

Article

Closed-Loop CPU-Aware Traffic Control for SDN-Enabled 5G/6G Networks in Open vSwitch Dataplanes

Stefan Biševac ¹, Živko Bojović ¹, Petar D. Bojović ^{2,*} and Ilija Doknić ³

¹ Faculty of Informatics and Computing, Singidunum University, 11000 Belgrade, Serbia; stefan.bisevac@kg.ac.rs (S.B.)

² The School of Computing, Union University, 11000 Belgrade, Serbia

³ The Faculty of Organizational Sciences, University of Belgrade, 11000 Belgrade, Serbia

* Correspondence: petar.bojovic@paxy.in.rs

Highlights

What are the main findings?

- The proposed closed-loop system maintains CPU usage strictly below configured targets (e.g., 1.02% vs. 3% target) through real-time XDP-based rate limiting at the NIC ingress, preventing data-plane overload.
- Predictable per-slice CPU isolation is achieved only within the Safe zone, while saturated conditions require hardware offloading such as SmartNICs to ensure strict isolation.

What are the implications of the main findings?

- Network operators can deploy the framework on commodity hardware to provide predictable CPU isolation between 5G/6G network slices, ensuring that traffic surges in one slice do not degrade performance in others.
- The results underscore the need for hardware-assisted traffic shaping (e.g., SmartNICs) to eliminate residual cross-slice interference and achieve full isolation at line rate.

Abstract

This paper presents a closed-loop, CPU-aware traffic-control framework for SDN in 5G/6G multi-tenant edge environments based on commodity KVM/OVS infrastructures. It couples fine-grained data-plane telemetry via eBPF with adaptive XDP rate limiting, coordinated by a PID controller in the OVS datapath. Unlike control-plane polling, it provides real-time feedback between CPU utilization and traffic regulation. Experiments in a virtualized multi-tenant OVS testbed (KVM/virtio-net) keep CPU below per-slice CPU targets (e.g., 1.02% for a 3% setpoint), with an under-target bias that avoids overshoot while preserving stable forwarding. We attribute this bias, at light load, to a supply-limited regime, conservative per-slice CPU accounting, and stability-oriented PID tuning, and introduce a low-latency profile that mitigates this bias for latency-sensitive slices. The XDP datapath achieves 1–3 μ s per-packet processing with 5–10% additional CPU overhead relative to an uninstrumented baseline, while using less CPU than OVS policing at comparable throughput. A 3% per-slice CPU target balances isolation and throughput, while 2% yields stricter isolation at the cost of higher packet loss. Software-based rate limiting can induce cross-slice interference; effective isolation holds below 1 Gbps aggregate load. Above this, shared Linux kernel overhead degrades isolation, causing significant loss; thus, XDP alone cannot ensure line-rate isolation, motivating SmartNICs. The design improves efficiency, predictability, and isolation, laying a foundation for intelligent traffic management in future resource-intensive applications.



Academic Editor: Pierluigi Siano

Received: 11 April 2026

Revised: 16 May 2026

Accepted: 20 May 2026

Published: 25 May 2026

Copyright: © 2026 by the authors.

Licensee MDPI, Basel, Switzerland.

This article is an open access article distributed under the terms and

conditions of the [Creative Commons](https://creativecommons.org/licenses/by/4.0/)

[Attribution \(CC BY\)](https://creativecommons.org/licenses/by/4.0/) license.

Keywords: Software-Defined Networking (SDN); Open vSwitch (OVS); eBPF; XDP; PID controller; traffic control; CPU-aware networking; 5G/6G; network slicing; smart cities

1. Introduction

Software-Defined Networking (SDN) separates the control of the network from the process of forwarding data. This separation allows for centralized programming and flexible management of the network. It leverages protocols such as OpenFlow to facilitate these changes [1]. This paradigm has become fundamental for dynamic infrastructures in cloud, edge, and emerging 5G/6G environments [2,3], where networks must continuously adapt to rapidly changing traffic demands and service requirements [4,5].

A representative example arises in edge–cloud environments supporting 5G/6G network slicing. In such systems, multiple slices share the same OVS-based dataplane on commodity hardware. When one slice experiences a sudden traffic surge, CPU resources may saturate, leading to increased latency and degraded throughput for other slices. This highlights CPU as a critical bottleneck and motivates the need for real-time, resource-aware traffic control mechanisms.

As SDN deployments grow in scale and complexity, ensuring stable forwarding performance in software-based dataplanes remains challenging. In many systems, packet forwarding is handled by OVS running on commodity servers. The processing capacity of such dataplanes depends heavily on CPU availability. Under high load, packet processing latency rises, and throughput may degrade, leading to unstable performance. Prior studies have shown that OVS performance is strongly affected by CPU contention. This makes it crucial to manage traffic based on resources in 5G and 6G environments, especially those that require strict guarantees for low latency and reliability. These environments often support multiple users and network slices. Most research on SDN optimization has focused on the control plane, improving scalability, reliability, and controller placement through distributed and hierarchical designs [6–9]. Although these methods enhance global network visibility, they rely on summary data and offer limited insight into runtime dataplane dynamics. Recent studies on software switches and programmable dataplanes (e.g., P4, eBPF) confirm that CPU usage is a dominant factor affecting packet forwarding stability [10]; however, many traffic management systems continue to overemphasize control-plane metrics and rely on coarse-grained measurements. Existing monitoring solutions, such as PayLess [11], provide low-cost controller-based data collection but rely on periodic polling, which prevents high-frequency visibility into dataplane states and fails to capture the system’s fine-grained dynamics.

The Extended Berkeley Packet Filter (eBPF) enables safe kernel instrumentation to monitor per-packet processing costs, while the eXpress Data Path (XDP) offers efficient rate control and early packet handling at the network ingress. This paper proposes a closed-loop, CPU-aware traffic control framework for SDN networks using OVS dataplanes that integrates high-frequency dataplane telemetry with an adaptive feedback loop to dynamically regulate traffic flows based on measured CPU utilization. The system uses a PID controller to map target CPU usage levels to real-time rate limits, preventing datapath overload and maintaining stable forwarding performance while improving efficiency, predictability, and fairness across traffic flows in virtualized switching environments.

The main contributions of this paper are as follows:

- C1. Fine-grained CPU utilization tracking using eBPF probes integrated into OVS datapaths.
- C2. High-performance rate limiting via XDP for early-stage traffic regulation.
- C3. Closed-loop control algorithm that maps CPU utilization to adaptive throughput limits.

- C4. Empirical characterization of CPU–throughput relationships and stability thresholds in virtualized OVS environments.
- C5. Demonstrated improvements for software-based switching in virtualized OVS dataplanes representative of commodity 5G/6G edge deployments.

