cnFlow: An eBPF-Driven Telemetry Framework for Cloud-Native Networks

Jisu Kim
Department of AI-Based Convergence
Dankook University
imjs0807@dankook.ac.kr

Jaehyun Nam*
Department of Computer Engineering
Dankook University
namjh@dankook.ac.kr

***Abstract*—Cloud-native platforms such as Kubernetes have transformed modern networking by decomposing applications into thousands of microservices communicating over heterogeneous protocols and dynamic multi-interface topologies. However, existing telemetry tools fall short in this environment: xFlow-style solutions omit application and identity context, while sidecar-based service meshes introduce high overhead and cannot observe hostNetwork or SR-IOV traffic. We present *cnFlow*, a sidecar-free, eBPF-based telemetry framework that provides low-overhead, context-rich observability for Kubernetes clusters. *cnFlow* attaches in-kernel eBPF programs to Traffic Control ingress/egress hooks on all pod and host interfaces, enabling real-time packet classification and export via lock-free ring buffers. In user space, it reconstructs higher-layer transactions, which are then enriched with Kubernetes metadata (e.g., namespace, service account, security context) and geolocation. Evaluations on a multi-protocol testbed show that *cnFlow* captures L3–L7 traffic (e.g., HTTP, Redis, DNS, ICMP) with 2% CPU overhead and sub-millisecond latency, outperforming Istio and Kubeshark in both efficiency and observability coverage.**

***Index Terms*—Cloud-Native Network, Telemetry, eBPF**

I. Introduction

Cloud-native platforms such as Kubernetes have reshaped modern computing infrastructure. By decomposing monolithic applications into fine-grained microservices, organizations can deploy, scale, and manage software with improved agility. These microservices interact across dynamically instantiated pods and virtual interfaces, often coordinated through complex overlay networks. According to a 2025 survey by the Linux Foundation Networking, more than 73% of organizations have migrated legacy workloads to cloud-native platforms, and the cloud-network telemetry market is projected to grow by 15.6% annually through 2035 [1]. This transition has created a critical need for observability tools capable of addressing the operational complexity of cloud-native environments.

Kubernetes, the de facto standard for container orchestration [2], supports dynamic scheduling and scalable microservice deployment. It enables advanced networking configurations through technologies such as Multus [3] and SR-IOV [4]. While these features offer performance benefits, they introduce challenges for network observability. Operators must analyze traffic patterns among ephemeral workloads that span namespaces, security domains, and heterogeneous interfaces. Observability systems must provide semantic context: the initiating identity, the associated service account and privilege level, the encryption status of communications, and the correctness of application-layer protocol behavior.

Most existing telemetry frameworks do not meet these requirements. Device-centric solutions such as NetFlow, sFlow, and IPFIX [5] sample traffic at the network edge and operate at the network and transport layers. These tools lack visibility into application-layer semantics and cannot attribute flows to containerized workloads. Service meshes such as Istio [6] and Linkerd [7] provide finer-grained insights by injecting sidecar proxies that inspect L7 traffic. However, they introduce considerable performance overhead [8], increase deployment complexity, and cannot observe hostNetwork or SR-IOV traffic. Their functions also duplicate capabilities that could be implemented more efficiently in lower system layers.

Recent approaches employ extended Berkeley Packet Filter (eBPF) [9], a safe, programmable in-kernel environment, to enable lightweight observability. Tools such as Pixie and Cilium Hubble use eBPF to monitor events and traffic from the kernel. Many rely on statically defined probes, which may miss certain execution paths. They typically lack support for stateful, protocol-aware session reconstruction and are not tightly integrated with Kubernetes metadata such as service accounts, namespaces, or node location. These constraints limit their utility in multi-tenant or regulated environments that demand end-to-end contextual visibility.

To address the limitations of existing telemetry approaches, we present ***cnFlow***, a sidecar-free telemetry framework based on eBPF, designed for production-grade Kubernetes environments. *cnFlow* attaches eBPF programs to Traffic Control (TC) ingress and egress hooks on all pod and host interfaces. This enables comprehensive visibility into L2 through L4 traffic, including communication generated by hostNetwork pods, overlay interfaces, and SR-IOV virtual functions. The framework operates entirely within the kernel without requiring kernel modifications or changes to application logic.

cnFlow consists of three core components: the kernel-side inspection engine, the user-space correlation engine, and the centralized enrichment manager. The kernel-side engine attaches eBPF programs to Traffic Control ingress and egress hooks on all pod and host interfaces to inspect packets in real time. It performs protocol classification using signature-

