow).

Remark 1: Scale heterogeneity. Microservice metrics span vastly different scales. Without proper handling, high-magnitude metrics may dominate the reconstruction error, masking anomalies in smaller-scale metrics when computing the L2-norm $\|\mathbf{x} - \mathbf{U}_k \mathbf{U}_k^\top \mathbf{x}\|$. We normalize each metric to zero mean and unit variance across time in the preprocessing module (§ 4.2). This normalization ensures all metrics contribute equally to the reconstruction error, regardless of their absolute scales.

4.1.2 Sketching for streaming operations

After t local scraping intervals, a μ-MONITOR will have observed a sequence of data points $[\boldsymbol{x}_1, \boldsymbol{x}_2, .., \boldsymbol{x}_t] \in \mathbb{R}^{m \times t}$. However, this dataset may have drifted from the training dataset M, i.e., the information contained in U_k is not sufficient to represent $M_t = M \,|\, [\boldsymbol{x}_1, \boldsymbol{x}_2, .., \boldsymbol{x}_t]$.

Thus, a key challenge with designing a sketch-based detector is to adapt to concept drift in an efficient and lightweight manner. We use FD-sketch to address this challenge.

Lightweight adaptation with FD-sketch. A strawman is to recompute U_k from scratch by computing a truncated SVD[5] (Singular Value Decomposition) on each incremental version of the initial training dataset with the new data points appended. This is very demanding in terms of storage and computation, as it would require storing and elaborating a matrix M_t that grows with t. The challenge is how to efficiently update the low-rank matrix in a *streaming* setup, i.e., at discrete time t obtain a matrix U_k using only information from time $t-1$.

[5] refers to the SVD decomposition of a matrix M where only the top-k largest singular values are retained (and associated vectors) — *i.e.,* $\text{SVD}_k(M) = U_k \Sigma_k V_k^T$.

Borrowing from matrix sketching [40, 65], we adopt the Frequent Direction sketch algorithm (FD-sketch) to solve this challenge. The algorithm approximates the *large* matrix $M_t \in \mathbb{R}^{m \times t}$ with a significantly smaller "sketch" matrix $S_t \in \mathbb{R}^{m \times \ell}$, with $\ell \ll t$, such that $S_t^\top \mathbf{x} \approx M_t^\top \mathbf{x}$. This means that to obtain the reconstruction basis $U_k \in \mathbb{R}^{m \times m}$ we can compute the rank-k SVD on S_t — *and not on* the matrix M_t. Moreover, since FD-sketch operates in a streaming setup, it can update the sketch S_t incrementally, using only S_{t-1} and the new sample $\boldsymbol{x}_t$ at any time t, if not regarded as anomalous. This means we do not need to store or elaborate a large matrix M_t. Table 4 summarizes the salient notation used in this section.

Although variants of FD exist–e.g., Randomized FD [41]–they often optimize for settings that do not match observability workloads–e.g., Randomized FD is optimized for sparse matrices; however, cloud-native metrics are typically not sparse. Hence, we focus on the original FD-sketch algorithm [65].

Putting everything together: anomaly score. A μ-MONITOR keeps in memory a low-rank matrix $U_k \in \mathbb{R}^{m \times m}$ and a sketch matrix S_t. For each new input data point $\boldsymbol{x}_t$ at time t, the μ-MONITOR uses U_k to compute a reconstruction error vector $\boldsymbol{\alpha}_t \in \mathbb{R}^m$ as:

$$\boldsymbol{\alpha}_t = \boldsymbol{x}_t - U_k U_k^\top \boldsymbol{x}_t \tag{1}$$

where each component $\boldsymbol{\alpha}_{t,i}$ quantifies how "anomalous" the metric $\boldsymbol{x}_{t,i}$ is, and U_k is the up-to-date low-rank approximation matrix computed using FD-sketch. The vector $\boldsymbol{\alpha}_t$ is used to derive an *anomaly score*, as in [47]. We say that there exists an anomaly at time t if $\|\boldsymbol{\alpha}_t\| \geq \gamma$. Finally, when the input data point is not anomalous, the μ-MONITOR updates the low-rank matrix as:

$$\begin{aligned} U_k &\leftarrow \mathsf{SVD}_k(S_{t-1}) \\ S_t &= \mathsf{FD\text{-}sketch}(S_{t-1}, \boldsymbol{x}_t) \quad \text{if } \boldsymbol{\alpha}_t < \gamma \end{aligned} \tag{2}$$

Remark 2: Workload shift. Legitimate workload variations (e.g., traffic changes, configuration updates) can cause sudden metric shifts that will be misclassified as anomalies. In this case, the sketch S_t should be updated to reflect the new baseline. We introduce a learning rate η for probabilistic sketch updates. When $\|\boldsymbol{\alpha}_t\| > \gamma$ – i.e., point is anomalous – the sketch is updated with probability η. The rationale is that if the anomaly is persistent, it will eventually be incorporated into the sketch. In this way, sustained deviations—after being initially detected and flagged—are treated as part of the updated normal behavior, preventing repeated false alarms for patterns that no longer represent meaningful anomalies.

Remark 3: How do we mitigate metrics dilution. Fundamentally, our sketch aims to minimize the Frobenius norm error and preserve the overall variance of the data, thus large deviations or outliers can heavily influence the learned subspace, potentially leading to false negatives if not handled carefully – a phenomenon often called metric dilution. We mitigate this effect with three design choices. First, we handle scale heterogeneity (Remark 1). Second, we employ one sketch per

microservice – rather than larger sketches. Third, we tune the hyperparameters of the different sketches independently (rank k, threshold γ and learning rate η), rather than adopting a one-size-fits-all static configuration. This is beneficial in practice (§ 6.3).

Takeaways: FD-sketch meets μView requirements. In summary, FD-sketch meets μView's requirements. The approximate sketch matrix S_t is continuously updated with new non-anomalous data points. Therefore, μView remains aligned with recent metrics trends (§ 6.4), which is important in dynamic microservice applications. Second, by maintaining a relatively small matrix S_{t-1}, FD-sketch avoids storage costs and computationally expensive SVD decompositions, remaining lightweight and fast. This makes FD-sketch well-suited for resource-constrained IPU environments, enabling high-frequency analysis for μView, as we quantify in § 6.1. Lastly, FD-sketch does not sacrifice explainability. Because the vector $\boldsymbol{\alpha}_t$ naturally shows how much each metric contributes to the overall anomaly score, Ops teams can identify the most *critical metrics* – for instance, by examining the top-j metrics with the highest reconstruction errors. This can help distinguish different classes of failures, such as issues in the network stack or disk I/O. At the same time, Ops teams do not need to configure per-metric thresholds, since detection relies on a single global threshold γ.

4.2 Metrics pre-processing

As part of the processing pipeline, and before entering the sketch, raw metric values undergo pre-processing transformations. Transformations are simple and account for these aspects: (1) metrics that are cumulative need to be buffered and accumulated at each sample (e.g., resource usage metrics like container_cpu_usage or container_network_transmit_bytes, which report ever-increasing counters), (2) metrics may need to be aggregated across replicas of the same service,[6] and (3) metrics need to be mean-centered and rescaled to unit standard deviation (cf. Remark 1). For rescaling, the mean and standard deviation are initialized from the training dataset M and periodically recomputed at runtime by maintaining running aggregates $\sum_i x_i$ and $\sum_i x_i^2$.

4.3 Observability hooks

Distributed tracing hook. Next, we describe how μView uses continuous local metrics analysis to anticipate problematic executions (as motivated in § 2) and enable distributed tracing to sample informative requests.

At every *local* scraping interval, μView outputs a binary decision on whether the tracer should sample user requests. Later, it informs the external distributed tracing library that was previously registered with the distributed tracing hook

[6] μView supports standard aggregation functions including max, min, sum and mean; it can be extended with custom aggregators.

Control APIs	Declaration
Initialize μ-MONITOR	void **configMicroMonitor**(ServiceID, pretrainedClassifier, scrapingInterval) void **registerMetrics**(List<MetricID>)
Configure Metrics	MetricID **createMetric**(ServiceID, Metric, Type, AggType) MetricObj **getMetric**(ServiceID, MetricID)
Add Hooks	HookID **registerHook**(List<ServiceID>, HookFn)
Runtime API	**Declaration**
Increment (counters)	void inc(MetricObj, value)
Decrement (gauges)	void dec(MetricObj, value)
Hook Interface	**Declaration**
HookFn	void _(MetricID, MetricValue, AnomalyScore)

(a) μView APIs and hook interface.

```
from microviewapi import MicroViewClient
from microviewapi import COUNTER
# 1. Connect with Microview
client = MicroViewClient(name="python-service")
# 2. Register metrics with control plane
metric = client.create_metric(name, COUNTER)
# 3. Update: increment counter
metric.inc()
```

(b) Instrumenting a Python service with the μView client library.

Figure 4: μView API and instrumentation examples.

(§ 5) about its decision. Logically, the μView sampling policy is based on (1) aggregating the output of all μ-MONITORs of services involved in a user request, and (2) deciding to sample the request if at least one of them has classified its metrics as anomalous. At this point, this sampling decision is held until the subsequent classification cycle, when a new decision will be taken based on a new metrics reading.

Ultimately, μView serves as the trigger; we envision that different realizations for exporting traces are possible (e.g., retroactive sampling [114]).

