Opportunistic Telemetry Transport in Hardware-Accelerated Observability Pipelines

Hadj Ahmed Chikh Dahmane
KAUST

Alessandro Cornacchia
KAUST

Marco Canini
KAUST

Abstract—**High-frequency telemetry is crucial to understand performance and failures in microservice-based cloud systems, but exporting metrics at sub-second granularity can contend with application traffic and saturate shared I/O resources such as PCIe, even when collection is offloaded to SmartNICs or Infrastructure Processing Units (IPUs). Existing designs typically rely on dedicated telemetry packets or frequent RDMA reads, which introduce extra system calls, DMA operations, and PCIe transactions. This work-in-progress paper explores an alternative, event-driven design: opportunistically collecting and transporting host metrics by piggybacking them on sub-MTU application packets already destined for the network. We attach eBPF programs to the transmit path so that, when a sub-MTU packet is sent, the kernel can *reuse* this I/O event to (1) read system metrics – thus amortizing the cost of context switches – and (2) embed telemetry before the packet traverses PCIe – thus avoiding new transactions for telemetry packets; the SmartNIC then strips and aggregates this metadata for downstream analysis. We sketch the architecture of this opportunistic telemetry transport, outline key challenges, and present an initial evaluation to quantify overheads in containerized microservice workloads.**

I. Introduction

Modern distributed systems rely on observability to understand performance and ensure they meet service-level objectives. From microservice-based cloud applications to emerging LLM inference frameworks [1] and GPU clusters, operators collect and analyze telemetry data to detect anomalies, diagnose regressions, and maintain predictable behavior.

Metrics time-series are a first-class telemetry category, for which fine-grained collection is essential for capturing transient events, invisible to low-frequency monitoring. Real-time use cases such as autoscaling, anomaly detection, and adaptive load balancing benefit from sub-second visibility into system state; without it, operators may react too late to load spikes or fail to catch short-horizon excursions that underpin SLO violations.

Conventional solutions rely on user-space polling mechanisms [2] to extract (*i.e.*, access) system metrics. Alternatively, extended Berkeley Packet Filter (eBPF) technology enables telemetry reading directly within kernel-space, reducing context switches and potentially lowering CPU overheads.

Regardless on how metrics are extracted on the host system, metrics are then collected for downstream usage, either by dedicated monitors on remote servers, or by accelerators, e.g., on the NIC SoC, where they serve as real-time input signals to offloaded functionalities [3], [4]. During metrics collection, data movements along the host-NIC datapath are involved. When operating at high sampling frequencies, these data transfers create pressure on the host-to-NIC datapath, and contend with the very workloads being observed which share the same network and I/O resources. Consequently, the latency tail of cloud-native applications might be compromised by real-time metrics observability. Because operators increasingly push toward real-time visibility [5], understanding and mitigating transport overheads is essential to achieving fine-grained observability without degrading application performance.

We propose the idea of *opportunistic telemetry*: rather than generating dedicated telemetry packets, we piggyback metrics onto sub-MTU packets that regularly traverse the host-to-NIC datapath. Packets that do not fill the Maximum Transmission Unit (MTU) are common in any host system: short RPC replies, health checks, protocol control messages (TCP ACKs, DNS, etc.), and others. These packets represent unused capacity that can carry telemetry data at no additional transport cost.

We observe that sub-MTU packets present a dual opportunity. Beyond serving as a vehicle for telemetry transport, their transmission events naturally trigger kernel execution, providing an occasion to read system metrics inline without additional system calls. By attaching instrumentation to the transmit path, the kernel can read system metrics and embed them into packets before they traverse the PCIe bus; a SmartNIC then strips and aggregates this metadata for downstream analysis. This approach has the potential to amortize the cost of observability across existing I/O operations, avoiding extra PCIe transactions, and reducing the need for dedicated telemetry packets. We envision Extended Berkeley Packet Filter (eBPF) programs as key enabling technology to implement this opportunistic telemetry transport, given their ability to execute safely within the kernel and attach to network events. Our approach shares principles with InvisiFlow [6], but focuses on the host-to-NIC datapath and the unique challenges of high-frequency system metrics observability.