* Jaehyun Nam (namjh@dankook.ac.kr) is the corresponding author.

based heuristics and exports flow-level metadata to user space through high-throughput, lock-free ring buffers. The user-space engine receives these flow records and reconstructs application-layer sessions by maintaining connection state, tracking TCP sequence numbers, and performing stream reassembly. This enables precise extraction of transactions for protocols such as HTTP, Redis, and DNS. The centralized enrichment manager aggregates flow data from all nodes and augments it with control-plane metadata obtained from the Kubernetes API, including pod identity, namespace, service account, security context, and deployment configuration. When available, node-level geolocation is also integrated. This component produces enriched, hierarchical telemetry that supports detailed performance monitoring and policy compliance analysis across the cluster.

We evaluate *cnFlow* in a Kubernetes-based testbed with representative microservices and widely used protocols such as HTTP/1.1, Redis, DNS, and ICMP. Compared to Istio and Kubeshark, *cnFlow* offers broader or equivalent protocol coverage with significantly lower resource overhead. It introduces sub-millisecond latency and consistently maintains CPU usage below 2 percent per node, demonstrating a practical balance between observability depth and operational efficiency.

Contributions. This paper makes the following contributions:

- Identification of key shortcomings in Kubernetes telemetry, including limited protocol coverage, insufficient context attribution, and excessive runtime overhead.
- Design and implementation of *cnFlow*, a sidecar-free telemetry framework that combines eBPF-based packet inspection, protocol-aware session reconstruction, and control-plane metadata integration.
- Experimental evaluation of *cnFlow* on a Kubernetes testbed, demonstrating precise L3–L7 flow capture and low-overhead, context-rich observability across pods, namespaces, and clusters.

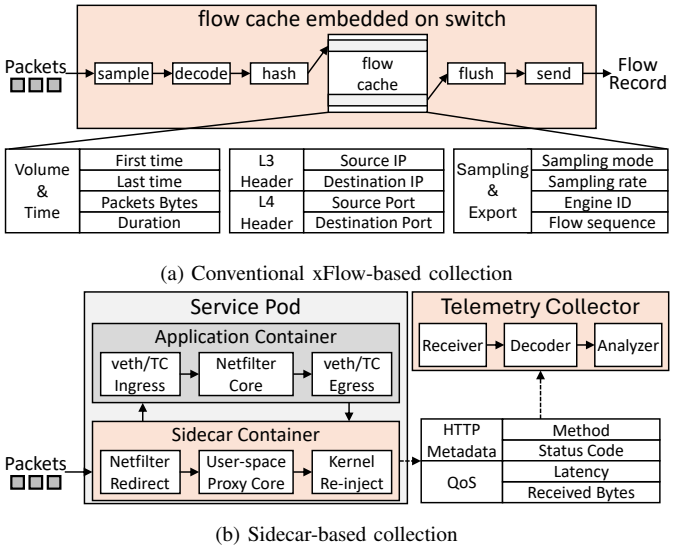
Paper Organization. Section II reviews foundational concepts and representative prior work. Section III details the system architecture and implementation of *cnFlow*. Section IV presents our experimental methodology and results. Section V discusses related and complementary research. Section VI concludes the paper with final remarks.

II. Background and Motivation

This section outlines the limitations of existing network-telemetry solutions in cloud-native environments, introduces eBPF as a foundation for efficient in-kernel monitoring, and motivates the need for a new framework that addresses current visibility and scalability challenges.

A. Cloud-Native Telemetry for Network Monitoring

Modern cloud-native applications consist of loosely coupled microservices that communicate through diverse protocols and dynamic API interactions [10]. Achieving visibility into these interactions, for performance monitoring, policy enforcement, and security diagnostics, requires accurate, real-time telemetry across the cluster. However, the decentralized and ephemeral nature of microservice communication complicates traditional monitoring approaches, which often lack sufficient granularity and context to trace interactions across service boundaries.



(a) Conventional xFlow-based collection

(b) Sidecar-based collection

Fig. 1: Conventional network-traffic collection methods: (a) xFlow family (NetFlow, sFlow, IPFIX) and (b) service-mesh sidecar approach.

Two primary approaches dominate existing telemetry systems. The first is device-centric monitoring (Fig. 1(a)), exemplified by protocols such as sFlow [11], NetFlow [12], and IPFIX [13]. These tools operate at the network edge, extracting flow-level data from switches and routers using L3/L4 fields such as IP addresses, ports, and transport protocols [14]. While they provide scalable, infrastructure-wide visibility, they lack awareness of application-layer behavior and workload context [15]. Furthermore, sampling limits temporal accuracy and prevents session-level flow reconstruction.