The rest of the paper is organized as follows: Section 2 reviews related work; Section 3 describes the proposed system design; Section 4 details the implementation; Section 5 presents the test environment; Section 6 provides experimental results and evaluation; and Section 7 concludes the paper.

2. Related Work

In this section, we review research in SDN and 5G/6G networks covering network monitoring methods—including controller-based telemetry and lightweight dataplane instrumentation via eBPF and XDP. While most prior work has focused on enhancing control-plane scalability and reliability, less attention has been devoted to the dynamics of software-based dataplanes such as OVS. This review establishes the research context and motivates the proposed CPU-aware traffic control framework.

2.1. SDN Control Plane Scalability and Reliability

Early SDN designs used one central controller, which had a complete view of the network and managed all the devices that forward data. This setup simplified network programmability and rule enforcement. The system encountered scalability and fault-management challenges, as Kreutz et al. [1], Nunes et al. [2], and Xia et al. [3] have documented. To overcome these challenges, several distributed and hierarchical control-plane models have been proposed, including HyperFlow [7], Kandoo [8], and the Elastic SDN Controller [9]. These controllers delegate tasks across multiple entities, enhancing decentralized decision making, synchronization, scalability, and responsiveness. Subsequent work on controller placement, load balancing, and state consistency [6,12,13] has further optimized control-plane performance in large-scale deployments. These distributed control-plane designs focus primarily on improving coordination, reliability, and fault recovery within the control layer. Most assume ideal dataplane conditions while overlooking hardware limitations of forwarding devices, such as CPU utilization, memory pressure, and NIC load [10,14,15]. Advanced distributed control systems perform poorly under data-processing overload, necessitating resource awareness. The CPU-aware traffic control framework improves performance by making control decisions based on real-time data [11]. This observation highlights the necessity of integrating resource awareness into SDN control, motivating the proposed CPU-aware traffic control framework.

2.2. SDN Data Plane Performance and Software Switching

While extensive research has focused on scalability and coordination in the SDN control plane [2,3,12,13], software-based forwarding elements in the data plane have received comparatively less attention. Among these, OVS stands as the de facto software switch in virtualized and cloud infrastructures, providing OpenFlow-compatible packet forwarding through a modular architecture that integrates userspace and kernel modules for flexible traffic management and programmability [1]. However, its performance is intrinsically tied to hardware resources, most notably CPU availability. Reference [10] presents the design and implementation details of OVS, demonstrating its flexibility across virtualization platforms but also revealing that the interaction between kernel datapaths and userspace modules introduces processing overhead. Later studies conducted detailed performance evaluations of OpenFlow-enabled and software-switch architectures [12,14], confirming that packet-processing throughput and latency in OVS are highly sensitive to CPU con-

tention and scheduling. Under heavy loads or multi-tenant deployments, multiple user processes compete for limited CPU cycles and memory bandwidth, leading to performance fluctuations and degraded quality of service. Recent experimental analyses also emphasize the resource contention problem in virtualized dataplanes [16]. When virtual machines and applications share CPUs or I/O channels, OVS performance degrades, causing unstable throughput and variable jitter even when available network bandwidth remains constant. This sensitivity is critical for 5G and 6G networks, which require low-latency and high-reliability data forwarding [4,5]. Despite the recognized link between dataplane performance and resource conditions, existing SDN designs still treat software switches as passive execution entities with fixed capacity. Current optimization methods often rely on controllers or statistical models to estimate network conditions [17]. They lack real-time feedback from the dataplane. Traditional northbound monitoring tools (e.g., Prometheus, Zabbix) can observe host metrics such as CPU, memory, and interface counters [18,19], but they are passive collectors with no real-time control feedback. Bhamare et al. [20] highlight that the absence of closed-loop coordination between control and data planes prevents the system from reacting to transient overload or CPU bottlenecks. Consequently, although prior work has firmly established the impact of hardware and CPU limitations on OVS performance, active control mechanisms that dynamically regulate dataplane resource usage have remained largely unexplored. These gaps motivate the development of resource-aware feedback frameworks that couple real-time CPU and memory telemetry from OVS with adaptive flow-control logic. Such closed-loop mechanisms enable SDN systems to not only monitor, but also actively stabilize dataplane performance under variable workloads, a direction further pursued in this work.

2.3. Network Monitoring and Telemetry Approaches

Effective monitoring in SDN networks depends on the ability of the control plane to collect and interpret states from the data plane in a timely manner. The earliest approaches, such as the PayLess framework [11,21], enabled simple, cost-effective monitoring through periodic controller-side polling. Although these systems provide basic visibility into throughput and active flows, their periodic nature introduces a significant delay between event detection and controller response; moreover, they manage only control functions and lack visibility into hardware resource states—CPU, memory, and interface usage—in the dataplane. In parallel, experiments and tools have been developed that enable dataplane performance testing, such as the OFLOPS framework [14], which evaluates OpenFlow-compatible switches under different loads. Although such mechanisms provide limited precision in packet-processing-time measurement, they serve primarily for performance evaluation and do not have integrated control feedback functionality. Similarly, external monitoring systems, such as Prometheus, Zabbix, and Nagios, allow the collection of metrics about the CPU, memory, and network interfaces of the hosts. However, they are passive tools whose role is limited to displaying and alerting, without the ability to influence the flows in the SDN data plane directly. Recent advances in Linux kernel technologies, such as eBPF and XDP [18,21], have enabled high-frequency telemetry and “in-band” measurements within the dataplane, thereby increasing the granularity of observing network traffic and processing costs. However, the application of these techniques in SDN ecosystems is still not associated with active control mechanisms. Existing solutions collect enough data for passive monitoring but lack a closed feedback loop that allows immediate correction of network behavior depending on resource conditions. The proposed CSRM framework addresses this gap through gRPC-based bidirectional telemetry. The CSRM framework enables real-time coupling of hardware load data with control

decisions, representing a significant advance toward fully autonomous and resource-aware monitoring in SDN.