Metrics sampler hook. The dynamic metric sampling mechanism allows μView to report informative metrics variation events promptly, and save the data volume generated for their collection. Our adaptive sampling mechanism builds on the core idea of using the anomaly score value as a measure of the informativeness of the collected metrics. For every local scraping interval, the sampler hook receives the per-service anomaly score $||\boldsymbol{\alpha}||^2$ from the μ-MONITORs. Then the dynamic sampler hook takes a per-service decision about whether the current metrics should be sent to a remote monitor system. The dynamic sampler hook decides to send metrics when the anomaly score is above a threshold. In this case, it notifies external libraries via the registered callback.

5 System implementation

μView API. To facilitate μ-MONITOR metrics collection and management, μView exposes a management API (Fig. 4a). This interface allows μView users to register metrics metadata and specify initial configurations – *e.g.*, μ-MONITORs's local scraping interval and pre-trained classifier. This interface can also be used to register observability hooks. To enable a general set of hooks that can take arbitrary actions based

on the insights generated by a μ-MONITOR, μView presents a simple interface (Fig. 4a) that all hooks must implement. In brief, all hooks are event-driven and must implement an interface that implicitly registers them as callbacks when insights are generated by a μ-MONITOR. Finally, the API provides a Prometheus-like client library to instrument microservices code. The set of per-service metrics is configured by Ops teams; we consider this as an initialization parameter for μView. The microservices must use this library to create metrics during their initialization (Control API), and manipulate metrics at runtime (Runtime API). μView supports standard Prometheus metric types [16] (counters, gauges and histograms). Fig. 4b shows an example of how to instrument a Python service with the μView client library.

μView data plane. We choose *off-path* IPUs [106] with System-on-Chip (SoC) cores separate from NIC cores. We select this type of IPUs for two main reasons. First, they typically feature more than a dozen SoC cores, aligning well with the task of metrics analysis, which naturally lends itself to embarrassingly parallel computation. For example, NVIDIA BlueField-3 [12] and Intel IPU E2000 [95] both boast 16 ARM cores. Second, unlike on-path solutions, they host a full-fledged OS and can run high-level code runtimes and libraries [73], simplifying development and increasing flexibility to process metrics samples.

The μView control plane spawns one agent on each of the IPU's available cores, binds the agent to a core and maintains a control channel (TCP connection) with it. At runtime, the control plane instantiates a new μ-MONITOR for each service and assigns the μ-MONITOR to one of the agents via the corresponding control channel. Thus, each agent runs the μ-MONITOR assigned to that core.

Host-to-NIC metrics data movement. μView leverages RDMA for host-IPU communication over the PCI bus, achieving host CPU bypass. The agents issue RDMA READs to fetch metrics from host memory. In this way, μView supports high-frequency metric reading without incurring the overhead of system calls and memory copies. We choose RDMA over proprietary DMA technologies (e.g., NVIDIA DOCA SDK [81]), because *(i)* it has been recently demonstrated to provide higher throughput than DMA for host-to-IPU [106] (SoC) data transfers [106] and *(ii)* of its generality and ease of portability across IPUs of different vendors. The agents receive the configuration of metrics to be fetched for each μ-MONITOR from the control plane, which includes the RDMA rkey and the RDMA remote address for each monitored metric. Each agent manages its own set of Queue Pairs (QPs) – we do not share QPs with other agents to avoid synchronization overheads.

Host memory management. The μView's memory management module manages a contiguous pool of shared memory, where microservice metrics are stored. This design introduces the challenge of enforcing access control boundaries across potentially untrusted tenants. Similar to SPRIGHT [83], the trust model in μView assumes that the microservices within the same application trust each other, but the microservices in different applications may not. Therefore, to limit unauthorized memory accesses, μView assigns a private shared memory segment to each microservice application. By allocating distinct shared memory segments, μView provides isolated security domains to different applications.[7]

To optimize memory access, the control plane groups metrics by service and allocates them contiguously in memory. Further, it groups microservices assigned to the same dataplane agent into the same RDMA MR to reduce the number of RDMA READs. For efficiency, metrics slots are implemented as a structured byte-aligned format, and the RDMA MRs are aligned with the 4KB page size in use.

6 Evaluation

System setup. Our testbed comprises 4 nodes, each equipped with 8 Intel Xeon E3-1230v6 CPUs at 3.50 GHz, 32 GB of RAM, and 100 Gbps network interfaces. We implemented the μView control plane and data plane components, and deployed the system on a NVIDIA BlueField-2 IPU with 40 Gbps network link capacity.

6.1 Performance analysis microbenchmarks

Setup. We measure the throughput of the μ-MONITOR analysis pipelines under a variety of microbenchmarks while varying the number of monitored microservices and metrics. We compare overheads of FD-sketch [65] and compare with two baselines: (1) *Variational Autoencoder* (VAE) [57]: representative of a neural network-based dimensionality reduction method. We adopt a 2-layer architecture for both the encoder and the decoder, with latent dimension of 10 and ReLU activation. (2) *Threshold*: each metric is compared against an operator-defined threshold and an alert is raised if the threshold is exceeded. Our experiments last 10 minutes; we sample throughput every 10 seconds and average the results over the entire duration. We discard the first minute to avoid sampling throughput during transitory, e.g., unfair share of RDMA bandwidth across the dataplane agents and cache warm-up effects. To observe the impact of pod changes, we fix the number of metrics to 64 while varying the number of pods from 8 to 256. To observe the impact of the number of metrics, we vary the metrics from 16 to 256 while fixing the number of pods to 8 (matching IPU cores).

Results. First, we measure the μ-MONITORs' processing throughput on the IPU for different anomaly detectors. Our results in Fig. 5 shows that FD-sketch enables higher throughput compared to the threshold-based classifier while being far more dynamic and adaptable. On the other hand, VAE's throughput is less than half of the throughput of FD-sketch due to neural network inference overhead.

[7] A complete security analysis of μView is beyond the scope of this paper.

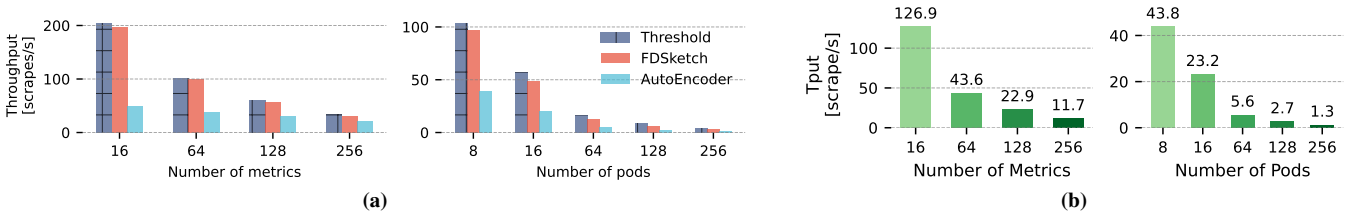


Figure 5: Performance of μ-MONITORs on Bluefield-2 [12] IPU (a) and μ-MONITORs + Prometheus exporters (b).

Second, we measure throughput when we also enable the Prometheus exporters in the μ-MONITORs. The exporter works as a proxy for RDMA-fetched metrics, exposing them to HTTP endpoint on the IPU for Prometheus to scrape. We use `wrk` tool to scrape the `/metrics` endpoint from a different node. We tested the worst-case scenario without parallelism, i.e., a single μView data plane agent. [8] Fig. 5b indicates that with a single agent, the IPU can support up to 256 pods with 64 metrics each at a throughput of 1.3 scrapes/s. This means we can decrease the scraping interval to 1s and still collect metrics from up to 256 pods, while not consuming CPU cycles on the worker node despite the high collection frequency.

6.2 μ-MONITORs local analysis

Benchmark applications. We used the following microservice benchmark suites: DeathStarBench (DSB) Hotel Reservation [32], an open-source popular benchmark application (15 services); we deployed a Docker-based version using the Blueprint compiler [21]. OnlineBoutique [13], a real-world application developed by Google, with 11 services that realize an online shop. We deployed OnlineBoutique on a 3-node Kubernetes cluster.

Fault injection. Our fault injector comprises the following performance and application-level anomalies from prior works [38, 84]: (1) *Memory capacity.* One or more threads continuously allocate/deallocate memory for a configurable duration and intensity. This kind of performance anomaly emulates memory leaks due to code bugs or inefficiencies, such as excessive heap memory usage or JVM garbage collector overhead, which are common scenarios in large applications. (2) *LLC pressure.* The injector introduces pressure on LLC bandwidth and/or capacity, by frequently accessing a small portion of the LLC or by performing random accesses of configurable intensity to cover the size of the LLC. (3) *I/O pressure.* We simulate I/O pressure by introducing random read/write operations on the disk, which can be caused by a high request rate to the database or by a high number of file operations. (4) *CPU usage.* We inject a CPU-intensive workload of configurable intensity (i.e., number of cores), which can be caused by a high request rate to the service, code bugs, or a high number of computations to serve a particular request. (5) *L7 failure.* Inspired by real-world failure stories [5], we trigger cache evictions in a Redis key-value store. Dynamic variations in Redis cache status can result in different downstream request execution paths.

[8] The agent receives scrape requests, issues RDMA reads on its QP, and feeds the responses to the μ-MONITORs analysis pipelines. Thus, at each Prometheus scrape, all metrics are collected and processed from all the pods on the node using a single thread.

Anomaly type	Injection tools
Memory pressure	ChaosMesh [2], `stress-ng` [31], `pmbw` [99]
LLC pressure	FIRM's `llc.c` [84]
I/O pressure	ChaosMesh [2], `stress-ng` [31]
CPU usage	ChaosMesh [2], `stress-ng` [31]
L7 failure	`redis-cli` [86], ChaosMesh [2]

Table 3: Anomaly injection setup.