The second approach is sidecar-based monitoring (Fig.1(b)), adopted by service mesh frameworks including Istio [6] and Linkerd [8]. By injecting Envoy proxies [16] alongside each pod, these systems can intercept and analyze L7 traffic, generating detailed service graphs and application-layer performance metrics. However, this design introduces non-trivial resource overhead due to additional containers, complicates deployment workflows and maintenance, and fails to capture traffic from hostNetwork pods or from auxiliary interfaces created by CNI extensions such as Multus or SR-IOV [17].

B. Extended Berkeley Packet Filter (eBPF)

To address the overhead and limitations of user-space-based telemetry, recent systems have adopted the Extended Berkeley Packet Filter (eBPF), an in-kernel virtual machine that enables safe execution of user-defined programs at runtime [17]. eBPF programs can be attached dynamically to a variety of kernel hooks, such as system calls, tracepoints, or Traffic Control (TC) ingress/egress paths. This flexibility enables event-driven telemetry with minimal impact on application performance.

By executing logic directly in kernel space and transferring event data to user space via high-throughput ring buffers, eBPF supports low-latency, high-frequency monitoring. These properties make it well suited to modern containerized systems that require scalable observability across ephemeral workloads, high-density nodes, and dynamic topologies [18].

C. Challenges in Cluster-Wide Telemetry and Observability

Despite recent advances, cloud-native observability systems have evolved. However, they still face several unresolved challenges in achieving comprehensive, low-overhead visibility.

First, reliance on sidecar proxies incurs non-trivial operational complexity and computational overhead. In scenarios involving hostNetwork pods or interfaces provisioned through Multus and SR-IOV, sidecars are bypassed entirely, resulting in fragmented visibility and critical blind spots [8]. A telemetry system must instead capture network activity at the kernel level without requiring application modifications or sidecar deployment. Without such coverage, security-critical or performance-sensitive traffic may go completely unobserved, undermining the effectiveness of the monitoring infrastructure.

Second, Kubernetes clusters exhibit heterogeneous communication patterns, involving protocols such as HTTP, DNS, gRPC, Redis, and ICMP [19]. Extracting meaningful protocol-specific fields (e.g., HTTP methods or DNS query types) and aggregating quality-of-service metrics (e.g., latency, jitter, packet loss) in real time is essential for effective performance diagnosis and SLA validation [20]. However, existing tools often rely on fixed sampling or partial instrumentation, limiting their effectiveness. In the absence of full protocol coverage and accurate session reconstruction, subtle performance regressions and application misbehaviors may go undetected.

Third, the growing adoption of Kubernetes in regulated industries has elevated the importance of security and auditability. Network telemetry must be able to associate each flow with Kubernetes-native context such as ServiceAccount, SecurityContext, deployment, and node-level geolocation [5]. This level of context is critical for enforcing access policies, performing forensic analysis, and enabling real-time threat detection. Yet most telemetry tools fail to integrate deeply with the Kubernetes control plane, preventing administrators from gaining comprehensive and trustworthy insights. Without contextual attribution, even accurate flow metrics provide limited operational or security value.

These limitations motivate the need for a unified framework that provides protocol-aware, context-rich telemetry directly from within the kernel, while scaling to production-grade Kubernetes environments without incurring significant overhead.

III. *cnFlow* Design

This section presents the architecture and operational workflow of *cnFlow*, which is designed to address the limitations of network observability in cloud-native environments by combining kernel-level traffic inspection with context-rich, hierarchical telemetry analysis.

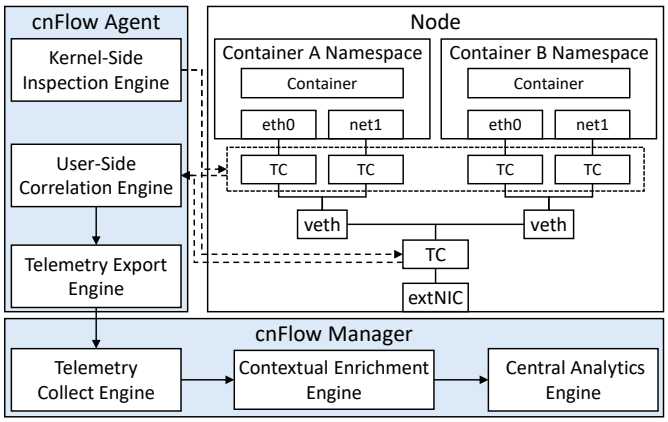


Fig. 2: The overall architecture of *cnFlow*.