In this work-in-progress paper, we sketch the architecture of opportunistic telemetry, outline key design choices and identify key challenges toward realizing this vision. Additionally, we present an initial evaluation to quantify the impact of monitoring frequency on application latencies in containerized microservice workloads. We conduct a controlled experimental study comparing eBPF-based CPU utilization monitoring with traditional `/proc`-based polling. We evaluate the CPU overhead introduced by each monitoring approach and quantify its impact on application flow completion time (FCT) across different sampling frequencies. Our preliminary results show that sending high-frequency telemetry can introduce significant contention, degrading FCT or leading to RPC errors in the worst case. These findings motivate the need for more efficient telemetry delivery mechanisms, such as kernel-level packet reuse and telemetry piggybacking, which are the focus of our ongoing work.

II. Background and Motivation

We first describe the anatomy of metrics collection (§ II-A, § II-B), then survey current metrics observability architectures, from traditional centralized pipelines to hardware-accelerated designs (§ II-C), and finally discuss the limitations that are still common to both paradigms (§ II-D).

A. Observability and Metrics

In cloud-native environments, where applications are decomposed into hundreds or thousands of loosely-coupled microservices running across containerized infrastructure, observability is essential for detecting failures, diagnosing performance issues, and enforcing service-level objectives (SLOs).

Metrics in cloud-native services. Metrics are quantitative measurements of system or application behavior sampled over time. They fall into two broad categories based on their source and update frequency:

- *System-level metrics* expose resource consumption at the operating system layer: CPU utilization, memory usage, disk I/O throughput, and network activity. These metrics are typically maintained by the kernel and updated at configurable intervals (e.g., every scheduling tick or on-demand via `/proc` reads).
- *Application-level metrics* capture service-specific state: request counts, queue depths, cache hit ratios, or database operation latencies. Unlike system metrics, application metrics are often updated synchronously with request processing, e.g., a counter increments on every RPC.

Metrics collection involves two related but distinct steps: *accessing* metric values from memory (§ II-B), and *transferring* them to downstream processing components (§ II-C).

B. Accessing Metrics

Application metrics are typically created and updated in-process using instrumentation libraries (e.g., offering optimized structures for counters, gauges, and histograms). Instead, system-level metrics are maintained in the kernel and exposed to user-space.x For example, the `/proc` filesystem provides per-process statistics (e.g., `/proc/[pid]/stat` for CPU time and scheduling information), while `/sys/fs/cgroup` exposes container-level resource accounting for memory, CPU, and I/O.

Unfortunately, accessing kernel-maintained metrics from user-space involves non-negligible overheads, due to system calls, user-kernel transitions, and text parsing [3], [7]. This cost is incurred despite the kernel already updates the underlying data structures regularly as part of its operations, leaving performance on the table for more efficient access methods.

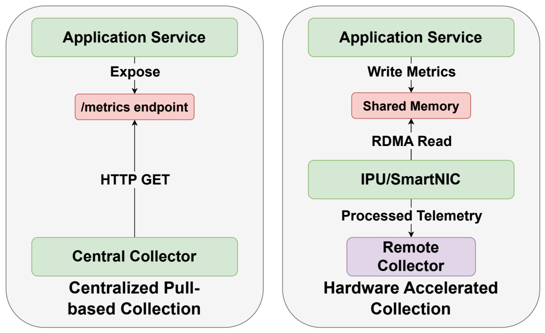

Fig. 1. Metrics collection architectures.

C. Transferring Metrics

The second step, data transferring, moves telemetry from the source to a location where they can be stored, queried, or analyzed. This transport step adds further overhead that varies with the collection architecture. We now survey two state-of-the-art paradigms for metrics collection, illustrated in Figure 1.

1) Centralized Collection Pipelines: the dominant approach to metrics observability relies on *pull-based* centralized collection. Systems such as Prometheus [8], Datadog, and OpenTelemetry collectors periodically scrape metrics from instrumented services over HTTP. Each service exposes an HTTP endpoint; the collector issues HTTP requests at a configured interval and parses the text-formatted response. This architecture imposes significant costs: each scrape triggers request handling, metrics serialization (often to Prometheus exposition format), and response transmission. For services with thousands of metrics, serialization alone can consume measurable CPU cycles. To limit these overheads, operators configure coarse scrape intervals[1]. This trades temporal resolution for resource efficiency, and transient spikes that last only seconds may go unobserved, e.g., metastable failure triggers [9], [10].