2.4. eBPF and XDP for Dataplane Observability and Control

The extended Berkeley Packet Filter (eBPF) is a sandboxed execution environment built into the Linux kernel [18,22]. eBPF allows developers to create custom programs that attach to kernel events. These programs inspect network packets and monitor system performance without modifying the kernel. Built on top of eBPF, the XDP provides a high-performance, programmable packet-processing layer at the earliest point of packet reception within the Linux network stack [23]. Together, eBPF and XDP offer a foundation for flexible and efficient dataplane programmability, allowing custom packet handling, telemetry extraction, and event-driven control at the network node level. Recent studies have explored the integration of eBPF and XDP for network performance monitoring, DDoS mitigation, and traffic shaping in cloud and edge computing environments [18]. Høiland-Jørgensen et al. [23] demonstrated that XDP can process packets with low and predictable overhead, achieving line-rate performance on commodity hardware. Their work also demonstrated XDP's suitability for implementing traffic control mechanisms such as rate limiting, where packets can be dropped at the earliest point in the network stack with minimal CPU overhead. Similarly, Ref. [24] introduced Aquila, an efficient in-kernel telemetry framework for cloud-native environments, which demonstrates the scalability and low overhead of eBPF-based instrumentation in production-grade systems. Tomovic et al. [25] proposed a run-time adaptive in-kernel BPF/XDP solution for 5G User Plane Function (UPF), demonstrating the feasibility of eBPF-based packet processing in carrier-grade 5G deployments with dynamic adaptation to network conditions. Their work validates the applicability of XDP for high-throughput, low-latency packet handling in 5G core networks and establishes the effectiveness of instrumenting kernel networking functions (such as `netif_receive_skb` and `__dev_queue_xmit`) for CPU time measurement. However, their approach targets single-tenant UPF deployments and does not address per-slice CPU attribution in multi-tenant environments. Despite the growing adoption of eBPF/XDP in kernel-space networking, relatively few research efforts have examined how these technologies can enhance dataplane observability and adaptive resource control in SDN environments with multiple isolated network slices sharing common kernel infrastructure. Building upon these advances, this work leverages eBPF probes and XDP hooks to achieve closed-loop CPU-aware control within OVS dataplanes. We propose an approach that extends beyond passive monitoring by integrating high-frequency telemetry and adaptive feedback mechanisms that dynamically regulate flow throughput based on real-time CPU utilization. This tight coupling of data monitoring and control enables rapid, adaptive resource management, addressing the limitations of traditional monitoring and enhancing stability and efficiency of software-based switching in SDN-based 5G and 6G environments.

2.5. Resource-Aware and Closed-Loop SDN Control

Resource-aware control mechanisms have long been studied as a means to enhance the adaptability and efficiency of networked systems [26]. Several studies have explored feedback-based optimization and control-theoretic models for load balancing across controllers and flow scheduling in SDN [26,27]. Traditional approaches employ control-loop algorithms (e.g., Proportional-Integral-Derivative controllers) to stabilize network performance under fluctuating traffic demands [27]. Santos et al. [28] demonstrate that machine learning can improve resource management. These ML-based controllers predict resource usage and adjust settings in real time, yielding better scalability and faster response times

in large-scale SDN systems. Despite these advances, most resource-aware mechanisms remain confined to the control plane, where they influence decision making at a relatively coarse time scale. The dataplane, where packet forwarding and processing occur, typically lacks direct awareness of resource conditions (e.g., CPU load, queue occupancy, or memory pressure). The separation between telemetry collection and actuation introduces latency, limiting the system's responsiveness to overloads or congestion and causing transient performance degradation, especially in software-based dataplanes such as OVS, where CPU contention directly impacts packet throughput. This paper presents a new control method that actively manages CPU usage. It operates above the dataplane layer in a closed-loop system, where real-time data from eBPF/XDP tools is used to adjust flow transmission rates based on resource availability. It continuously regulates these rates using feedback logic. The integration of observability and control in the dataplane allows for low-latency adaptation that enhances traditional controller-level optimization. This results in a more responsive and self-stabilizing forwarding plane, maintaining high throughput under variable processing loads.

SDN-based resource management has attracted considerable research interest for 5G/6G network slicing, particularly in smart city deployments where real-time applications demand reliable network performance [29]. Botez et al. [30] proposed an efficient network slicing mechanism with SDN and heuristic algorithms for 5G networks, focusing on Kubernetes-based NFV orchestration and dynamic resource allocation across network slices. Similarly, Alhussein et al. [31] presented an SDN-based network slicing mechanism for scalable 4G/5G core networks, addressing the challenges of multi-tenant resource isolation and QoS guarantees. While these works demonstrate the effectiveness of SDN for slice-level resource orchestration, they primarily focus on control-plane coordination and do not address dataplane-level CPU awareness or closed-loop feedback control within software switches.

Prior research has shown that resource-aware feedback control remains valuable in SDN. However, most existing implementations operate at timescales incompatible with real-time dataplane dynamics. We propose a new system that integrates monitoring and control with eBPF/XDP techniques, embedding a closed-loop control model in the datapath. This enables rapid flow-rate adaptation, enhancing responsiveness to CPU-level packet-processing load.

2.6. Application of Closed-Loop CPU-Aware Traffic Control in Smart City Scenarios

Smart city platforms aggregate heterogeneous latency-sensitive services over shared edge/cloud infrastructures, increasingly enabled by 5G/6G networks. In such environments, contention for shared computational resources makes CPU overload in the data plane a key factor affecting latency, responsiveness, and reliability. Consequently, Closed-Loop CPU-Aware Traffic Control has been investigated in smart city scenarios, including V2X communication, IoT sensor aggregation, and edge-based video analytics, where strict service requirements necessitate efficient resource management. These cross-domain requirements are typically addressed through closed-loop CPU-aware mechanisms combining ingress traffic policing, per-slice CPU monitoring in the data plane, and feedback-based control of computational budgets. To meet these requirements, we implement a closed-loop design that integrates XDP-based token-bucket ingress policing, eBPF-driven per-slice CPU telemetry in Open vSwitch, and a PID controller that enforces slice-level CPU targets through runtime OVS policy updates.

Enhanced V2X (eV2X) communication in 5G systems encompasses V2V, V2I/V2P, and V2N services supporting applications such as platooning, cooperative driving, and intersection assistance [32]. These services impose stringent requirements on end-to-end

latency, jitter, reliability, and service continuity under high mobility, along with traffic prioritization and congestion control. Such characteristics motivate adaptive CPU-aware traffic management approaches that prevent dataplane overload and maintain per-slice computational budgets, ensuring resource availability for latency-critical vehicular services under dynamic load conditions. For eV2X/advanced driving, typical end-to-end targets include reliability of up to approximately 99.999% and latency in the range of 3–10 ms. These requirements can be addressed by per-slice CPU budgets enforced via a PID controller driven by eBPF-derived OVS telemetry, coupled with XDP-based ingress policing that constrains non-critical flows to reserve CPU headroom for vehicular traffic. City-wide IoT sensor aggregation systems require scalable data collection, event processing, and device management across heterogeneous network environments [33]. Key requirements include scalability, interoperability, and efficient edge processing to reduce latency and backhaul overhead. The inherently bursty and distributed nature of IoT traffic further emphasizes the need for adaptive CPU-aware resource allocation mechanisms capable of preventing overload propagation in shared edge infrastructures. These characteristics can be mapped to fine-grained eBPF-based CPU monitoring with PID-controlled per-slice targets and XDP-based ingress rate limiting to mitigate overload propagation across shared dataplane resources. For IoT aggregation, the objective is to achieve high reliability across a wide range of transmission periodicities, while latency requirements typically range from sub-second to second-level timescales.