A. System Overview

As illustrated in Fig. 2, cnFlow consists of two major components: the agent and the manager. The agent is deployed as a DaemonSet on each Kubernetes node and comprises three subsystems: the kernel-side inspection Engine, the user-side correlation engine, and the telemetry export engine. The kernel engine attaches eBPF programs to ingress and egress hooks in the Linux Traffic Control (TC) subsystem, enabling the real-time capture and initial parsing of network packets directly at the kernel level. Parsed metadata is streamed through high-performance ring buffers to the user space, where the correlation engine reconstructs session-level flows, maintains connection state, and decodes application-layer protocols. This engine supports detailed analysis of protocols such as HTTP, DNS, and Redis, and derives QoS metrics including latency, throughput, jitter, and packet loss. The export engine then transmits the enriched telemetry to the manager.

The manager consists of the telemetry collect Engine, the contextual enrichment engine, and the central analytics engine. It receives telemetry data from all agents and combines it with Kubernetes-native metadata, such as pod name, namespace, ServiceAccount, SecurityContext, and deployment attributes, as well as node-level geolocation. The contextual enrichment engine augments each flow record with this information, while the central analytics engine performs cluster-wide correlation and analysis. This integrated processing enables operators to observe system-wide service interactions, identify anomalous traffic behavior, and assess policy compliance effectively.

B. Network Telemetry in Traffic Control

To enable comprehensive yet efficient traffic inspection, *cnFlow* attaches lightweight, event-driven eBPF programs to the ingress and egress queueing discipline (qdisc) of each node's network interfaces using the Linux Traffic Control (TC) subsystem (Fig. 3). This approach reliably captures both container and host traffic directly in kernel space, avoiding user-space interception and unnecessary packet duplication. It significantly minimizes context switches and consistently maintains stable system performance under high traffic loads.

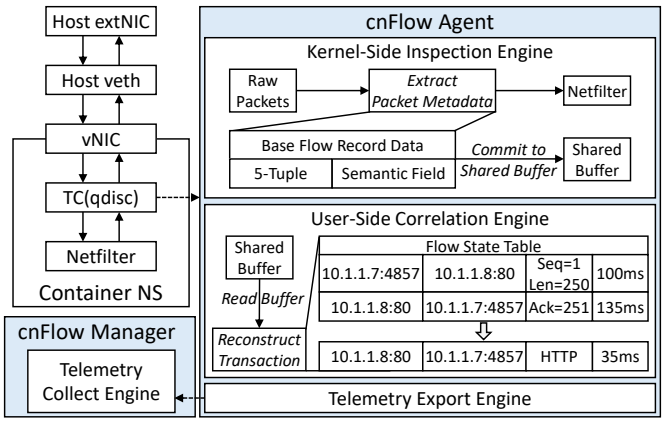


Fig. 3: Packet interception via eBPF at TC ingress/egress layer.

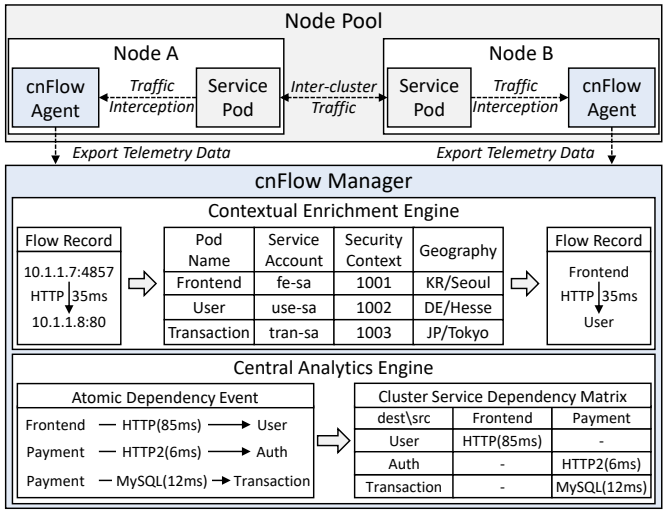


Fig. 4: Hierarchical telemetry synthesis for cluster-wide insight and contextual network performance diagnostics.