2) Hardware-Accelerated Collection: an alternative approach [3] offloads metrics collection and processing to dedicated hardware accelerators, such as Infrastructure Processing Units (IPUs) or SmartNICs [11]–[13]. IPUs are programmable network interface devices equipped with general-purpose cores (e.g., ARM Cortex-A) capable of running custom software independently of the host CPU. In this model, the IPU acts as an *observability sidecar*: it continuously reads metrics from host memory, performs in-situ analysis (filtering, aggregation, anomaly detection), and exports summarized results to external collectors. By moving telemetry processing off the host CPU, IPU-based designs free application cores for tenant workloads while enabling higher collection frequencies than centralized pull-based systems. However, hardware acceleration does not eliminate data movement costs within the host network. The IPU must still fetch metrics from host memory, and this data movement introduces its own overhead, as we discuss next.

D. Overheads and Contentions for Telemetry Transfers

Both centralized and hardware-accelerated collection architectures face a common challenge: transferring metrics data

[1]30 seconds or longer is common in production [3], [7]

from where it is generated (the host) to where it is processed (being the collector or an IPU). This data transfer step [2], often overlooked, can become the dominant source of overhead at high collection frequencies.

CPU overhead in centralized pipelines. In traditional pull-based systems, metrics transport occurs over the network stack. Each scrape involves:

1) *System calls*: the service handles an incoming HTTP request, requiring socket operations (`accept`, `read`, `write`) that transition between user and kernel space.
2) *Serialization*: metrics must be formatted into the exposition format, e.g., Prometheus.
3) *Network I/O*: the serialized response traverses the network stack and reaches the NIC, or, for stack-bypass technologies, directly reaches the NIC, e.g., via PCIe.

These overheads scale with both the number of metrics and the scrape frequency. Consequently, production systems accept coarse sampling intervals as the price of acceptable overhead. Unfortunately, this limits observability resolution and delays anomaly detection.

PCIe bottleneck in hardware-accelerated pipelines. Hardware-accelerated designs such as μView [3] bypass the network stack by using RDMA for host-to-IPU data transfer. RDMA eliminates CPU involvement in the data path: the IPU's network adapter issues read requests directly to host memory over the PCIe bus, and responses flow back without host CPU interrupts or memory copies. This approach dramatically reduces per-transfer CPU overhead. However, it introduces a different bottleneck: PCIe bus contention. The PCIe interconnect is a shared resource; high-frequency RDMA reads compete with other PCIe traffic (NVMe storage, GPU communication, network I/O) for bandwidth and, more importantly, for transaction slots. Consequently, frequent RDMA reads can occupy the PCIe bus, leading to queuing delays that increase end-to-end latency for all PCIe traffic. In cloud environments, where metrics collection traffic competes with tenant's network I/O, aggressive collection can degrade application performance, even though it consumes no host CPU cycles.

E. Summary

Collecting metrics at high frequency is essential for observability in cloud-native systems, but it incurs non-trivial overheads. Metrics collection involves two steps: *accessing* metric values from memory, and *transferring* them to downstream processing components. Both steps introduce overheads that limit achievable sampling granularity. First, reading metrics from kernel-maintained data structures requires system calls and context switches when performed from user space. Second, transporting metrics to collectors or accelerators introduces additional communication costs. While RDMA-based collection saves CPU overheads, it might shift the contention hotspot to PCIe. Fundamentally, neither approach to metrics transport is immune to contentions with user workloads.

[2]we use the terminology transfer or transport interchangeably

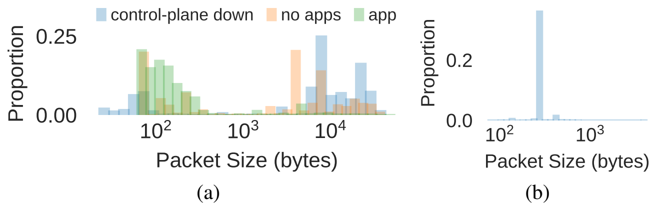

Fig. 2. Packet size distribution across (a) Three scenarios on a `k8s` cluster; when app is deployed, we generate a request load of 100 RPS (b) An idle server in our testbed, not part of the `k8s` cluster.

These observations motivate the idea of this work: rather than optimizing the transport mechanism itself, we ask whether metrics collection can be *amortized* across network communication regularly occurring in end-hosts.

III. Opportunistic Telemetry

A. Overview

We elaborate on a simple yet still unexplored opportunity: leveraging existing I/O events generated by regular host traffic to both *access* and *transfer* observability metrics, thus amortizing the telemetry overheads. We term this approach opportunistic telemetry, synthesized in Fig. 3.