Multi-access Edge Computing (MEC) extends cloud capabilities to the network edge, enabling low-latency processing for applications such as video analytics, AR/VR, industrial automation, and public safety [34]. For video analytics and other MEC applications, typical end-to-end latency budgets range from 50 to 100 ms, depending on the application. MEC environments require multi-tenant isolation, dynamic resource orchestration, and mobility support across distributed edge nodes. These requirements motivate adaptive CPU-aware traffic control mechanisms capable of maintaining service isolation and performance stability under varying load conditions. These requirements can be addressed by combining adaptive OVS traffic steering with PID-controlled per-slice CPU budgets and XDP-based ingress policing across distributed edge nodes.

Despite significant progress in these domains, existing approaches generally treat traffic management and CPU resource allocation in isolation, lacking unified mechanisms that jointly regulate dataplane forwarding behavior and per-service computational budgets under dynamic load conditions. CPU contention directly degrades QoS in SDN/Open vSwitch-based edge environments, where it increases software dataplane latency under congestion. This gap has motivated unified closed-loop CPU-aware traffic control for SDN-enabled 5G/6G networks, integrating real-time CPU monitoring with adaptive traffic steering and ingress control in Open vSwitch dataplanes, enabling coordinated management of traffic flows and computational resources to improve latency stability, service isolation, and system robustness under varying load conditions. These budgets further motivate a microsecond-scale per-packet datapath and the relocation of control loops outside the hot path. We quantify our XDP-based approach and safe zone behavior in Sections 5 and 6, and discuss the implications of hardware offloading under saturation in Sections 6 and 7.

3. Proposed System Architecture

This section presents the theoretical foundations of the proposed CPU-aware traffic control framework. The architecture integrates SDN principles with closed-loop control theory to achieve real-time resource management in software-based dataplanes.

3.1. System Model Overview

The proposed architecture follows a layered SDN model enhanced with CPU-aware feedback control. Figure 1 illustrates the complete system model, showing the interaction between architectural layers and functional components.

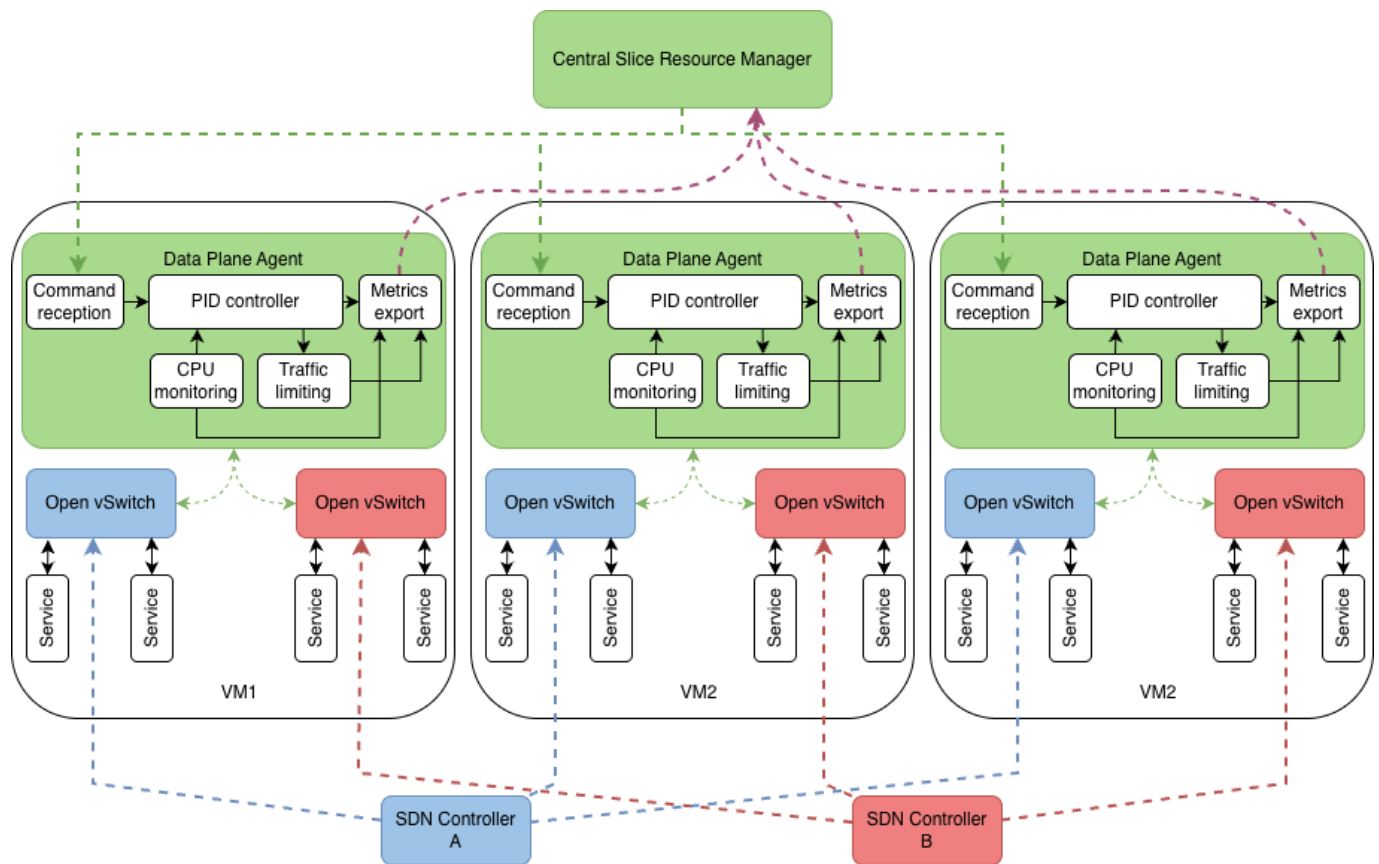


Figure 1. Proposed system architecture showing SDN layers and component interactions. The Central Slice Resource Manager (CSRM) communicates directly with the Data Plane Agent. The Data Plane Agent collects CPU telemetry from OVS instances and enforces rate limits at the NIC level.

The architecture comprises the following key components:

- Data Plane Agent (Per-VM agent): eBPF-based CPU monitoring, XDP rate limiting, gRPC (Google Remote Procedure Call) metrics to CSRM, and CPU-target updates; runs a local PID loop for low-latency closed-loop control.
- OVS instances (multiple OVS bridges per host): Each represents a slice and operates independently, yet all share CPU resources, creating contention the system must manage.
- SDN Controllers A and B manage flows/policies; CSRM is orthogonal, focusing only on CPU-aware rate limiting.
- Services (end-user applications and network services) are connected to each OVS instance, generating and consuming traffic that flows through the managed dataplane.