Protocol identification is performed within the kernel using signature-based classification instead of static port numbers, which are unreliable in dynamic microservice environments. Each packet is scanned for protocol-specific signatures, magic bytes, and header patterns to accurately classify the protocol regardless of transport-layer configuration. Key fields and metadata are then extracted and streamed to user space through high-throughput, lock-free ring buffers.

In user space, the correlation engine reconstructs bidirectional TCP sessions by tracking sequence and acknowledgment numbers. It handles retransmissions and out-of-order segments to reassemble complete transactions. This session-aware analysis enables accurate parsing of application-layer protocols and extraction of metadata such as HTTP methods, DNS queries, and Redis commands. Packet-level metrics are aggregated into session-level quality indicators, including latency, jitter, throughput, and loss, which support SLA monitoring and anomaly detection. The resulting flow records are forwarded to the Manager for centralized enrichment and analysis.

C. Hierarchical Flow Telemetry Synthesis

As shown in Fig. 4, the manager aggregates telemetry from all nodes and synthesizes it into hierarchical observability layers. At the lowest level, packet metadata such as IP addresses, ports, payload sizes, and timestamps are collected. These are enriched with Kubernetes-specific context, including pod identifiers, namespace, deployment configuration, and security annotations. At the pod level, *cnFlow* builds comprehensive communication profiles that include API endpoints, response times, error rates, and authorization information, providing visibility into service-to-service dependencies and enabling early detection of performance bottlenecks or policy violations.

At the namespace and cluster-wide levels, *cnFlow* aggregates interactions to uncover macro-level traffic patterns, inter-namespace dependencies, and compliance violations. It supports advanced correlation features that allow administrators to trace complex interactions across namespaces and pinpoint the root cause of anomalies or unauthorized behaviors. To further extend observability, *cnFlow* integrates external sources such as geolocation databases and cloud provider APIs. This multi-dimensional telemetry model is particularly valuable in multi-tenant or geographically distributed clusters where regulatory compliance and global traffic analysis are required.

IV. Evaluation

A. Test Environments

A three-node Kubernetes v1.29.15 cluster was deployed on virtual machines running Ubuntu 22.04. Each node was provisioned with 4 vCPUs, 8 GB of RAM, and 256 GB of storage on a single physical machine. Flannel was configured as the primary Container Network Interface (CNI). For comparative evaluation, Istio [6] v1.26.2 was deployed in Permissive mode with Envoy sidecar injection enabled for all application pods. Kubeshark [21] v52.7.8, a Kubernetes-native traffic analyzer, was also deployed to facilitate comparative analysis.

B. Hierarchical Telemetry Validation with Contextual Insights

cnFlow captures diverse network traffic and enriches it with Kubernetes metadata, including namespaces, pod names, service accounts, security contexts, and geographic information. This telemetry provides multi-level observability, enabling precise correlation of network sessions and improved operational understanding that existing solutions typically lack.

At the pod level, *cnFlow* reconstructs request–response interactions, extracting protocol details such as HTTP methods, API endpoints, and DNS queries (Fig. 5(a-b)). Linking these transactions with pod-level metadata allows detailed inspection of individual microservice behavior, which is essential for debugging and security management.

At the namespace level, *cnFlow* aggregates traffic to uncover service interactions and dependencies. It parses Kafka topics and Redis commands, mapping them to corresponding pods and services (Fig. 5(c)), thereby clarifying communication patterns within the namespace and supporting effective operational control.

```
[PUT, /api/users/1, Request, PSH+ACK]
default/http-client(South Korea, 10.244.0.12:49626, pod-network, sa:default)
=== HTTP/1.1(0.20ms, uid:root, ttl:64, seq:1388819361) ===>
default/http-server(South Korea, 10.244.0.13:8080, pod-network, sa:default)
```

(a) Pod-level HTTP request–response timeline

```
[NOERROR, A, google.com, udp:36B]
default/http-client(South Korea, 10.244.0.12:43398, pod-network, sa:default)
=== DNS/UDP(uid:root, ttl:64, IPv4 address lookup) ===>
dns-server(10.96.0.10:53, external-dns)
```

(b) Pod-level DNS query–response timeline

```
Timestamp: 2025-07-10 06:25:35 Namespace: default
ServiceAccounts:
 └─ default (Role: cache-manager, Capabilities: none)
Pods Composition:
 ├─ kafka-server     (Gyeonggi-do, South Korea, podNetwork)
 ├─ redis-client     (Gyeonggi-do, South Korea, podNetwork)
 └─ zookeeper-server (Gyeonggi-do, South Korea, podNetwork)
Protocol Distribution:
 ├─ KAFKA:  2 topics   (cnflow, ictc), 26.7ms avg
 └─ REDIS:  2 commands (SADD, ZADD), 3.1ms avg
```