Our fault injector combines industry-standard ChaosMesh [2] with common performance stressing tools (cf. Table 3), and uses a modified version of FIRM to schedule faults on a service. We develop a custom Blueprint plugin to support this tooling. To evaluate the detection of short-term failures in complex scenarios [45], we inject anomalies lasting no more than 30 seconds. The injector triggers anomalies uniformly at random into services, with adjustable intervals, duration, and intensity, and we allow simultaneous injection in multiple services.

6.3 Distributed tracing experiments

We evaluate the performance of the μView distributed tracing sampling policy (§ 4.3).

Observability data. We create training and test datasets by collecting metrics and traces. For each microservice, we collect 35 system-level metrics from the OS, using cAdvisor [42] (c.f. § 2). We also collect application metrics where applicable (e.g., 38 metrics for Redis key-value store). We generate the training datasets in a controlled environment (Appendix C.2), where we allocate conservative vCPUs and memory limits of each microservice, i.e., such that tail resource utilization is no larger than 10%. We use Prometheus [16] to collect and store metrics, *with a local scraping interval of 1 second*, unless otherwise specified.[9] For traces, we use Jaeger [10]. We ran our experiments for 1 hour, using the first 25 minutes worth of data for the training and validation datasets (corresponding

[9] We also set the `housekeeping_interval` in cAdvisor consistently.

to 1500 sampled vectors) and the remaining 35 minutes for the test dataset.

We annotate traces in our test datasets that violate SLOs. We group traces by API and use the 3σ-rule to establish latency SLO violations for each API, as different API calls may correspond to different SLOs. We also annotate traces that present errors (e.g., HTTP 5xx status codes). We refer to traces violating SLOs or containing errors as *symptomatic traces*.

Comparisons. We measure *coverage*—the percentage of collected symptomatic traces–and *overhead*—measured as a percentage of false positives, i.e., the percentage of *non-symptomatic traces* unnecessarily collected via the tracing hook policy. We compare FD-sketch against the following anomaly detection methods:

- *PCA*: a classical dimensionality-reduction approach with the same memory footprint as FD-sketch $O(km)$; however, it requires full data access and offline retraining, with no support for streaming updates;
- *k-Nearest Neighbor (k-NN)*: a distance-based method that marks a data point as anomalous if it is far from its nearest neighbor in the training dataset (which contains only normal samples).
- *VAE (Variational Autoencoder)* [57]: a deep learning encoder-decoder structure that reconstructs input vectors from a learned latent representation;

The methods include both classical and deep learning approaches for dimensionality reduction. Our selection of PCA and 1-NN is motivated by a recent study [89] demonstrating their performance are competitive with more complex deep learning methods on several multivariate time-series AD benchmarks. All methods are trained on the same dataset of microservice metrics.

To better contextualize μView's performance, we also compare with an "oracle" – a detector having exact knowledge about the anomaly injection pattern (i.e., ground-truth) – and with head-based sampling [15, 25], introduced in § 2.1.

μView anticipates symptomatic traces. Fig. 6 shows the coverage-overhead trade-off of μView's distributed tracing hook, for our two workloads. With around 20% sampled traces, FD-sketch captures ≈80% of anomalous traces, achieving a favorable trade-off between coverage and overhead. Random head-based sampling would require collecting nearly 80% of all traces to achieve comparable coverage. PCA offers the best trade-off between coverage and false positives, but requires offline retraining and full data access. Compared to FD-sketch, PCA can be regarded as an *offline* counterpart, offering a competitive reference on the performance achievable by FD-sketch without streaming constraints. In contrast, for the same training data volume, both VAE and 1-NN miss many anomalies and trigger trace sampling less frequently than the oracle.

Impact of sketch optimizations (§ 4.1). In Fig. 7, we run a sensitivity analysis and show performance with per-μ-MONITORs hyperparameter optimization (red dots) and without (other points). For the points without per-μ-MONITORs tuning, the sketch configuration is static and equal for all services. Namely, we start by setting $k = m/5$ as per sketch guidelines [47]. Then, we perturb this value. Similarly, for l (Table 4). Finally, for each configuration obtained, we introduce our learning rates η and test values ranging over a log-space.

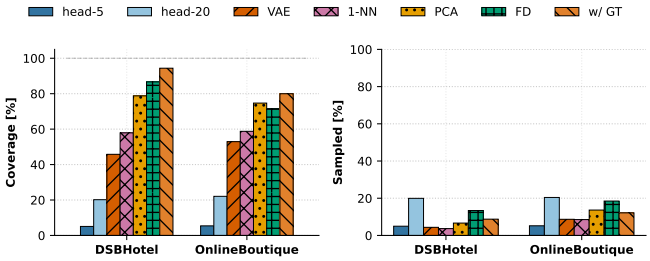
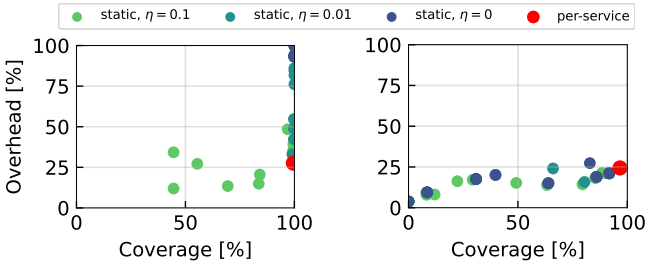


Figure 6: μView's performance for the distributed tracing use-case. Coverage is the percentage of symptomatic traces collected, and sampled denotes the overall percentage of collected traces.



(a) OnlineBoutique [13] **(b) DSB HotelReservation [32]**

Figure 7: Classification performance for the distributed tracing UC, with our per-microservice sketch optimization (red dots) and without (other dots).

First, we observe that μView finds a sweet spot (red dots) between coverage and overheads, thanks to independent hyperparameter tuning on each μ-MONITORs. It lies on the empirical Pareto front of the performance profile, dominating other static configurations (other dots). Second, we can observe the impact of the learning rate η for the OnlineBoutique workload. Without learning rate ($\eta = 0$) there is a tendency to raise many false alarms (dark blue points), resulting in high overhead. This is because if one of the metrics deviates abruptly and μView keeps raising anomalies, the sketch is not updated ($\eta = 0$). On the other hand, for $\eta = 0.1$, the sketch is updated too frequently, with chances of incorporating faulty samples during the update step and thus contaminating the sketch low-dimensional matrix representation (which should represent non-faulty states only). This translates in low coverage. The behavior depends on how different metrics and services react to the anomalies that are injected, and there's no one-size-fits-all choice for η. This is evident by the different performance profiles across workloads. For OnlineBoutique, the performance profile has the largest variance along the

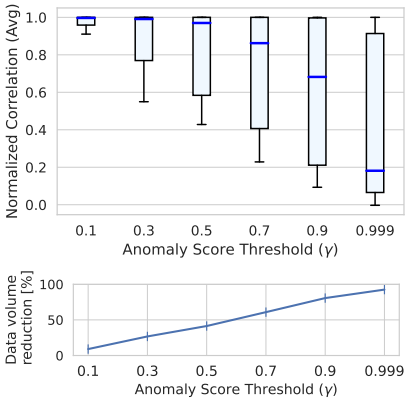
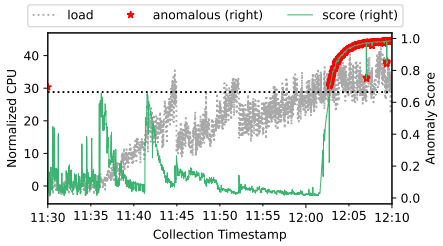
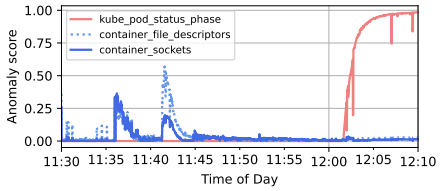


Figure 8: Performance of the adaptive metrics sampler hook for different anomaly score thresholds.



(a) Anomaly score of a `frontend` service. The horizontal line represents the anomaly score threshold.



(b) Identification of the top-3 relevant metrics.

Figure 9: Evaluation under non-stationary load. The sketch can discriminate between short-term overload, i.e., before the Kubernetes HPA rescales the workload and critical overload conditions.

overhead dimension (Fig. 7a), for DSB hotel, the performance profile has larger variance along the coverage axis (Fig. 7b). Therefore, μView treats η as a tunable hyperparameter and we found that per-μ-MONITORS tuning of η achieves good performance in practice.

6.4 Adaptive sampling experiments

Lastly, we demonstrate how μView can reduce the data volume generated by metric collection while preserving accuracy and thus reducing ingestion and storage costs. For this experiment, we consider a single instance of a Redis microservice, to include both systems and application metrics. We train the sketch for 30 minutes, and run the dynamic sampler on a test dataset M collected over 10 minutes. Borrowing notation from § 4.1, let's denote as M_i the time-series of metric $i = 1,..,m$. Then, we apply the sketch dynamic sampler, as described in § 4.3, and obtain a filtered time-series $\tilde{M}_i$ with reduced data volume, for every metric time-series M_i. We evaluate the trade-off between accuracy and data volume reduction. As a measure of accuracy, we use the Normalized Cross-Correlation (NCC), defined as:

$$NCC_i(\tau = 0) = \frac{corr(M_i, \tilde{M}_i)}{corr(M_i, M_i)}$$

The cross-correlation at a time lag $\tau = 0$ measures the similarity between the two time series. We normalize $NCC_i(0)$ by the autocorrelation at lag 0 of the original time-series. Since the dynamic sampler hook filters out some samples, M_i and $\tilde{M}_i$ will have different lengths. To compute $NCC_i(0)$, we fill missing samples of $\tilde{M}_i$ by propagating the value of the last collected sample until the next useful sample. This emulates the behavior of a centralized collector that, at any time before the next available collection, would only know the last sample.