3.2. Architectural Layer Justification

The placement of components across architectural layers follows fundamental principles of SDN design and control theory. This subsection provides theoretical justification for each design decision.

3.2.1. Data Plane Layer

By operating in the data plane, OVS performs the core packet forwarding function as the most latency-sensitive operation in the network stack. It implements a two-tier forwarding architecture composed of fast path (Kernel Datapath) and slow path (Userspace). On the fast path, the OVS kernel module (`openvswitch.ko`) keeps a flow cache that processes most packets without involving userspace. When packets match stored flows, they are processed directly in the kernel datapath. This results in microsecond-scale latency. On the slow path, packets missing the kernel cache go to the `ovs-vswitchd` daemon for flow table lookup and rule installation. This path offers a specific approach that enhances complex flow-matching capabilities, despite a slight increase in latency. The CPU usage of OVS mainly comes from the fast path, which is the main focus for managing resources. By using eBPF to track kernel functions, the system measures CPU time in the specific areas where resource conflicts occur.

3.2.2. Control Logic Placement

The architecture splits control logic into two tiers. A local Data Plane Agent runs a PID controller on the same host as its OVS instances to minimize latency and support a 500 ms control loop without network delays. This proximity enables rapid reactions to host-level conditions.

A centralized CSRM handles policies requiring global visibility (e.g., fair CPU allocation across hosts and priority-based slice management). By gathering information from different hosts, it connects local actions to the overall goals of the system. This design has a clear structure. By employing fast local loops for stability and slower global loops for optimization this approach helps ensure that the system is both responsive and consistent.

3.3. Component Interaction Model

Understanding the causal relationships between system components is essential for analyzing system behavior. The proposed model follows a hierarchical communication model with three complementary patterns. In the vertical control and data path, the CSRM communicates with Data Plane Agents. It sends CPU commands downwards and receives telemetry data upwards. This communication occurs asynchronously, enabling continued operation despite network delays. Each Data Plane Agent interfaces with its local OVS instances using eBPF maps and kernel interfaces. This tight coupling enables microsecond-scale interactions. Traditional SDN controllers manage flow rules with OVS, and the CSRM maintains this process without interfering, ensuring compatibility with current SDN systems.

3.4. Temporal Action Flow

Figure 2 traces the end-to-end timing of the control loop. A packet from the traffic source first reaches the NIC Filter, which immediately enforces the rate-limiting policy by passing admissible traffic and dropping excess. Packets that are accepted proceed to the Forwarding Engine for processing. During this time, the Forwarding Engine continuously sends metrics to the local Data Plane Agent. On a slower cadence, the agent aggregates and exports telemetry to CSRM, which evaluates policy and global state to compute updated CPU targets. The updated targets are immediately transmitted to the agent (shown by a dotted line). Thus, control-system latency does not affect packet processing. The agent sets a traffic speed limit and configures the NIC filter accordingly. It creates a feedback loop that adjusts traffic based on current conditions. This separation of fast data and slower control paths ensures high-throughput packet processing and timely adaptation.

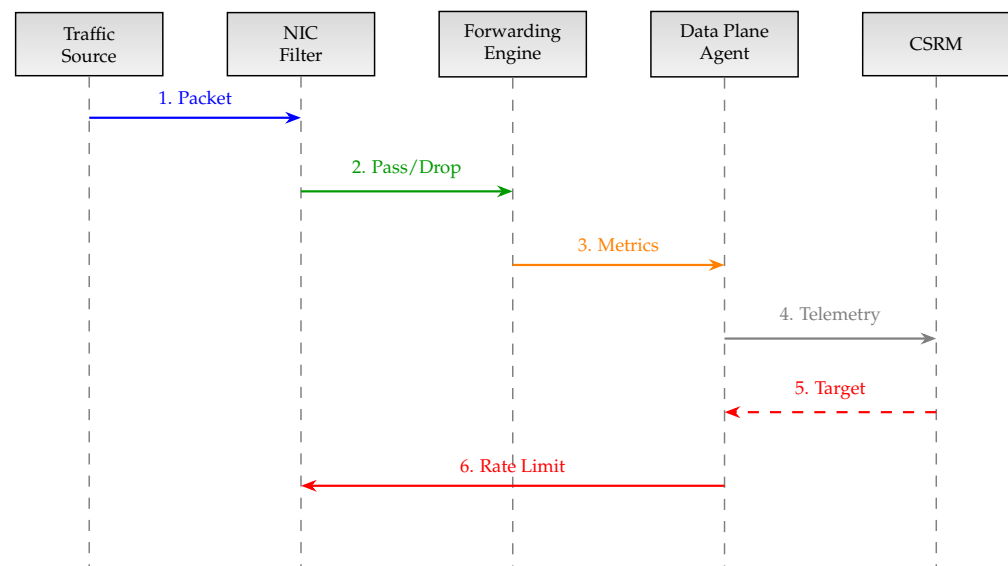


Figure 2. Temporal action flow diagram showing the sequence of operations from packet arrival to rate adjustment. Steps 1–3 occur per-packet; steps 4–6 occur per control interval.

3.5. Closed-Loop Control Model

The proposed framework implements a closed-loop feedback system that continuously adjusts traffic rates based on measured CPU utilization. Figure 3 illustrates the control loop structure, which maps directly to the temporal action flow presented in Figure 2.

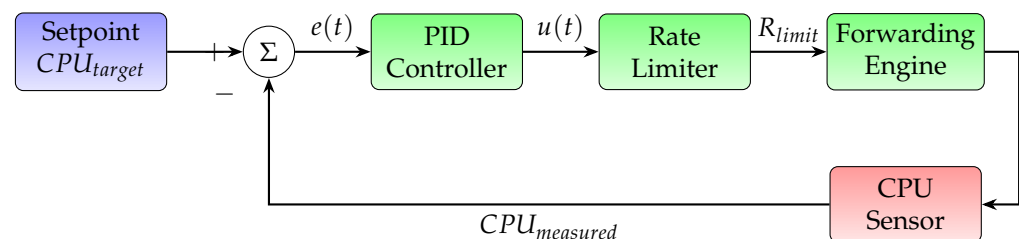


Figure 3. Closed-loop control diagram. The PID controller computes rate adjustments based on the error between target and measured CPU utilization. The Rate Limiter enforces the computed limit, and the CPU Sensor measures the resulting resource consumption.