(c) Namespace-level Kafka and Redis transaction statistics

```
Timestamp: 2025-07-10 07:40:59 Cluster: ICTC cluster
Protocol Distribution              | Traffic Flow Analysis
___________________________________|_____________________________________
HTTP/1.1: 400 flows (91.7%, 6.5ms) | Internal(Pod↔Pod): 132 flows (0.14ms)
HTTP/2:   24 flows (5.5%, 0.08ms)  | Inbound(External→Pod): 152 flows (16.68ms)
ICMP:     12 pings (2.7%, 32.8ms)  | Outbound(Pod→External): 152 flows (16ms)

Geographic Distribution            | Security Overview
___________________________________|_____________________________________
Domestic:      72.8% (4.14ms avg)  | Active Pods:   10 pods
International: 27.2% (32.35ms avg) | Privileged:    1 pods (10.0%)
                                   | Host Network:  4 pods (40.0%)
```

(d) Cluster-wide multi-protocol traffic overview (HTTP/1.1, HTTP/2, ICMP)

Fig. 5: Protocol-coverage validation across pod, namespace, and cluster scopes using *cnFlow*. Subfigures demonstrate that *cnFlow* faithfully captures and reconstructs transactions.

At the cluster level, *cnFlow* consolidates telemetry across all namespaces, capturing multi-protocol interactions such as HTTP/2 streams and ICMP requests. By combining geographic and security metadata, it distinguishes internal, inbound, and outbound traffic (Fig. 5(d)) and facilitates analysis of regional traffic characteristics, latency variations, and compliance with data sovereignty requirements.

Compared to conventional monitoring solutions that offer only basic protocol coverage and limited contextual data, *cnFlow* leverages an eBPF-based architecture to provide multi-scope observability. This capability enhances performance analysis, accelerates fault diagnosis, and strengthens security and compliance management in Kubernetes environments.

C. Performance Evaluation

We evaluated the performance overhead of *cnFlow* in terms of throughput, latency (Fig. 6), and CPU utilization. Throughput was measured under concurrency levels of 10, 25, 50, and 100. Latency was evaluated as the round-trip time (RTT) for complete request–response cycles.

For HTTP/1.1 and HTTP/2 workloads (Fig. 6(a–b,f–g)), *cnFlow* introduced throughput reductions of 31.2% and 18.7%, respectively. In comparison, Kubeshark exhibited higher overheads of 38.7% and 41.5%, while Istio showed significant degradation of 84.7% and 84.3%. Latency increased by 60.6% and 15% under *cnFlow*, compared to 224.6% and 30.6% for Kubeshark and 477.7% and 560% for Istio. These results can be attributed to the fact that *cnFlow* performs parsing and metadata extraction entirely in kernel space, whereas Kubeshark incurs additional user-space copying and TLS inspection overhead, and Istio introduces heavy sidecar-proxy overhead and frequent context switching.

Redis traffic (Fig. 6(c,h)) showed an 8.5% throughput decrease under *cnFlow*, owing to efficient in-kernel protocol detection and command parsing. By contrast, Kubeshark and Istio experienced more substantial reductions of 29.3% and 53.1%, respectively, due to their reliance on resource-intensive user-space inspection. Redis latency increased by 17% with *cnFlow*, whereas Kubeshark and Istio observed increases of 45.8% and 76.1%, respectively.