Figure 8 reports the results for different anomaly score thresholds. For adaptive sampling, the anomaly score threshold controls the aggressiveness of the sampler. When the threshold is large, metrics are sampled only in the presence of large fluctuations. This translates into low similarity with the original series. For example, moving from $\gamma = 0.9$ to $\gamma = 0.99$, NCC drops more than half (Fig. 8 top). Correspondingly, larger thresholds imply larger data savings. We compute the amount of data volume reduction as the ratio between the number of samples in the filtered metric $\tilde{M}_i$ and the number of samples in the unfiltered metric M_i (Fig. 8 bottom). We observe that while the data volume reduces linearly, the accuracy plateaus between $\gamma = 0.3$ and $\gamma = 0.5$, then drops following a logarithmic trend. This suggests that there exists an optimal trade-off between accuracy and storage overhead. For example, at threshold 0.5, the adaptive sampler can reduce the ingested data volume by 50% with a less than 5% reduction in the accuracy.

6.5 Performance under autoscaling conditions

We study the performance of the sketch detector under dynamic workloads and autoscaling.We answer: *is the sketch a good fit for the dynamic nature of a microservice application?*

We focus on autoscaling as a representative scenario of concept drift. We consider the task of discriminating a *transient* overload from *persistent* overload. With transient overload, we refer to a temporary increase in resource demand due to an increase in incoming request load (i.e., RPS). This situation is solved once the Horizontal Pod Autoscaler (HPA) deploys additional service replicas to match the ingress workload. Consequently, we do not expect anomalies to be raised if the HPA can respond to the increased demand. However, because the system runtime state deviates from the fault-free

state, this scenario challenges the adaptability of FD-sketch under realistic, non-stationary, conditions.

We run this experiment using a *frontend* service. We deploy the HPA for this service and start the deployment with a single replica. We configure the node capacity by setting the Kubernetes control plane's `maxPods` parameter, which limits the maximum number of microservices allowed to be scheduled on a single node. While collecting the test dataset, we increase the load linearly by adding a new user every 25 seconds on average, e.g., starting from 11:35am as shown in Fig. 9a. Each user generates a request every 3 seconds. We plot this load in the gray curve in Fig. 9a, representing the maximum CPU consumption across all service replicas.

A strawman is to train the sketch under stationary load conditions, i.e., with a constant request arrival rate. We found this strawman does not work well, as the sketch outputs numerous false positives around the transient workload variations (plot omitted). Then, we trained the sketch under variable but controlled load conditions—i.e., without triggering persistent overload. The results are highlighted in Figs. 9a and 9b. We derive the following takeaways.

Takeaway-1: adaptation at runtime. During the first transient overload, i.e., before the first time the HPA rescales the workload, around 11:45am, the anomaly score (green curve) rises with two spikes slightly below the threshold. After the HPA rescales the workload for the first time, the load is uniformly split across the available replicas. Subsequently, the anomaly score doesn't drop to zero but remains significantly lower than before the scaling, indicating that μView has adjusted its classification to the new data distribution.

Takeaway-2: streamlined operational effort. Around 12:03pm, when the cluster enters a persistent overload state, the red curve in Fig. 9a shows μView starting to report anomalies. μView relieves platform engineers from the burden of handcrafting per-metric thresholds. Thanks to the sketch mechanism, with μView we only tune a single anomaly score threshold and not a per-metric configuration.

Despite using a single threshold, Ops teams can still trace back to the culprit metrics. In Fig. 9b we plot the top-3 metrics in $\boldsymbol{\alpha}_t$ by decreasing component magnitudes (§ 4.1). We see that the initial spikes before the first workload rescaling were due to metrics that directly relate to the number of open connections, such as `container_sockets`. Instead, the detection of persistent overload can be explained with the metric `kube_pod_status_phase`, since several pods failed to start due to the node having saturated its scheduling capacity.

7 Discussion

7.1 Limitations

Scope of FD-sketch. While μView demonstrates effectiveness in practice for the use cases in § 4.3, the approach inherits the limitations of dimensionality reduction methods. FD-sketch primarily identifies anomalies that cause a structural deviation from the learned normal patterns. Its sensitivity might be limited in the presence of: *(i)* subtle point anomalies such as small-magnitude deviations in individual metrics that do not perturb the overall correlation structure and *(ii)* training dataset contaminated by anomalous data points with high variance from the normal behavior. μView primarily targets anomalies that cause pronounced subspace deviations, such as performance-related anomalies that often manifest as significant coordinated deviations from established operational patterns across multiple microservice metrics (e.g., system, application, proxy) [98].

Handling abrupt workload shifts A practical challenge for μView is distinguishing between true anomalies and legitimate workload shifts caused by traffic changes, configuration updates, or operational interventions. In the case of a legitimate workload shift, our approach–like other anomaly detection algorithms–will initially output high anomaly scores before the system adapts to the new baseline, thanks to metric normalization and our probabilistic sketch update mechanism, η. Meanwhile, the sketch may still generate numerous alerts. To control the exported observability data volume during such transitions, we recommend integrating μView with existing budget-based sampling policies. For example, Sifter [59] and Sieve [52], propose methods to map anomaly scores to sampling probabilities. Similar mechanisms could be implemented by the observability hooks. The key takeaway is that the μView's adaptation mechanisms for FD-sketch address baseline shifts on the longer-term, while short-term transitions are best handled by specific sampling policies of the observability hooks. The particular sampling policy for individual use-cases is orthogonal to sketch-based anomaly detection.

Training effort. μView, like other unsupervised learning methods, requires a training phase in a controlled environment on fault-free data (Appendix C.2), and performs better when tuning FD-sketch hyperparameters for each microservice. This may introduce operational overhead. However, we argue that this overhead is limited in practice. First, microservice applications often have a staging environment that mirrors the production setup, which can be used for training. Second, the hyperparameter tuning process is automated in μView (Appendix C.2), minimizing manual intervention.

7.2 Is IPU the only choice?

Our choice to run μView on IPUs is not fundamental and other realizations are possible. However, there is an economic argument in favor of our design given current trends in cloud data centers. Virtualization is arguably the most significant factor driving the business model of cloud providers. Profitable CPU cycles are those that execute tenant workloads, and cloud providers aim to maximize the use of these cycles to retain a competitive edge in the market. Instead, μView occupies cycles for high-resolution metrics preprocessing (§ 4.2) and

streaming analysis (§ 4.1). By offloading these stages to the IPU, cloud providers can significantly increase the value they offer. Moving these tasks away from the host CPUs ensures that the latter are freed up for more tenant workloads, which directly translates into more rentable and revenue-generating CPU cycles. Moreover, the isolation of these monitoring tasks within the IPUs enhances operational efficiency. Since system metrics are handled independently of the tenants' workloads, there's less risk of interference or resource contention. This not only improves performance but also boosts reliability, as workloads are shielded from potential bottlenecks caused by internal system monitoring.

In the competitive cloud landscape, where every additional rentable cycle can make a difference, this model adds clear value. The IPU-based approach ensures that critical observability operations are efficiently managed without sacrificing the primary goal of running tenant workloads. Consequently, cloud providers can increase profitability while offering an optimized service to their customers.

8 Related work

Troubleshooting microservice applications. There is significant work on microservices observability data: trace [35, 52, 87, 93, 100, 114], logs [80, 82, 112], metrics [98, 107], but few have considered non-centralized processing of metrics. Fay [33] and recently Hindsight [114] proposed retroactive sampling but did not focus on the underlying metrics analysis mechanisms;

Metrics anomaly detection. While root cause analysis methods need a holistic view across multiple data sources, anomaly detection (AD) is typically based on time-series metrics and runs online. Statistical rule-based methods [30, 70, 110] and deep learning (DL) based methods [47, 51, 103, 113] are used to detect anomalies in large-scale systems. Unsupervised DL-based approaches based on encoder-decoder architectures [103, 104], similarly to μView, perform AD via dimensionality reduction, though requiring more training data. This aligns with the common deployment model of anomaly detection, which is often performed locally on individual metric streams, making it amenable to localized, in-situ processing.

Offloading to IPU. Recent work has demonstrated the benefits of offloading to IPUs for network packet processing [29, 60, 108], accelerating key-value stores [66], distributed file systems and transactions [55, 64, 90], GPU-centric applications [102], and microservices [67]. μView differs in its application of offloading for observability.

Sketches for telemetry. Sketches [44, 68, 69, 77, 78] have long been explored as a fundamental primitive for network telemetry over sampling due to their lightweight and approximate nature. Unfortunately, due to network flow characteristics and hardware limitations, network telemetry focuses on counting sketches, e.g., count-min, whereas μView explores a distinctly different sketch optimized for online anomaly detection. Recently, PromSketch [118] proposed an approximate query cache based on sketches for time-series data, but its focus is on reducing query latencies and costs on the collector nodes, whereas μView targets lightweight real-time analytics on the workload nodes.

9 Conclusion

We presented μView, a system to improve observability by increasing metrics analysis resolution. Engineers can run μView with custom classifiers as well as register custom observability hooks. μView triggers actionable insights about microservices' state. μView reduces operational burden through the use of a catch-all anomaly score threshold, improves explainability by highlighting relevant metrics that contribute to anomalies, and enables substantial cost savings with negligible loss in measurement accuracy. μView also increases the coverage of faulty requests compared to OpenTelemetry head sampling policies, while keeping the overhead low. We deployed μView on a BlueField-2 IPU and demonstrated it can read, process, analyze and export metrics from hundreds of microservices simultaneously, while not saturating all IPU cores.