Observation 1: packet transmission already triggers kernel execution. When a host transmits a packet, the network stack processes the outgoing data, and the driver schedules DMA transfers. During this process, the CPU executes in kernel context. This execution context, whether in software interrupt context (softirq) or process context, provides an opportunity to perform additional lightweight work without incurring the overhead of a separate system call or context switch.

Observation 2: outgoing packets smaller than MTU. Further, we observe that many outgoing packets in cloud systems are smaller than MTU, generated by applications, orchestration layers, and network protocols. Common sources of sub-MTU packets in cloud-native systems include: (i) *RPC responses* carrying compact payloads such as status codes or small JSON objects; (ii) *Health checks* producing minimally sized HTTP responses; (iii) *Service-mesh control messages* exchanged by sidecars (e.g., Envoy); (iv) *Database responses* from cache hits (Redis, Memcached); (v) *Control packets* such as TCP ACKs, ICMP messages, DNS queries and responses, ARP requests, etc. These packets traverse the host-to-SmartNIC datapath regularly, and their transmission events represent opportunities to opportunistically take advantage of the kernel execution context and perform telemetry collection operations.

To make our observation concrete, we conducted a preliminary measurement campaign in a self-hosted three-node Kubernetes cluster. We measured the packet size distributions under an idle-cluster scenario—i.e., when no microservice are running—and under a busy-cluster scenario, in which we deployed an 11-microservice application [14]. These measurements were taken from a TC hook point on the transmit path of a Kubernetes worker node, where we expect to see the most sub-MTU packets generated by application responses and control messages. We also studied a third scenario where we ablated control plane traffic by revoking TLS certificates for control plane components (API server, controller manager, scheduler), while keeping the application deployed. This scenario limits control-plane messaging, e.g., scheduler binding requests, leader election heartbeats, etc., which typically generates small HTTPS/gRPC messages below MTU. Fig. 2a shows that the proportion of sub-MTU packets increases progressively: with the control plane disabled, sub-MTU packets represent a small fraction of traffic (37.22%); when the control plane is enabled, the proportion of sub-MTU packets increases slightly (37.99%); and with microservices deployed, the proportion increases further with 93.44% of the packets being sub-MTU. The average traffic rates are 173 KB/s, 226 KB/s, and 579 KB/s, respectively, across the three scenarios. For completeness, we also measured the packet size distribution on an idle server in our testbed, not part of the Kubernetes cluster. This server only exchanges management and monitoring traffic, with an average traffic rate of 120 KB/s. As expected under these conditions, we observe that the majority of the packets are sub-MTU, as shown in Fig. 2b.

Opportunity 1: *inline* metrics reading. Because sub-MTU packet transmission already executes in kernel context, we can read local system metrics at that moment without additional system calls. An eBPF program attached to the transmit path can invoke helpers such as `bpf_perf_event_read()` to sample CPU counters, cache statistics, scheduler state, or other kernel-maintained metrics inline. While previous work has explored eBPF-based telemetry collection [3], [15], none has systematically leveraged packet transmission events as triggers to read system-level metrics.

Opportunity 2: telemetry *piggybacking*. Sub-MTU packets contain unused capacity—*slack*—that can carry telemetry data. These packets represent "free" transmission opportunities that traverse the host-to-NIC datapath. They already incur fixed DMA and PCIe transaction overheads independent of payload size (descriptor setup, TLP headers, doorbell notifications); embedding compact telemetry metadata into this slack avoids generating separate telemetry packets. At small payload sizes, the marginal cost of extending an existing DMA transfer is smaller than the fixed overhead of initiating a separate PCIe transaction for dedicated telemetry.

Together, inline metrics reading and telemetry piggybacking constitute *opportunistic telemetry*. Opportunistic telemetry addresses the transport overhead identified in § II-D: it mitigates the need of dedicated telemetry packets, avoids extra system calls and context switches, and requires no additional PCIe transactions beyond those already scheduled for application traffic.

B. Proposed Design

An eBPF-based approach. We envision the design to operate as follows. On the host, an eBPF program intercepts outgoing packets at the traffic control (TC) layer. For packets with sufficient slack, the program reads relevant metrics inline and appends telemetry metadata to the packet buffer. The augmented packet crosses the PCIe bus as a single DMA transfer, requiring no additional transactions. On the SmartNIC, a parser identifies piggybacked telemetry, strips it from the application payload, and forwards each component to its destination: the original packet continues to the network, while telemetry flows to the accelerator's processing pipeline or an external collector.