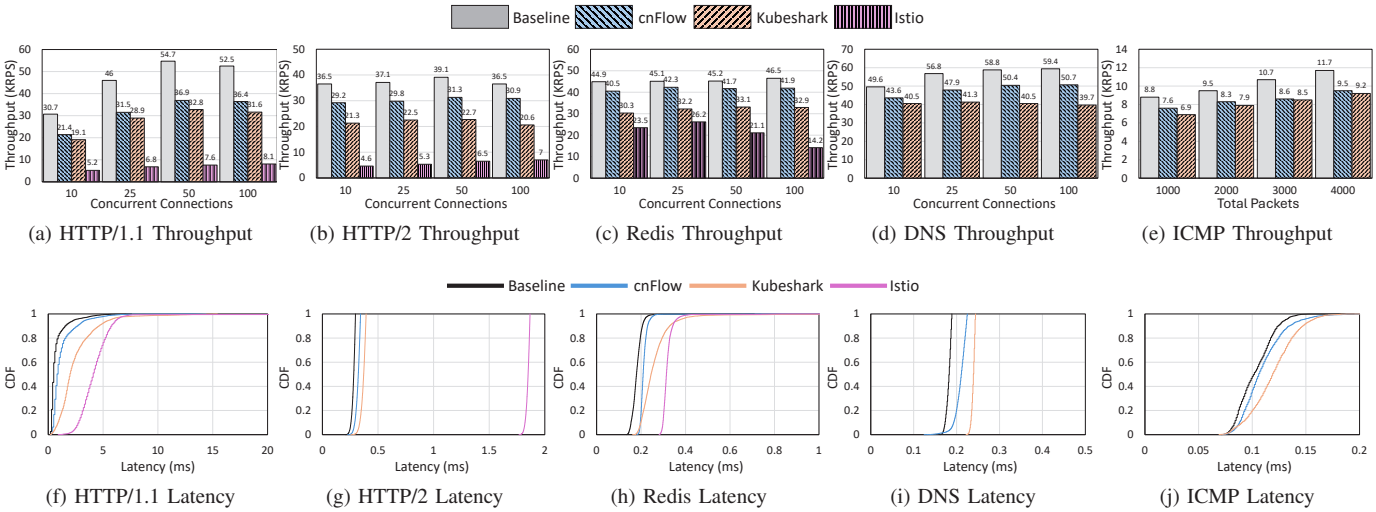
For DNS traffic (Fig. 6(d,i)), *cnFlow* reduced overall throughput by only 14.2%, significantly lower than Kubeshark's 27.5% overhead caused by repeated kernel-to-user data transfers. DNS latency overhead for *cnFlow* remained modest at 15%, compared to 31.5% for Kubeshark.

ICMP performance (Fig. 6(e,j)) showed a 16.3% throughput decrease for *cnFlow*, resulting from precise per-packet metadata capture. Kubeshark exhibited a slightly higher overhead of 20.3% due to additional user-space event correlation complexity. Latency overhead was minimal for *cnFlow* at 6.5%, compared to 15.8% for Kubeshark.

CPU utilization for *cnFlow* remained low, averaging 1.82% in user space and 1.57% in system space. Kubeshark consumed slightly more resources, with 1.93% user and 1.94% system usage, while Istio exhibited significantly higher consumption at 5.93% user and 1.55% system. These results demonstrate the efficiency of *cnFlow*'s in-kernel telemetry processing approach, which minimizes overhead while maintaining comprehensive observability.

V. Related Work

Network Telemetry and Analytics. Prior work on network telemetry has focused on performance and scalability, but they often lack the contextual richness required in dynamic Kubernetes environments. Akbari et al. [5] proposed a sustainable telemetry framework for Beyond 5G systems using tools such as Kepler and Prometheus, but without integration of workload metadata such as pod identity or namespace. Similarly, PCANT [15] streamlines deployment via programmable components but falls short in capturing control-plane context, limiting its applicability for fine-grained policy analysis. Compact telemetry formats introduced by Landau-Feibish et al. [14] enable efficient data handling, yet lack hierarchical organization and application-layer insights. In contrast, *cnFlow* combines low-level packet metrics with Kubernetes-native metadata to generate enriched, hierarchical telemetry suitable for detailed operational diagnostics and multi-tenant policy enforcement.



(a) HTTP/1.1 Throughput (b) HTTP/2 Throughput (c) Redis Throughput (d) DNS Throughput (e) ICMP Throughput

(f) HTTP/1.1 Latency (g) HTTP/2 Latency (h) Redis Latency (i) DNS Latency (j) ICMP Latency

Fig. 6: End-to-end throughput (top row) and latency (bottom row) for HTTP/1.1, HTTP/2, Redis, DNS and ICMP.

eBPF-Based Network Observability. eBPF has emerged as a key enabler for in-kernel observability due to its efficiency and flexibility. Sauron [18] applies eBPF to 5G/6G networks for security and performance monitoring but offers limited integration with container orchestration platforms. eZtunnel [17] reduces service-mesh overhead by bypassing sidecars with eBPF, yet omits telemetry synthesis or contextual enrichment. NetworkShortcut [19] uses eBPF for latency optimization but does not support session reconstruction or protocol-aware analysis. *cnFlow* extends these efforts by combining event-driven eBPF programs with user-space correlation and Kubernetes-aware enrichment, offering a multi-dimensional telemetry platform that scales to complex production environments.