Acknowledgments

We thank our shepherd, Alan Zaoxing Liu, and the anonymous reviewers for their helpful feedback.

References

[1] The next wave of google cloud infrastructure innovation: New c3 vm and hyperdisk. https://cloud.google.com/blog/products/compute/introducing-c3-machines-with-googles-custom-intel-ipu, 2022.

[2] Chaosmesh. https://chaos-mesh.org/, 2023.

[3] Decomposing twitter: Adventures in service-oriented architecture. https://www.infoq.com/presentations/twitter-soa/, 2023.

[4] Ec2 instance types. https://aws.amazon.com/ec2/instance-types/, 2023.

[5] Gitlab issue #1601. https://gitlab.com/gitlab-com/gl-infra/scalability/-/issues/1601#top, 2023.

[6] Growth engineering at netflix — accelerating innovation. https://netflixtechblog.com/growth-engineering-at-netflix-accelerating-innovation-90eb8e70ce59, 2023.

[7] Intel infrastructure processing unit (intel ipu) asic e2000. https://www.intel.com/content/www/us/en/products/details/network-io/ipu/e2000-asic.html, 2023.

[8] Introducing domain-oriented microservice architecture. https://www.uber.com/en-PT/blog/microservice-architecture/, 2023.

[9] Istio. https://istio.io/, 2023.

[10] Jaeger. https://www.jaegertracing.io/, 2023.

[11] M3: Uber's open source, large-scale metrics platform for prometheus. https://www.uber.com/en-PT/blog/m3/, 2023.

[12] Nvidia bluefield networking platform. https://www.nvidia.com/en-us/networking/products/data-processing-unit/, 2023.

[13] Online boutique. https://github.com/GoogleCloudPlatform/microservices-demo, 2023.

[14] Opentelemetry. https://opentelemetry.io/, 2023.

[15] Opentelemtry — sampling. https://opentelemetry.io/docs/concepts/sampling/, 2023.

[16] Prometheus: Monitoring system & time series database. https://prometheus.io/, 2023.

[17] Zipkin. https://zipkin.io/, 2023.

[18] Aws managed service for prometheus. https://aws.amazon.com/prometheus/pricing/#Pricing, 2024.

[19] Phillipe Ajoux, Nathan Bronson, Sanjeev Kumar, Wyatt Lloyd, and Kaushik Veeraraghavan. Challenges to adopting stronger consistency at scale. In *HotOS*. USENIX Association, 2015.

[20] Pradeep Ambati, Inigo Goiri, Felipe Frujeri, Alper Gun, Ke Wang, Brian Dolan, Brian Corell, Sekhar Pasupuleti, Thomas Moscibroda, Sameh Elnikety, Marcus Fontoura, and Ricardo Bianchini. Providing SLOs for Resource-Harvesting VMs in cloud platforms. In *USENIX OSDI*. USENIX Association, 2020.

[21] Vaastav Anand, Deepak Garg, Antoine Kaufmann, and Jonathan Mace. Blueprint: A toolchain for highly-reconfigurable microservice applications. In *SOSP*. Association for Computing Machinery, 2023.

[22] Sundaram Ananthanarayanan, Masoud Saeida Ardekani, Denis Haenikel, Balaji Varadarajan, Simon Soriano, Dhaval Patel, and Ali-Reza Adl-Tabatabai. Keeping master green at scale. In *ACM EuroSys*, EuroSys '19, New York, NY, USA, 2019. Association for Computing Machinery.

[23] Dan Ardelean, Amer Diwan, and Chandra Erdman. Performance analysis of cloud applications. In *USENIX NSDI*. USENIX Association, 2018.

[24] Open Telemetry Authors. Open telemetry collector. https://opentelemetry.io/docs/collector/transforming-telemetry/, 2025.

[25] The Jaeger Authors. Jaeger tracing sampling. https://www.jaegertracing.io/docs/1.27/sampling/, 2025.

[26] Utkarsh Ayachit. Exploring an automated testing strategy for infrastructure as code. https://techcommunity.microsoft.com/t5/azure-high-performance-computing/exploring-an-automated-testing-strategy-for-infrastructure-as/ba-p/3971715, 2023.

[27] Luiz André Barroso, Jimmy Clidaras, and Urs Hölzle. The datacenter as a computer: An introduction to the design of warehouse-scale machines. *Synthesis lectures on computer architecture*, 8(3):1–154, 2013.

[28] Nathan Bronson, Abutalib Aghayev, Aleksey Charapko, and Timothy Zhu. Metastable failures in distributed systems. In *HotOS*. Association for Computing Machinery, 2021.

[29] Marco Spaziani Brunella, Giacomo Belocchi, Marco Bonola, Salvatore Pontarelli, Giuseppe Siracusano, Giuseppe Bianchi, Aniello Cammarano, Alessandro Palumbo, Luca Petrucci, and Roberto Bifulco. hXDP: Efficient software packet processing on FPGA NICs. In *USENIX OSDI*. USENIX Association, 2020.

[30] Mike Chow, Yang Wang, William Wang, Ayichew Hailu, Rohan Bopardikar, Bin Zhang, Jialiang Qu, David Meisner, Santosh Sonawane, Yunqi Zhang, Rodrigo Paim, Mack Ward, Ivor Huang, Matt McNally, Daniel Hodges, Zoltan Farkas, Caner Gocmen, Elvis Huang, and Chunqiang Tang. ServiceLab: Preventing tiny performance regressions at hyperscale through Pre-Production testing. In *USENIX OSDI*, 2024.

[31] community. stress-ng (stress next generation). https://github.com/ColinIanKing/stress-ng, 2024.

[32] Christina Delimitrou. DeathStarBench benchmark suite for cloud microservices. https://github.com/delimitrou/DeathStarBench, 2024.

[33] Úlfar Erlingsson, Marcus Peinado, Simon Peter, and Mihai Budiu. Fay: Extensible distributed tracing from kernels to clusters. In *SOSP*, SOSP '11, page 311–326, New York, NY, USA, 2011. Association for Computing Machinery.

[34] Daniel Firestone, Andrew Putnam, Sambhrama Mundkur, Derek Chiou, Alireza Dabagh, Mike Andrewartha, Hari Angepat, Vivek Bhanu, Adrian Caulfield, Eric Chung, et al. Azure accelerated networking:{SmartNICs} in the public cloud. In *USENIX NSDI*. USENIX Association, 2018.

[35] Rodrigo Fonseca, George Porter, Randy H. Katz, and Scott Shenker. X-Trace: A pervasive network tracing framework. In *USENIX NSDI*. USENIX Association, 2007.

[36] Cloud Native Computing Foundation. Envoy proxy. https://www.envoyproxy.io, 2024.

[37] Yu Gan and Christina Delimitrou. The architectural implications of cloud microservices. *IEEE Computer Architecture Letters*, 17(2):155–158, 2018.

[38] Yu Gan, Mingyu Liang, Sundar Dev, David Lo, and Christina Delimitrou. Sage: practical and scalable ml-driven performance debugging in microservices. ASPLOS '21, pages 135–151. Association for Computing Machinery, 2021.

[39] Yu Gan, Yanqi Zhang, Kelvin Hu, Dailun Cheng, Yuan He, Meghna Pancholi, and Christina Delimitrou. Seer: Leveraging big data to navigate the complexity of performance debugging in cloud microservices. In *ASPLOS*, ASPLOS '19, page 19–33, New York, NY, USA, 2019. Association for Computing Machinery.

[40] Mina Ghashami, Edo Liberty, and Jeff M. Phillips. Efficient frequent directions algorithm for sparse matrices. In *ACM KDD*. Association for Computing Machinery, 2016.

[41] Mina Ghashami, Edo Liberty, and Jeff M. Phillips. Efficient frequent directions algorithm for sparse matrices. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '16, page 845–854, New York, NY, USA, 2016. Association for Computing Machinery.

[42] Google. cadvisor. https://github.com/google/cadvisor, 2024.

[43] Boris Grubic, Yang Wang, Tyler Petrochko, Ran Yaniv, Brad Jones, David Callies, Matt Clarke-Lauer, Dan Kelley, Soteris Demetriou, Kenny Yu, and Chunqiang Tang. Conveyor: One-Tool-Fits-All continuous software deployment at meta. In *USENIX OSDI*. USENIX Association, 2023.

[44] Arpit Gupta, Rob Harrison, Marco Canini, Nick Feamster, Jennifer Rexford, and Walter Willinger. Sonata: Query-driven streaming network telemetry. In *ACM SIGCOMM*, SIGCOMM '18, page 357–371, New York, NY, USA, 2018. Association for Computing Machinery.

[45] Vipul Harsh, Wenxuan Zhou, Sachin Ashok, Radhika Niranjan Mysore, Brighten Godfrey, and Sujata Banerjee. Murphy: Performance diagnosis of distributed cloud applications. In *ACM SIGCOMM*. Association for Computing Machinery, 2023.

[46] Haiyu Huang, Xiaoyu Zhang, Pengfei Chen, Zilong He, Zhiming Chen, Guangba Yu, Hongyang Chen, and Chen Sun. Trastrainer: Adaptive sampling for distributed traces with system runtime state. *Proceedings of the ACM Software Engineering*, 1(FSE), jul 2024.

[47] Hao Huang and Shiva Prasad Kasiviswanathan. Streaming anomaly detection using randomized matrix sketching. *VLDB Endowment*, 2015.