The control loop comprises three tightly coupled components that regulate ingress bandwidth to a target CPU budget. A CPU sensor continuously measures the CPU time spent on packet processing and reports the resulting utilization ($CPU_{measured}$). The PID controller compares this measurement to the setpoint (CPU_{target}) to compute the error $e(t)$ and produces a control signal $u(t)$ that adjusts the input rate. The rate limiter enforces the resulting bound (R_{limit}) at the network ingress, ensuring that the forwarding engine admits traffic only at that rate. The sensor then observes the effect and feeds the updated utilization back to the controller, closing the loop and ensuring stable adherence to the CPU budget under varying workloads.

The following subsections detail the theoretical models for each component: CPU measurement, rate limiting, and PID control.

3.6. Per-Slice CPU Measurement Challenge

A fundamental challenge in multi-tenant software switching environments is accurately attributing CPU consumption to individual network slices. In virtualized network environments, multiple isolated network slices (implemented as separate namespaces or virtual bridges) share a single operating system kernel. While each slice appears isolated at

the network level, all packet processing occurs through common kernel code paths. The forwarding engine (e.g., a kernel datapath module) is loaded once and serves all slices simultaneously, all slices compete for the same physical CPU cores, and the kernel does not inherently track which slice caused a particular CPU expenditure.

The key insight enabling per-slice CPU measurement is that each packet carries identifying information that can be used to attribute processing time to its originating slice. Attribution can rely on the destination IP address (each slice uses a distinct IP subnet), the ingress interface identifier (packets arrive on slice-specific virtual interfaces), or flow metadata (forwarding rules tag packets with slice identifiers). By instrumenting kernel functions at the point where packet metadata is accessible, the proposed model can record CPU time and associate it with the correct slice.

Measurement Model

To attribute CPU consumption to individual slices, the proposed model must measure the time spent in kernel functions that process packets. The measurement targets four key functions in the Linux networking stack: the OVS datapath entry point (`ovs_dp_process_packet`), virtual Ethernet transmission (`veth_xmit`), network interface reception (`netif_receive_skb`), and device transmit queuing (`__dev_queue_xmit`). Each function receives a socket buffer (`sk_buff`) containing packet metadata, including the network interface index (`ifindex`) that identifies the originating slice. Using eBPF kprobes, the proposed model attaches measurement probes at function entry and exit points, recording timestamps and extracting slice identifiers from packet metadata without modifying kernel source code.

For each instrumented function f , we capture entry and exit timestamps:

$$T_f = t_{exit} - t_{entry} \quad (1)$$

where T_f is the CPU time consumed by function f in nanoseconds. The eBPF program accumulates time across all packet invocations; the total per-slice CPU time over a measurement interval is the sum across all four instrumented functions:

$$CPU_{slice} = \sum_{f \in F} T_f = T_{ovs} + T_{veth} + T_{netif} + T_{xmit} \quad (2)$$

The CPU utilization percentage is then:

$$CPU_{\%} = \frac{CPU_{slice}}{\Delta t \times N_{cores}} \times 100 \quad (3)$$

Normalization and setpoint convention. Unless otherwise stated, $CPU_{\%}$ is normalized to the total host processing capacity across N_{cores} as in (3); that is, 100% corresponds to all cores fully utilized. On the 2-core testbed used in this work, a single fully saturated core appears as 50% total CPU. All per-slice targets (e.g., 2%, 3%, 5%) and reported OVS per-slice utilizations are expressed on this total-host basis rather than per-core.

This approach enables fine-grained, per-slice CPU accounting despite the shared kernel infrastructure, providing the foundation for slice-aware resource management.

3.7. Rate Limiting Model

The proposed model employs a token bucket algorithm for rate limiting, a classical traffic shaping mechanism that provides both rate control and burst tolerance. The bucket state is maintained per slice:

$$tokens(t) = \min(B_{max}, tokens(t - \Delta t) + R_{limit} \times \Delta t) \quad (4)$$

where:

- B_{max} is the maximum bucket size (burst capacity);
- R_{limit} is the configured rate limit (bytes/second);
- Δt is the time since last packet.

A packet of size S bytes is forwarded if $tokens \geq S$; otherwise, it is dropped:

$$action = \begin{cases} PASS & \text{if } tokens \geq S \\ DROP & \text{otherwise} \end{cases} \quad (5)$$

The token bucket provides two key properties: (1) long-term average rate is bounded by R_{limit} , and (2) short-term bursts up to B_{max} bytes are permitted, accommodating bursty traffic patterns common in network applications.

The rate limiter operates at the network ingress point, before packets enter the forwarding engine. This placement ensures that dropped packets consume minimal CPU resources, as they are discarded before incurring the cost of forwarding table lookups and packet transformations.

3.8. PID Controller Design

The proposed model employs a discrete PID (Proportional-Integral-Derivative) controller to regulate CPU utilization by adjusting traffic rate limits.

3.8.1. Control Law

The control output at time step n is:

$$u(n) = K_p \cdot e(n) + K_i \cdot \sum_{j=0}^n e(j) \cdot \Delta t + K_d \cdot \frac{e(n) - e(n-1)}{\Delta t} \quad (6)$$

Each component within a control system possesses a distinct function. The proportional (K_p) term provides an immediate response that is directly related to the current error; thus, larger errors yield proportionately larger corrections. The integral (K_i) term aggregates past errors to eliminate any steady-state offset, thereby ensuring that the system ultimately attains the desired target, even in the presence of persistent disturbances. The derivative (K_d) term forecasts potential future errors based on the rate of change, which aids in mitigating oscillations and enhancing the overall stability of the system.

3.8.2. Rate Limit Update

The bandwidth limit is updated incrementally:

$$R_{limit}(n) = R_{limit}(n-1) + u(n) \quad (7)$$

This incremental update ensures smooth transitions and prevents abrupt rate changes that could cause traffic bursts or packet loss spikes.

3.8.3. Dead Zone Implementation

To prevent oscillations caused by measurement noise and minor fluctuations, the controller implements a dead zone:

$$e_{effective}(n) = \begin{cases} 0 & \text{if } |e(n)| < \epsilon \\ e(n) & \text{otherwise} \end{cases} \quad (8)$$

where ϵ is the dead zone threshold (set to 0.5% in our implementation). This prevents unnecessary rate adjustments when CPU utilization is within acceptable bounds, improving system stability and reducing control overhead.

3.8.4. Sampling Interval Selection and Sub-Second Burst Handling

The 500 ms control period is a deliberate design choice balancing measurement reliability, control stability, and system overhead. The eBPF CPU sensor accumulates kernel-function execution times and reports one measurement per 100 ms interval; using five consecutive sensor readings per PID cycle reduces the coefficient of variation of the CPU estimate to below 5%, yielding a stable error signal. Reducing the interval to 100 ms (single-sample operation) amplifies measurement noise, which—given the existing proportional gain $K_p = 5.0$ —would drive the controller into limit-cycle oscillations without re-tuning all three gains. Conversely, intervals above one second introduce sluggish response to genuine load ramps and allow CPU usage to drift significantly above the setpoint before correction begins.