Host-side components. eBPF provides the mechanism to implement opportunistic telemetry entirely within the kernel. eBPF programs can attach to various kernel hook points, enabling both metrics reading and packet augmentation without user-space involvement.

For metrics reading, eBPF programs can access kernel-maintained data structures directly. Programs attached to scheduler tracepoints or network hooks can read per-CPU counters, hardware performance counters, and memory statistics. Unlike `/proc`-based approaches, these accesses require no file parsing, system calls, or context switches. Metrics can be accumulated in eBPF maps for efficient retrieval during packet processing.

For packet interception (necessary for piggybacking), eBPF programs attach to the Traffic Control (TC) layer on the egress path. The TC hook executes synchronously during packet transmission, before the packet crosses the PCIe bus to the NIC. At this point, the program inspects packet length, identifies sub-MTU candidates, and appends telemetry by extending the packet buffer and writing metadata to the tail. Piggybacked telemetry must be encoded such that the SmartNIC can identify and extract it without modifying application semantics. We adopt a trailer-based encoding: telemetry is appended after the original payload, followed by a fixed-size length marker.

Metrics reading and piggybacking can execute synchronously, or the TC hook can retrieve pre-cached values from eBPF maps populated by other tracepoints (§ III-C).

SmartNIC-side extraction. Modern SmartNICs provide the programmability required for telemetry extraction. Upon receiving a packet, the SmartNIC inspects the trailer for a valid length marker. If present, it extracts the telemetry payload, truncates the packet to restore the original application data, and forwards each component independently: the application packet continues to the network egress, while telemetry is routed to onboard processing cores or an external collector. This separation occurs entirely on the SmartNIC without host CPU involvement. If telemetry must be forwarded to an external collector, the SmartNIC also implements communication proxies compatible with collector's protocols.

C. Challenges

While conceptually simple, realizing the vision of opportunistic telemetry piggybacking raises several challenges.

Traffic idleness. Piggybacking on sub-MTU packets inherently depends on traffic availability: during low or idle traffic periods, collection frequency drops as a result of the scarcity of piggybacking opportunities. During these periods,

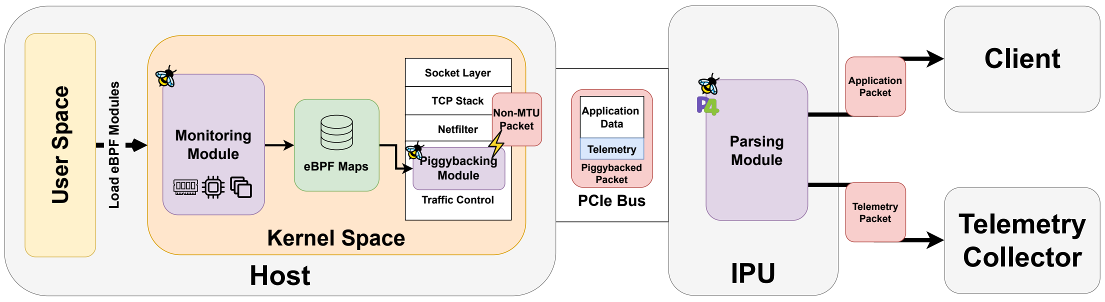

Fig. 3. Proposed Opportunistic Telemetry Design

opportunistic telemetry may fail to provide the desired visibility into system state. In addition, not all sub-MTU packets have sufficient slack for telemetry. Packets near MTU size cannot accommodate additional data without fragmentation. Fundamentally, opportunistic telemetry's collection rate depends on the distribution of sub-MTU packets and available piggybacking opportunities.

To tackle this challenge, a dedicated transport protocol is required, coordinating between the host eBPF program and the SmartNIC parser. We envision the transport protocol to balance opportunistic piggybacking with a periodic heartbeat: piggybacking maximizes efficiency during active phases, while a heartbeat ensures baseline visibility when traffic is idle. Further, the protocol must handle fragmentation gracefully: if a packet carrying telemetry fragments, the SmartNIC must reassemble it before extraction.

Reliability & failure handling. Piggybacking telemetry onto regular traffic introduces loss, reordering, and fragmentation challenges. To cope with losses, the system can embed sequence numbers or lightweight checksums in the telemetry payload, enabling the collector to identify gaps. When losses exceed acceptable thresholds, a control-plane mechanism can trigger fallback to a user-space exporter that transmits critical metrics via dedicated packets, trading the efficiency benefits of piggybacking for delivery guarantees. Operators can also configure redundancy by duplicating high-priority metrics across multiple carrier packets, increasing the probability that at least one copy arrives.