[48] L. Huang, X. Nguyen, M. Garofalakis, J. M. Hellerstein, M. I. Jordan, A. D. Joseph, and N. Taft. Communication-efficient online detection of network-wide anomalies. In *IEEE INFOCOM*, pages 134–142. IEEE, 2007.

[49] Lexiang Huang, Matthew Magnusson, Abishek Bangalore Muralikrishna, Salman Estyak, Rebecca Isaacs, Abutalib Aghayev, Timothy Zhu, and Aleksey Charapko. Metastable failures in the wild. In *OSDI 22*. USENIX Association, 2022.

[50] Peng Huang, Chuanxiong Guo, Lidong Zhou, Jacob R. Lorch, Yingnong Dang, Murali Chintalapati, and Randolph Yao. Gray failure: The achilles' heel of cloud-scale systems. In *HotOS*, 2017.

[51] Tao Huang, Pengfei Chen, and Ruipeng Li. A semi-supervised vae based active anomaly detection framework in multivariate time series for online systems. In *Proceedings of the ACM Web Conference 2022*, WWW '22, 2022.

[52] Zicheng Huang, Pengfei Chen, Guangba Yu, Hongyang Chen, and Zibin Zheng. Sieve: Attention-based sampling of end-to-end trace data in distributed microservice systems. In *IEEE ICWS*, pages 436–446, 2021.

[53] Darby Huye, Yuri Shkuro, and Raja R. Sambasivan. Lifting the veil on Meta's microservice architecture: Analyses of topology and request workflows. In *USENIX ATC*. USENIX Association, 2023.

[54] Joab Jackson. Debugging microservices: Lessons from google, facebook, lyft. "https://thenewstack.io/debugging-microservices-lessons-from-google-facebook-lyft/", July 2018.

[55] Jongyul Kim, Insu Jang, Waleed Reda, Jaeseong Im, Marco Canini, Dejan Kostić, Youngjin Kwon, Simon Peter, and Emmett Witchel. LineFS: Efficient smartnic offload of a distributed file system with pipeline parallelism. In *SOSP*, pages 756–771. Association for Computing Machinery, 2021.

[56] Myunghwan Kim, Roshan Sumbaly, and Sam Shah. Root cause detection in a service-oriented architecture. *SIGMETRICS Perform. Eval. Rev.*, 41(1):93–104, jun 2013.

[57] Diederik P Kingma and Max Welling. Auto-encoding variational bayes, 2022.

[58] Anukool Lakhina, Mark Crovella, and Christiphe Diot. Characterization of network-wide anomalies in traffic flows. In *IMC*. Association for Computing Machinery, 2004.

[59] Pedro Las-Casas, Giorgi Papakerashvili, Vaastav Anand, and Jonathan Mace. Sifter: Scalable sampling for distributed traces, without feature engineering. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '19. Association for Computing Machinery, 2019.

[60] Yanfang Le, Hyunseok Chang, Sarit Mukherjee, Limin Wang, Aditya Akella, Michael M. Swift, and T. V. Lakshman. Uno: Uniflying host and smart nic offload for flexible packet processing. In *SoCC*, SoCC '17, page 506–519, New York, NY, USA, 2017. Association for Computing Machinery.

[61] Greg Leffler. The hidden cost of sampling in observability. "https://www.splunk.com/en_us/blog/devops/the-hidden-cost-of-sampling-in-observability.html", May 2021.

[62] Tomer Levy. There's no value in observability bloat. let's focus on the essentials. https://devops.com/theres-no-value-in-observability-bloat-lets-focus-on-the-essentials/, 2023.

[63] Kevin Lew and Sangeeta Narayanan. Lessons from building observability tools at netflix. "https://netflixtechblog.com/lessons-from-building-observability-tools-at-netflix-7cfafed6ab17", June 2018.

[64] Junru Li, Youyou Lu, Qing Wang, Jiazhen Lin, Zhe Yang, and Jiwu Shu. AlNiCo: SmartNIC-accelerated contention-aware request scheduling for transaction processing. In *USENIX ATC*. USENIX Association, 2022.

[65] Edo Liberty. Simple and deterministic matrix sketching. In *ACM KDD*. Association for Computing Machinery, 2013.

[66] Ming Liu, Tianyi Cui, Henry Schuh, Arvind Krishnamurthy, Simon Peter, and Karan Gupta. Offloading distributed applications onto SmartNICs using iPipe. In *ACM SIGCOMM*, page 318–333, New York, NY, USA, 2019. Association for Computing Machinery.

[67] Ming Liu, Simon Peter, Arvind Krishnamurthy, and Phitchaya Mangpo Phothilimthana. E3: Energy-Efficient microservices on SmartNIC-Accelerated servers. In *USENIX ATC*, pages 363–378, Renton, WA, July 2019. USENIX Association.

[68] Zaoxing Liu, Ran Ben-Basat, Gil Einziger, Yaron Kassner, Vladimir Braverman, Roy Friedman, and Vyas Sekar. Nitrosketch: Robust and general sketch-based monitoring in software switches. In *ACM SIGCOMM*, SIGCOMM '19, page 334–350, New York, NY, USA, 2019. Association for Computing Machinery.

[69] Zaoxing Liu, Hun Namkung, Georgios Nikolaidis, Jeongkeun Lee, Changhoon Kim, Xin Jin, Vladimir Braverman, Minlan Yu, and Vyas Sekar. Jaqen: A high-performance switch-native approach for detecting and mitigating volumetric ddos attacks with programmable switches. In *USENIX Security*. USENIX Association, 2021.

[70] Chang Lou, Cong Chen, Peng Huang, Yingnong Dang, Si Qin, Xinsheng Yang, Xukun Li, Qingwei Lin, and Murali Chintalapati. RESIN: A holistic service for dealing with memory leaks in production cloud infrastructure. In *OSDI 22*, 2022.

[71] Alex Marchioni, Luciano Prono, Mauro Mangia, Fabio Pareschi, Riccardo Rovatti, and Gianluca Setti. Streaming algorithms for subspace analysis: Comparative review and implementation on iot devices. *IEEE Internet of Things Journal*, 2023.

[72] Dirk Merkel. Docker: lightweight Linux containers for consistent development and deployment. *Linux Journal*, 2014(239), 2014.

[73] Meta. Pytorch. https://pytorch.org, 2025.

[74] Ulrich Mike. Addressing cascading failures. https://sre.google/sre-book/addressing-cascading-failures/.

[75] George Miranda. A sample of sampling, and a whole lot of observability at scale. "https://www.honeycomb.io/blog/sampling-observability-slack", Jan 2023.

[76] MongoDB. MongoDB. https://www.mongodb.com/resources/products/capabilities/how-to-monitor-mongodb-and-what-metrics-to-monitor, 2024.

[77] Hun Namkung, Zaoxing Liu, Daehyeok Kim, Vyas Sekar, and Peter Steenkiste. Sketchovsky: Enabling ensembles of sketches on programmable switches. In *USENIX NSDI 23*. USENIX Association, 2023.

[78] Srinivas Narayana, Anirudh Sivaraman, Vikram Nathan, Prateesh Goyal, Venkat Arun, Mohammad Alizadeh, Vimalkumar Jeyakumar, and Changhoon Kim. Language-directed hardware design for network performance monitoring. In *ACM SIGCOMM*. Association for Computing Machinery, 2017.

[79] NetData. Netdata. https://www.google.com/search?client=safari&rls=en&q=netdata&ie=UTF-8&oe=UTF-8, 2024.

[80] Francisco Neves, Nuno Machado, and JosÃ© Pereira. Falcon: A practical log-based analysis tool for distributed systems. In *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 534–541, 2018.

[81] Nvidia. Doca sdk. https://developer.nvidia.com/networking/doca, 2024.

[82] Adam Oliner, Archana Ganapathi, and Wei Xu. Advances and challenges in log analysis. *Commun. ACM*, 55(2):55–61, feb 2012.

[83] Shixiong Qi, Leslie Monis, Ziteng Zeng, Ian-chin Wang, and K. K. Ramakrishnan. Spright: extracting the server from serverless computing! high-performance ebpf-based event-driven, shared-memory processing. In *ACM SIGCOMM*, 2022.

[84] Haoran Qiu, Subho S. Banerjee, Saurabh Jha, Zbigniew T. Kalbarczyk, and Ravishankar K. Iyer. FIRM: An intelligent fine-grained resource management framework for SLO-Oriented microservices. In *USENIX OSDI*. USENIX Association, 2020.

[85] Kamala Ramasubramanian, Ashutosh Raina, Jonathan Mace, and Peter Alvaro. Act now: Aggregate comparison of traces for incident localization, 2022.

[86] Redis. Redis. https://redis.io, 2024.

[87] Raja R. Sambasivan, Alice X. Zheng, Michael De Rosa, Elie Krevat, Spencer Whitman, Michael Stroucken, William Wang, Lianghong Xu, and Gregory R. Ganger. Diagnosing performance changes by comparing request flows. In *USENIX NSDI*. USENIX Association, 2011.

[88] Harshit Saokar, Soteris Demetriou, Nick Magerko, Max Kontorovich, Josh Kirstein, Margot Leibold, Dimitrios Skarlatos, Hitesh Khandelwal, and Chunqiang Tang. ServiceRouter: Hyperscale and minimal cost service mesh at meta. In *USENIX OSDI*. USENIX Association, 2023.

[89] M. Saquib Sarfraz, Mei-Yen Chen, Lukas Layer, Kunyu Peng, and Marios Koulakis. Position: quo vadis, unsupervised time series anomaly detection? In *Proceedings of the 41st International Conference on Machine Learning*, ICML'24, 2024.