A two-layer defense mechanism handles the mismatch between the 500 ms PID cycle and sub-second traffic dynamics. At the sub-millisecond level, the XDP token bucket absorbs burst packets before they reach the OVS datapath: tokens accumulated during quiescent periods permit short-duration bursts up to B_{max} , and excess packets are dropped at the NIC without incurring forwarding CPU cost. The PID layer then corrects the resulting sustained rate trend on the 500 ms timescale. This separation of concerns means that transient CPU spikes caused by individual bursts are absorbed elastically by the token bucket, while the PID eliminates the long-term mean deviation.

We acknowledge that in high-density 5G/6G environments, where traffic variability can occur on scales of tens of milliseconds, a shorter control interval may be warranted. Reducing the period to 100–200 ms while maintaining stability would require lower proportional gain or an adaptive gain-scheduling strategy, and quantifying the optimal interval under realistic 5G traffic profiles is reserved for future work.

3.8.5. Light-Load Bias and Safety Margin

At low offered rates, the PID controller may operate in a supply-limited regime, where the admissible rate exceeds the incoming traffic, preventing utilization from reaching the target setpoint. Moreover, the per-slice CPU measurement reflects time spent in instrumented OVS/kernel functions, while fixed per-packet overheads outside the measurement scope (e.g., softirq and socket processing) dominate at low traffic rates, resulting in a conservative utilization estimate. To ensure stability, the controller configuration employs a low integral gain, a dead zone of approximately one half percent, coarse update intervals of 500 ms, and an additional safety margin in the mapping from CPU utilization to the admissible rate. These design choices mitigate oscillations and overshoot but introduce a systematic under-target bias under light-load conditions.

4. Implementation

This section describes the implementation of the proposed CPU-aware traffic control framework, mapping the theoretical architecture presented in Section 3 to corresponding software components and technologies. The complete source code for the CSRM framework is publicly available [35].

4.1. Implementation Overview

The system employs Linux network namespaces to create isolated virtual network segments (slices). Each slice contains an independent OVS instance, providing complete datapath isolation. The architecture consists of three namespace types:

- Host namespace—Contains the physical uplink interface (ens19), the Linux bridge (br-phy), and the host-side veth endpoints. The gRPC server and control components execute in this namespace.
- OVS namespaces (ovs-1, ovs-2)—Each contains an independent OVS instance with its own datapath (br0), flow tables, and forwarding logic. The OVS processes (ovs-vswitchd, ovsdb-server) run within their respective namespaces.
- Test namespaces (test-1, test-2)—Contain the endpoint applications (e.g., iperf3 servers) that generate or receive traffic. These namespaces simulate tenant workloads.

The virtual network topology implements a hierarchical bridging architecture as shown in Figure 4. The physical uplink (ens19) is connected to a Linux bridge (br-phy) in the host namespace. Each OVS instance in the OVS namespaces is connected to br-phy via a dedicated veth pair, ensuring that all traffic to/from the OVS instances passes through the host-level bridge and uplink.

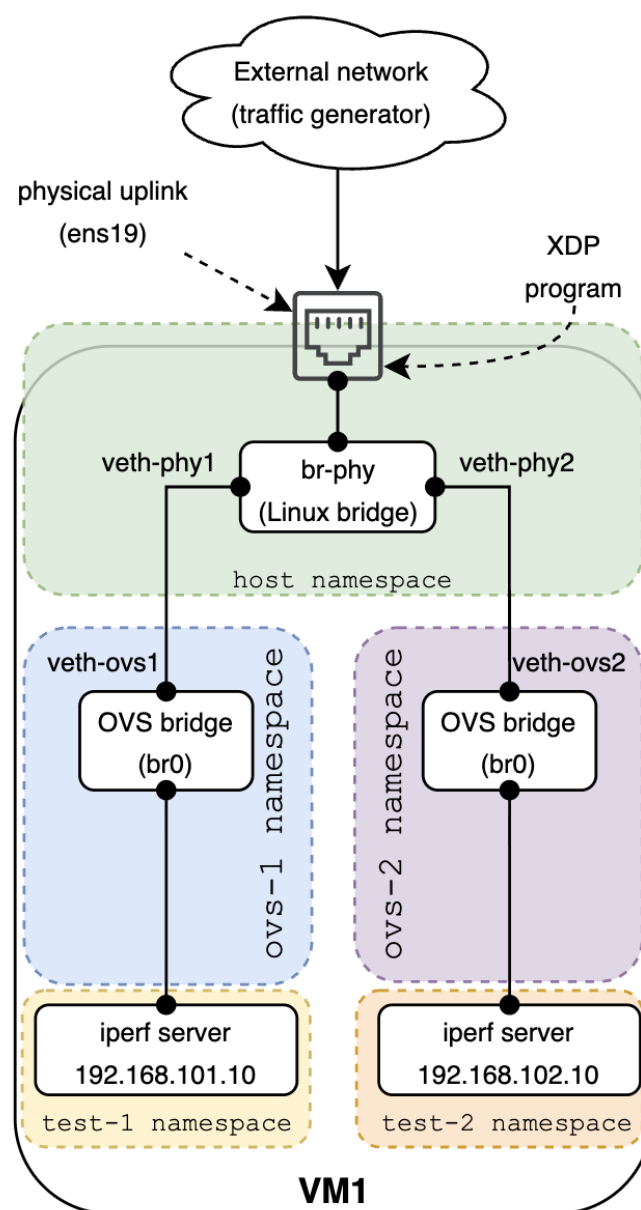


Figure 4. Implementation topology showing the OVS host with two network slices (ovs-1, ovs-2), namespace-based isolation, and external traffic generators.

4.2. Software Module Mapping

The implementation is distributed across two hosts: the Data Plane Host running OVS and the Control Host running CSRM. Table 1 lists the modules deployed on the Data Plane Host, while Table 2 lists the modules on the Control Host.

Table 1. Data Plane Host modules (OVS Host).

Component	Module	Description
Data Plane Agent	gRPC_o.py	gRPC server, XDP integration, metrics export
PID Controller	pid_controller.py	PID algorithm and bandwidth adjustment
CPU Sensor	ovs_cpu_full.bpf.c	eBPF kprobes for OVS function instrumentation
Rate Limiter	(embedded in agent)	XDP token bucket rate limiting
gRPC Interface	ovs_monitor.proto	Protocol Buffers service definition

Table 2. Control Host modules (CSRM).

Component	Module	Description
Slice Limit API	slice_limit_api.py	Client API for limit control, calibration, and monitoring
PID Controller	pid_controller.py	Local PID control loop (can run on control host)
gRPC Stubs	ovs_monitor_pb2.py	Generated Protocol Buffers stubs

Figure 5 shows how these modules implement the closed-loop control model from Figure 3, with technology-specific components.