Integration with existing eBPF exporters. We are not the first to propose reading system-level metrics via eBPF. Existing eBPF-based exporters already collect a variety of system metrics at various hooks in the kernel [5], [15], [16]. While opportunistic telemetry can read kernel-level metrics *synchronously* with packet transmission, it must also integrate with telemetry read by existing exporters asynchronously. To this end, we propose a shared eBPF map to store pre-cached metrics collected by other eBPF programs attached to different kernel hooks. The transport protocol should then coordinate retrieval of cached metrics and encapsulate them within piggybacking opportunities. This flexibility allows operators to decouple sampling rate from export rate: metrics can be read at high frequency and cached, then exported opportunistically when packets become available.

IV. Work-in-Progress Experiments

We developed a telemetry collection microbenchmark to quantify the overheads discussed in § II-D. Our preliminary evaluation addresses the following questions:

- **Q1:** What CPU overhead do different telemetry collection mechanisms impose, and how does this overhead scale with sampling frequency?
- **Q2:** How does high-frequency telemetry collection affect application-level performance, measured as request FCT?

A. Experimental setup

Our setup consists of three main components: a load generator, an application server, and a telemetry collector. All components are deployed and orchestrated in a three-nodes Kubernetes cluster. We generate the request workload using Locust [17]. The locust generator simulates a load of 575 clients with a wait time of 0.0001s. We selected these values empirically such that we saturate the server capacity without causing failures due to high load. The application server runs an Apache Web server, hosting a simple PHP frontend page. The monitoring module collects telemetry using either *event-driven kernel-level* mechanisms or *periodic user-space polling*. For both approaches, we monitor the metric `process_CPU_utilization` and, depending on the experiment configuration, send samples to the remote collector via UDP at configurable frequencies.

Implementation highlights. In the event-driven mode, we capture telemetry directly during kernel events, by attaching an eBPF program to the `sched:sched_switch` tracepoint, which fires on every context switch. We maintain two BPF hash maps: one recording the timestamp at which a process becomes runnable, and another accumulating the total CPU time per process. At each context switch, we compute the runtime of the outgoing process using kernel timestamps obtained via `bpf_ktime_get_ns()`, and we update the start time of the incoming process accordingly. We later compute CPU utilization in user space as the ratio of accumulated CPU time to elapsed wall-clock time.

The second monitoring approach relies on periodic polling of the `/proc/<pid>/stat` interface. For each target process, the monitoring script extracts user-mode and kernel-mode CPU times (`utime` and `stime`), converts them from clock ticks to seconds using `SC_CLK_TCK`, and computes utilization based on successive deltas. This approach incurs the overheads discussed in § II-D, including system calls and context switches.

B. Results & Discussion

*1) CPU Utilization of Monitoring Tools (**Q1**):* We first evaluated the CPU overhead of the monitoring tools themselves (**Q1**) by running each tool on its own for 30 seconds at different sampling frequencies.

Figure 4a shows the measured CPU utilization for both eBPF-based and `/proc`-based monitoring. As expected, utilization increases with higher sampling frequencies. At low frequencies (1 s), both methods impose minimal overhead ($< 0.05\%$ CPU). At high frequencies, eBPF consistently consumes less CPU than the `/proc`-based method. For example, at 1 ms frequency, eBPF uses approximately 3% CPU while `/proc` consumes around 8.8%. The gap widens further at extremely high frequencies, with eBPF consuming 64% less CPU than `/proc` at $10\mu s$. These results demonstrate the computational efficiency of eBPF in high-resolution monitoring scenarios compared to `/proc`-based method.

*2) Impact on Request Latency (**Q2**):* To understand how monitoring impacts application performance (**Q2**), we measured the average FCT of requests issued by the load generator. For each configuration, we allowed the system to reach a steady state over 1200 s, and we run monitoring for 5000 s while recording the FCT.