[90] Henry N. Schuh, Weihao Liang, Ming Liu, Jacob Nelson, and Arvind Krishnamurthy. Xenic: SmartNIC-accelerated distributed transactions. In *ACM SIGOPS*, SOSP '21, page 740–755, New York, NY, USA, 2021. Association for Computing Machinery.

[91] Bernhard Schölkopf, John Platt, and Thomas Hofmann. *In-Network PCA and Anomaly Detection*, pages 617–624. 2007.

[92] Ben Sigelman. Three pillars of observability with zero answers | lightstep blog. "https://lightstep.com/blog/three-pillars-zero-answers-towards-new-scorecard-observability", December 2018.

[93] Benjamin H Sigelman, Luiz André Barroso, Mike Burrows, Pat Stephenson, Manoj Plakal, Donald Beaver, Saul Jaspan, and Chandan Shanbhag. Dapper, a large-scale distributed systems tracing infrastructure, 2010.

[94] Akshitha Sriraman and Thomas F Wenisch. μ suite: a benchmark suite for microservices. In *2018 IEEE International Symposium on Workload Characterization (IISWC)*, pages 1–12. IEEE, 2018.

[95] Naru Sundar, Brad Burres, Yadong Li, Dave Minturn, Brian Johnson, and Nupur Jain. 9.4 an in-depth look at the intel ipu e2000. In *ISSCC*, 2023.

[96] IOP Systems. Rezolus: High-resolution systems performance telemetry. https://github.com/iopsystems/rezolus.

[97] Hound Technology. Honeycomb filter processor. https://docs.honeycomb.io/manage-data-volume/filter/filter-processor/, 2025.

[98] Jörg Thalheim, Antonio Rodrigues, Istemi Ekin Akkus, Pramod Bhatotia, Ruichuan Chen, Bimal Viswanath, Lei Jiao, and Christof Fetzer. Sieve: Actionable insights from monitored metrics in distributed systems. In *ACM/IFIP/USENIX Middleware Conference*, pages 14–27, 2017.

[99] Bingmann Timo. pmbw - Parallel Memory Bandwidth Benchmark / Measurement. https://panthema.net/2013/pmbw/, 2024.

[100] Mert Toslali, Emre Ates, Alex Ellis, Zhaoqi Zhang, Darby Huye, Lan Liu, Samantha Puterman, Ayse K. Coskun, and Raja R. Sambasivan. Automating instrumentation choices for performance problems in distributed applications with VAIF. In *ACM SoCC*, SoCC '21, page 61–75, New York, NY, USA, 2021. Association for Computing Machinery.

[101] Abhishek Verma, Luis Pedrosa, Madhukar R. Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. Large-scale cluster management at Google with Borg. In *EuroSys*. Association for Computing Machinery, 2015.

[102] Zeke Wang, Hongjing Huang, Jie Zhang, Fei Wu, and Gustavo Alonso. FpgaNIC: An FPGA-based versatile 100gb SmartNIC for GPUs. In *USENIX ATC*. USENIX Association, 2022.

[103] Zexin Wang, Changhua Pei, Minghua Ma, Xin Wang, Zhihan Li, Dan Pei, Saravan Rajmohan, Dongmei Zhang, Qingwei Lin, Haiming Zhang, Jianhui Li, and Gaogang Xie. Revisiting vae for unsupervised time series anomaly detection: A frequency perspective, 2024.

[104] Zexin Wang, Changhua Pei, Minghua Ma, Xin Wang, Zhihan Li, Dan Pei, Saravan Rajmohan, Dongmei Zhang, Qingwei Lin, Haiming Zhang, Jianhui Li, and Gaogang Xie. Revisiting vae for unsupervised time series anomaly detection: A frequency perspective. In *ACM WWW 2024*, New York, NY, USA, 2024.

[105] Zhe Wang, Teng Ma, Linghe Kong, Zhenzao Wen, Jingxuan Li, Zhuo Song, Yang Lu, Guihai Chen, and Wei Cao. Zero overhead monitoring for cloud-native infrastructure using RDMA. In *USENIX ATC*. USENIX Association, 2022.

[106] Xingda Wei, Rongxin Cheng, Yuhan Yang, Rong Chen, and Haibo Chen. Characterizing off-path SmartNIC for accelerating distributed systems. In *USENIX OSDI*, 2023.

[107] Li Wu, Johan Tordsson, Erik Elmroth, and Odej Kao. Microrca: Root cause localization of performance issues in microservices. In *IEEE NOMS*, 2020.

[108] Jiarong Xing, Yiming Qiu, Kuo-Feng Hsu, Songyuan Sui, Khalid Manaa, Omer Shabtai, Yonatan Piasetzky, Matty Kadosh, Arvind Krishnamurthy, T. S. Eugene Ng, and Ang Chen. Unleashing smartnic packet processing performance in p4. In *ACM SIGCOMM*, ACM SIGCOMM '23, page 1028–1042, New York, NY, USA, 2023. Association for Computing Machinery.

[109] Asaf Yigal. Three observability trends that will resonate in 2024 – and what to do about them. https://vmblog.com/archive/2023/11/22/logz-io-2024-predictions-three-observability-trends-that-will-resonate-in-2024-and-what-to-do-about-them.aspx, 2023.

[110] Dong Young Yoon, Yang Wang, Miao Yu, Elvis Huang, Juan Ignacio Jones, Abhinay Kukkadapu, Osman Kocas, Jonathan Wiepert, Kapil Goenka, Sherry Chen, Yanjun Lin, Zhihui Huang, Jocelyn Kong, Michael Chow, and Chunqiang Tang. Fbdetect: Catching tiny performance regressions at hyperscale through in-production monitoring. In *ACM SOSP*, 2024.

[111] Guangba Yu, Pengfei Chen, Hongyang Chen, Zijie Guan, Zicheng Huang, Linxiao Jing, Tianjun Weng, Xinmeng Sun, and Xiaoyun Li. Microrank: End-to-end latency issue localization with extended spectrum analysis in microservice environments. In *ACM Web Conference 2021*, pages 3087–3098, 2021.

[112] Ding Yuan, Haohui Mai, Weiwei Xiong, Lin Tan, Yuanyuan Zhou, and Shankar Pasupathy. SherLog: Error diagnosis by connecting clues from run-time logs. In *ACM ASPLOS*, page 143–154, New York, NY, USA, 2010. Association for Computing Machinery.

[113] Chaoli Zhang, Tian Zhou, Qingsong Wen, and Liang Sun. Tfad: A decomposition time series anomaly detection architecture with time-frequency analysis. page 2497–2507. ACM, 2022.

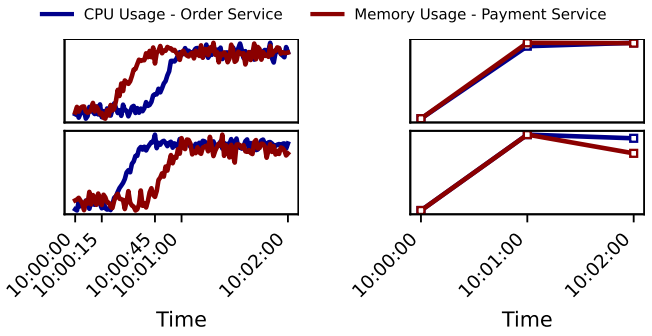
[114] Lei Zhang, Zhiqiang Xie, Vaastav Anand, Ymir Vigfusson, and Jonathan Mace. The benefit of hindsight: Tracing Edge-Cases in distributed systems. In *USENIX NSDI*. USENIX Association, 2023.

[115] Yanqi Zhang, Íñigo Goiri, Gohar Irfan Chaudhry, Rodrigo Fonseca, Sameh Elnikety, Christina Delimitrou, and Ricardo Bianchini. Faster and cheaper serverless computing on harvested resources. In *ACM SOSP*. Association for Computing Machinery, 2021.

[116] Yongle Zhang, Kirk Rodrigues, Yu Luo, Michael Stumm, and Ding Yuan. The inflection point hypothesis: a principled debugging approach for locating the root cause of a failure. In *SOSP*. Association for Computing Machinery, 2019.

[117] Xiang Zhou, Xin Peng, Tao Xie, Jun Sun, Chao Ji, Dewei Liu, Qilin Xiang, and Chuan He. Latent error prediction and fault localization for microservice applications by learning from system trace logs. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2019, page 683–694, New York, NY, USA, 2019. Association for Computing Machinery.

[118] Zeying Zhu, Jonathan Chamberlain, Kenny Wu, David Starobinski, and Zaoxing Liu. Approximation-first timeseries query at scale. *Proc. VLDB Endow.*, 2025.



(a) 1 sec. sampling **(b) 1 min. sampling**

Figure 10: Different anomalies propagation patterns — *e.g.*, Order → Payment (bottom) and Payment → Order (top) — may appear identical with coarse-grained monitoring.

A Resolution matters

Fig. 10 illustrates how two distinct failure scenarios can appear identical when observed through traditional monitoring systems with coarse-grained sampling. In Fig. 10a (top), at 10:00:15, the Payment Service experiences a memory leak, causing its memory usage to increase in 15 seconds. This degradation causes the Payment Service to respond slowly. By 10:00:30, the Order Service begins experiencing CPU pressure as it accumulates a backlog of requests and initiates retries, with its CPU utilization rising. In Fig. 10b (bottom), the sequence begins differently. At 10:00:15, the Order Service experiences a sudden traffic spike, causing its CPU utilization to jump. The flood of requests from the Order Service then overwhelms the Payment Service. By 10:00:45, the Payment Service's memory usage escalates to high values as its connection pool becomes exhausted. When sampled at the standard one-minute interval (10:01:00), both scenarios present almost identical metrics, which makes it very hard to identify the root cause with traditional monitoring. Fine-grained temporal analysis preserves critical timing relationships, enabling an accurate diagnosis.