VI. Conclusion

This paper introduced *cnFlow*, an eBPF-based observability framework tailored for Kubernetes environments. *cnFlow* overcomes key limitations of conventional monitoring systems by delivering protocol-aware telemetry enriched with Kubernetes-native metadata, including pod identities, namespaces, service accounts, security contexts, and geographic location. Its hierarchical and context-aware design enables fine-grained visibility across pod, namespace, and cluster scopes, facilitating precise performance analysis and policy compliance tracking. Through empirical evaluation with diverse protocols (HTTP/1.1, HTTP/2, Redis, DNS, and ICMP), *cnFlow* achieved substantially lower overhead than representative tools such as Kubeshark and Istio. By combining in-kernel efficiency with user-space correlation and contextual enrichment, *cnFlow* provides a scalable and practical observability solution for production-grade cloud-native infrastructures.

Acknowledgment

This work was supported by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIT) (No. RS-2023-00212738).

References

[1] Linux Foundation Networking, "Global Open Source Networking Survey Reveals Massive Insights into Cloud Native Adoption, OpenRAN, and Domain-Specific AI Priorities, with over 92% Relying on Open Source Projects," The Linux Foundation, Mar. 2025.
[2] "Kubernetes," https://kubernetes.io/.
[3] "Multus CNI," https://github.com/k8snetworkplumbingwg/multus-cni.
[4] "SR-IOV CNI plugin," https://github.com/openshift/sriov-cni.
[5] M. Akbari, R. Bolla, R. Bruschi, F. Davoli, C. Lombardo, and B. Siccardi, "A Monitoring, Observability and Analytics Framework to Improve the Sustainability of B5G Technologies," in *International Conference on Communications Workshops*. IEEE, 2024, pp. 969–975.
[6] "Istio," https://istio.io/.
[7] "Linkerd," https://linkerd.io/.
[8] X. Zhu, G. She, B. Xue, Y. Zhang, Y. Zhang, X. K. Zou, X. Duan, P. He, A. Krishnamurthy, M. Lentz, D. Zhuo, and R. Mahajan, "Dissecting Overheads of Service Mesh Sidecars," in *ACM Symposium on Cloud Computing*. ACM, 2023, pp. 142–157.
[9] "eBPF: Extended Berkeley Packet Filter," https://ebpf.io/.
[10] C. Pahl, A. Brogi, J. Soldani, and P. Jamshidi, "Cloud Container Technologies: A State-of-the-Art Review," *Transactions on Cloud Computing*, vol. 7, no. 3, pp. 677–692, 2019.
[11] "sFlow," https://sflow.org/.
[12] "Cisco IOS and NX-OS NetFlow," https://www.cisco.com.
[13] "IPFIX," https://datatracker.ietf.org/wg/ipfix/about/.
[14] S. Landau-Feibish, Z. Liu, and J. Rexford, "Compact Data Structures for Network Telemetry," *ACM Computing Surveys*, vol. 57, no. 8, pp. 1–31, Mar. 2025.
[15] R. Varloot and L. Noirie, "PCANT: Programmable Capture and Analysis of Network Traffic," in *Conference on Innovation in Clouds, Internet and Networks and Workshops*. IEEE, 2023.
[16] "Envoy," https://www.envoyproxy.io/.
[17] F. E. Rodriguez Cesen and C. E. Rothenberg, "eZtunnel: Leveraging eBPF to Transparently Offload Service Mesh Data Plane Networking," in *International Conference on Cloud Networking*. IEEE, Nov. 2024.
[18] D. Soldani, P. Nahi, H. Bour, S. Jafarizadeh, M. F. Soliman, and L. Di Giovanna, "eBPF: A New Approach to Cloud-Native Observability, Networking and Security for Current (5G) and Future Mobile Networks (6G and Beyond)," *IEEE Access*, vol. 11, May 2023.
[19] W. Yang, P. Chen, G. Yu, H. Zhang, and H. Zhang, "Network shortcut in data plane of service mesh with eBPF," *Journal of Network and Computer Applications*, vol. 222, no. C, p. 103805, Feb. 2024.
[20] J.-Y. Kim and J. W.-K. Hong, "Distributed QoS monitoring and edge-to-edge QoS aggregation to manage end-to-end traffic flows in Differentiated Services networks," *Journal of Communications and Networks*, vol. 3, no. 4, pp. 324–333, Dec. 2001.
[21] "Kubeshark," https://www.kubeshark.co/.