We first compared `/proc`-based monitoring when telemetry is either sent to a UDP collector (*sending*) or kept local (*no sending*). Figure 4b shows the average FCT for each sampling frequency. Cases with flow failures are not plotted, but total failures are reported. In the *no sending* configuration, average FCT linearly increases across frequencies, ranging from 12,836 ms to 15,471 ms at 1 s and $10\mu s$ respectively, with no failures observed. At high frequencies, when sending telemetry, significant flow failures occur due to increased contention. For instance, at $10\mu s$ there were 965,761 total failed flows. However, at lower frequencies (1 ms and above), no failures occurred, and average FCT values are comparable to the no-sending scenario.

These results indicate that high-frequency telemetry transmission can significantly impact application performance due to network contention and the increasing trend in both cases indicate CPU contention's effect on FCT.

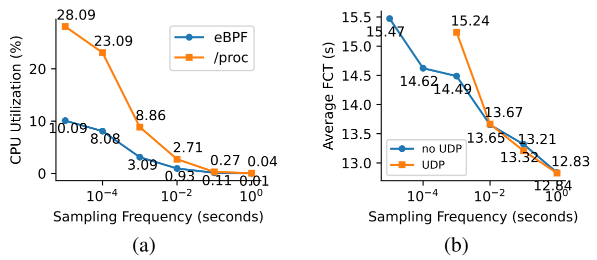

Fig. 4. Monitoring overhead comparison. (a) eBPF vs `/proc` CPU overhead. (b) `/proc` monitoring impact on FCT; cases with failures (below 1ms) are omitted.

V. Conclusion

We introduced opportunistic telemetry, a design that piggybacks metrics onto sub-MTU packets using eBPF, amortizing collection overhead across existing I/O operations.

Our current design and evaluation focus on microservice-based cloud systems, nevertheless the concept of opportunistic telemetry extends beyond this setting. An interesting direction is to explore its applicability to LLM training and inference workloads, where fine-grained observability of GPU utilization, memory bandwidth, and inter-GPU communication is critical for performance optimization but costly to collect at high frequency. Similarly, extending our framework to incorporate GPU-level metrics as telemetry sources, rather than only CPU and system-level counters, could enable unified observability across heterogeneous accelerator deployments. These directions present additional challenges, such as coordinating telemetry collection across PCIe and NVLink fabrics, and exploiting piggybacking opportunities in collective communication patterns, which we leave for future work.

References

[1] Y. Zheng, Y. Hu, T. Yu, and A. Quinn, "AgentSight: System-level observability for AI agents using eBPF," in *Workshop on Practical Adoption Challenges of ML for Systems*, 2025.

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

[3] A. Cornacchia, T. A. Benson, M. Bilal, and M. Canini, "Observability is eating your cores: Fine-grained analysis of microservice metrics with IPU-hosted sketches," in *USENIX NSDI*, 2026.

[4] J. Lin, A. Cardoza, T. Khan, Y. Ro, B. E. Stephens, H. Wassel, and A. Akella, "RingLeader: Efficiently offloading Intra-Server orchestration to NICs," in *USENIX NSDI*, 2023.

[5] I. Systems, "Rezolus: High-resolution systems performance telemetry," https://github.com/iopsystems/rezolus.

[6] Y. Zhang, L. Yu, G. Antichi, R. B. Basat, and V. Liu, "Enabling silent telemetry data transmission with InvisiFlow," in *USENIX NSDI*, 2025.

[7] Z. Wang, T. Ma, L. Kong, Z. Wen, J. Li, Z. Song, Y. Lu, G. Chen, and W. Cao, "ZERO overhead monitoring for cloud-native infrastructure using RDMA," in *USENIX ATC*, 2022.

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

[9] N. Bronson, A. Aghayev, A. Charapko, and T. Zhu, "Metastable failures in distributed systems," in *ACM HotOS*, 2021.

[10] L. Huang, M. Magnusson, A. B. Muralikrishna, S. Estyak, R. Isaacs, A. Aghayev, T. Zhu, and A. Charapko, "Metastable failures in the wild," in *USENIX OSDI 22*, 2022.

[11] "NVIDIA BlueField networking platform," https://www.nvidia.com/en-us/networking/products/data-processing-unit/, 2023.

[12] "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.

[13] "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.

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

[15] eBPF Foundation, "What is eBPF?" accessed: 2025-12-07. [Online]. Available: https://ebpf.io/what-is-ebpf/

[16] I. Babrou, "Introducing ebpf_exporter," accessed: 2026-01-11. [Online]. Available: https://blog.cloudflare.com/introducing-ebpf_exporter/

[17] Locust Project, "Locust: Scalable load testing framework," accessed: 2025-12-07. [Online]. Available: https://locust.io