B Ingestion vs Generation experiment

Experiment Setup In Fig. 2, we *(i)* focus on dissecting the per-node CPU overhead due to generation and ingestion of metrics, respectively, and *(ii)* evaluate how the overhead changes for higher sampling rates. For this experiment, we deployed a cAdvisor instance and a Prometheus metrics collector on two separate nodes (details in § 6.2). cAdvisor generates 105 system-level metrics for Docker containers running on the same node, and is queried periodically by Prometheus running on a separate node. We tune the granularity of generation and ingestion by changing the `housekeeping_interval` in cAdvisor and `scraping_interval` in Prometheus, respectively. We evaluate the average CPU cores consumed by cAdvisor for different numbers of containers. Each experiment lasts for 5 minutes.

C Sketch analysis pipeline

C.1 Summary of relevant notation

Table 4 summarizes the relevant notation used in § 4.1.

Symbol	Description
m	Sketch rows
ℓ	Sketch columns
k	rank-k SVD truncation
$\boldsymbol{\alpha}_t$	Anomaly score at t-th sampling time
γ	Anomaly score threshold
η	Learning rate

Table 4: Summary of relevant FD-sketch notation.

C.2 Offline configuration workflow

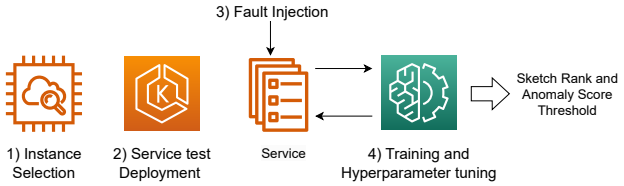
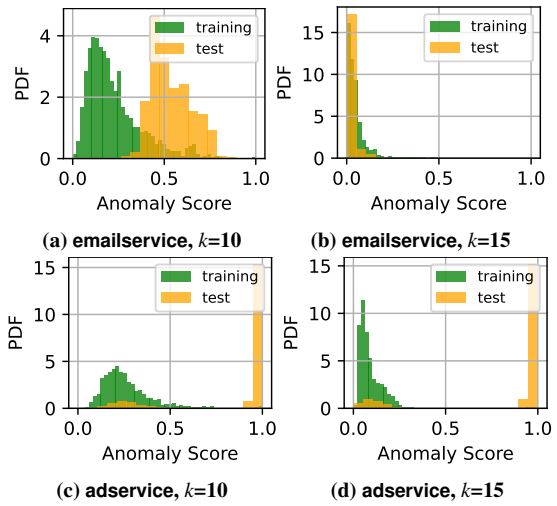


Figure 11: Offline configuration workflow.

Fig. 11 illustrates the offline configuration workflow for using μView. Following best practices [4, 26], we expect that Ops teams run automated tests to validate deployments before service instances enter production. We leverage this phase to train the μ-MONITORs and tune their hyperparameters: namely, the rank k of the sketch-based classifier (explained in § 4) and the anomaly score threshold γ. This is done for each μ-MONITOR, which corresponds to the number of microservices.

We start by collecting a training dataset of metrics, which we assume to be free of anomalies. We complement the training dataset with a set of metrics collected while synthetic faults are injected (validation dataset). The proportion of training and validation datasets is 70% and 30%, respectively.

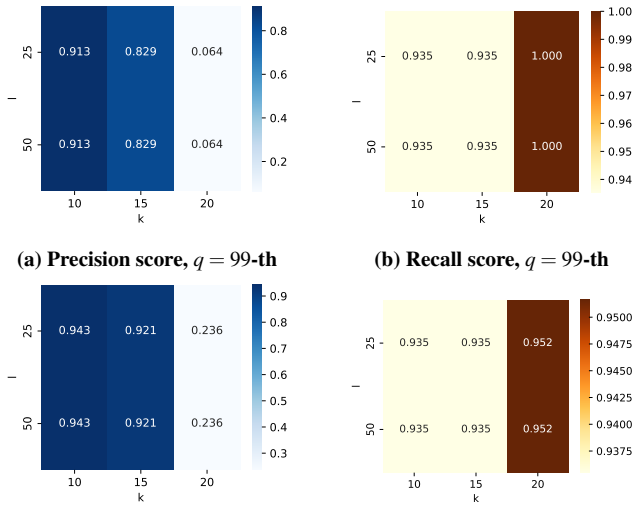
The hyperparameters are chosen by grid-searching possible values as follows. To pick the threshold γ, we consider the distribution of anomaly scores over the training dataset and set γ to a sufficiently large percentile (e.g., 90-99th). We compute the F1-score over the validation set. We repeat this process for each value of (k, l) and pick the ones that maximize the F1-score (we elaborate in Appendix C.3).



(a) **emailservice, k=10** (b) **emailservice, k=15**

(c) **adservice, k=10** (d) **adservice, k=15**

Figure 12: Anomaly score distributions of the training and test datasets for different services of the OnlineBoutique application and different sketch size k (but same number of metrics). The two services exhibit very different behaviors for the same parameters, highlighting the challenge faced by μView of per-service tuning.



(a) **Precision score, $q = 99$-th** (b) **Recall score, $q = 99$-th**

(c) **Precision score, $q = 99.9$-th** (d) **Recall score, $q = 99.9 - th$**

Figure 13: Impact of varying sketch parameters on *precision* (blue color map) and *recall* (orange color map) metrics. The anomaly detection threshold is set to the q-th percentile of the anomaly score distribution, i.e., $\gamma = F(q)$.

Recall μView's sketch-based approach is adaptive to workload changes (§ 4). We do not expect to retrain μ-MONITORs unless the workload changes drastically. In practice, we expect to retrain the μ-MONITORs only when the application code is updated.

C.3 Hyperparameter tuning guidelines

We share our experience in tuning FD-sketch hyperparameters, according to the methodology in Appendix C.2. The goal of this section is to highlight the unique challenges we faced while applying FD-sketch to our specific microservice domain, which prevented us from a straightforward adoption of the sketch with a static configuration as per original guidelines [65]. Our main finding is that different microservices require individually fine-tuned sketch parameters and exhibit very different behaviors in terms of anomaly score, even for the same parameters. This leads to the need for per-service tuning, as implemented by μView.

For these micro-benchmarks, we deployed the OnlineBoutique application and deterministically injected anomalies only on a controlled subset of services. We report results for two representative services, which were not touched by anomalies. We fixed the sketch size l and varied the SVD rank k, *i.e.*, the truncation size in SVD. Only k components will be used by the detector when trying to reconstruct an input vector. Since we assume the training data contains a baseline of anomaly-free metric samples (i.e., non-anomalous), a natural choice is to derive a threshold around the tail (i.e., high percentile) of the anomaly score distribution. We walk through the mutual relationship of these parameters in Figure 12. Here we show the results for different values of k and different services.

Considering small values of k (Figs. 12a and 12c), we note that the sketch tends to underfit the training data. This behavior is more evident for the `adservice`, where the tail of the training score distribution approaches 1, and largely overlaps with the test distribution (i.e., the two distributions have low KL-divergence). Under these circumstances, the definition of the threshold γ is challenging. If γ is set to a high percentile, the number of false negatives increases quickly, whereas a smaller γ would generate several false positives. On the other hand, for $k = 15$ (Fig. 12d) the training distribution is more concentrated around 0 and its support does not overlap with the test distribution. This way, it's easy to choose γ around the tail of the training distribution.

Next, we note that the sketches exhibit radically different behavior for the two services, even for the same choice of rank k[10]. Remember that neither of the two services contains anomalies in the test data. Thus, a good classifier would behave like in Fig. 12a, with the training and test distribution closely overlapping. For the `emailservice`, both choices of k lead to a healthy distribution for the classifier. On the contrary, for the `adservice` the training score distribution is extremely skewed. In this case, the sketch doesn't generalize well to different services for the same parameters, which motivates our per-service tuning and design.

Detection performance. We vary the sketch reconstruction

[10]The choice of k as per original guidelines [47] depends on the number of input metrics m. We collect the same number of metrics for the two services we highlight

rank k, and the sketch size l, and measure the precision and recall of the classifier for different anomaly detection thresholds derived from the q-percentile of the anomaly score distribution. We report the results in Figure 13. We observe that increasing k gives better recall but lower precision. This is because increasing k leads to a skewed anomaly score distribution, which intuitively corresponds to higher sensitivity of the sketch to metrics variations and ultimately leads to overfitting. For example, for $k = 20$, a small noise in the metrics of the test data produces large reconstruction errors, which increases the number of false positives and reduces precision. This effect is mitigated by choosing a higher anomaly score threshold. On the other hand, we observe that the recall score benefits from more skewed anomaly score distribution, but is generally less sensitive to the parameter k . Overall, we observe that both precision and recall do not significantly change when increasing the sketch size from $l = 25$ to $l = 50$. We conclude that the choice of k and γ should be made based on the desired trade-off between precision and recall.

```
def IPU_processing_loop():

    # Users can provide custom detection models
    micromonitor = MicroMonitor(model = 'FD-sketch')
    while True:
        # RDMA read metrics
        metrics = rdma_read(MR.host_addr, MR.rkey)

        # Apply streaming classification and invoke hooks
        ↪  (e.g, filters)
        out = micromonitor.classify(metrics):
        if anomaly(out):
            micromonitor.hook(metrics)

        # Tunable local processing frequency
        sleep(local_scraping_interval)
```

Figure 14: Simplified pseudo-code of μView IPU local processing in the IPU, for a single microservice.