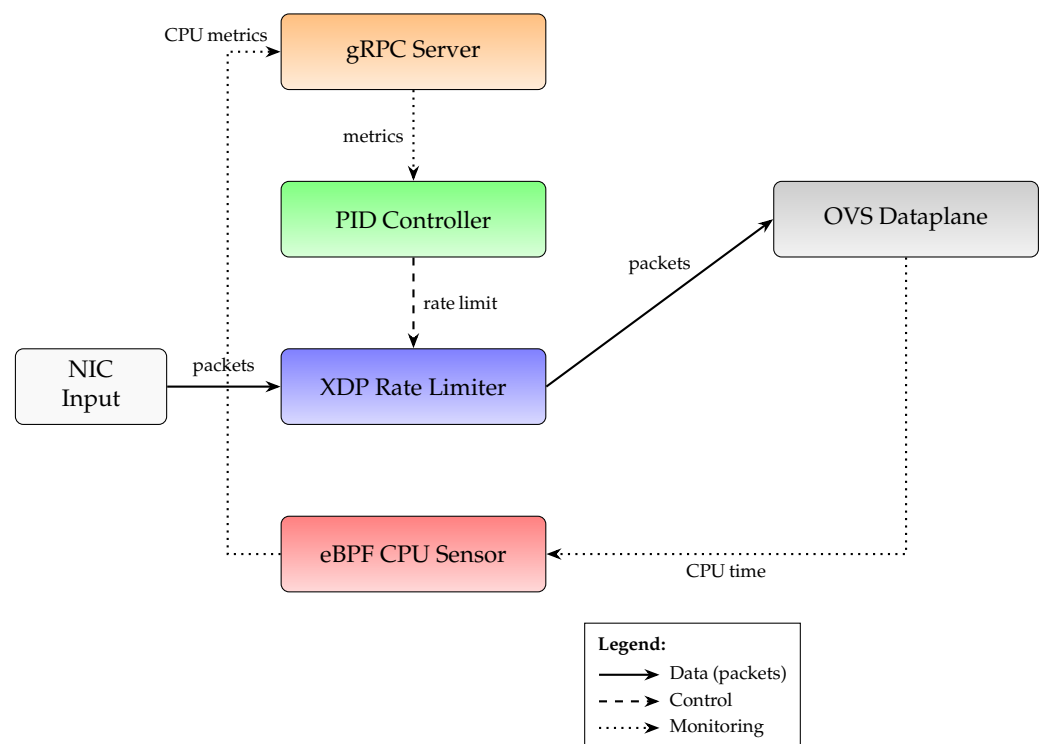


Figure 5. Software module interaction diagram showing data flow between implementation components on the Data Plane Host. Solid arrows: data (packet) flow; dashed arrows: control; dotted arrows: monitoring. Packets arrive at the NIC, are filtered by the XDP Rate Limiter, and forwarded to the OVS Dataplane. The eBPF CPU Sensor measures OVS processing time via kprobes and reports CPU metrics through the gRPC Server to the PID Controller, which adjusts the rate limit accordingly.

The control flow operates as follows: (1) the gRPC Server receives CPU metrics from the eBPF sensor and exposes them to the CSRM; (2) the PID Controller computes rate

limit adjustments based on the error between target and measured CPU; (3) the XDP Rate Limiter enforces the computed limits by filtering packets at the NIC driver level; (4) the eBPF CPU Sensor instruments OVS kernel functions via kprobes and reports CPU time back to the gRPC server. The OVS Dataplane processes packets that pass the XDP filter, with its CPU consumption measured by the eBPF probes.

4.3. eBPF-Based CPU Measurement

Extended Berkeley Packet Filter (eBPF) is a Linux kernel technology that allows running sandboxed programs in kernel space without modifying kernel source code or loading kernel modules. We use eBPF kprobes to implement the per-slice CPU measurement described in Section 3.6. Four kernel functions, described in Table 3, are instrumented to capture the complete packet processing path:

Table 3. eBPF kprobe instrumentation points.

Function	skb Param	Description
ovs_dp_process_packet	PARM2	OVS datapath processing
veth_xmit	PARM1	Virtual Ethernet transmission
netif_receive_skb	PARM1	Network interface receive
__dev_queue_xmit	PARM1	Device transmit queue

The total CPU time for a packet is the sum of all processing stages, as described in Section 3.6:

$$T_{total} = T_{ovs} + T_{veth} + T_{netif} + T_{xmit} \quad (9)$$

The CPU time measurement employs a kprobe/kretprobe pair for each instrumented function is presented in Algorithm 1.

Algorithm 1 eBPF CPU Measurement

```

procedure ONFUNCTIONENTRY(skb)
    pid_tgid ← GETCURRENTPIDTGID()
    ifindex ← skb → dev → ifindex
    start_map[pid_tgid] ← {timestamp : now(), ifindex}

procedure ONFUNCTIONEXIT()
    pid_tgid ← GETCURRENTPIDTGID()
    entry ← start_map[pid_tgid]
     $\Delta t$  ← now() − entry.timestamp
    cpu_time[entry.ifindex] ← cpu_time[entry.ifindex] +  $\Delta t$ 
    DELETE(start_map[pid_tgid])
  
```

The algorithm operates as follows. On each instrumented kernel function entry, ONFUNCTIONENTRY records the current process identifier (*pid_tgid*) and the network interface index (*ifindex*) of the socket buffer, storing them together with a nanosecond timestamp in a per-PID hash map (*start_map*). This associates the start of a kernel function execution with the specific OVS slice (identified by *ifindex*) that triggered it. On function return, ONFUNCTIONEXIT retrieves the saved entry for the same PID, computes the elapsed kernel time $\Delta t = \text{now}() - \text{entry.timestamp}$, and accumulates it in a per-interface counter (*cpu_time*[*ifindex*]). The entry is then deleted to prevent stale state from accumulating. Because each OVS slice uses a distinct network interface, the per-*ifindex* accumulation naturally partitions CPU consumption by slice without requiring any explicit tagging of packets.

The accumulated CPU time is converted to percentage:

$$CPU_{\%} = \frac{\Delta T_{ns}}{T_{interval} \times 10^9 \times C} \times 100 \quad (10)$$

where ΔT_{ns} is the delta of accumulated CPU time, $T_{interval}$ is the measurement interval (0.1 s or 100 ms), and C is the number of CPU cores.

The eBPF program stores per-function CPU time in separate hash maps, allowing fine-grained analysis. The Data Plane Agent periodically reads these maps and exports aggregated metrics via gRPC.

4.4. XDP-Based Rate Limiting

eXpress Data Path (XDP) is a high-performance packet processing framework that executes eBPF programs at the earliest point in the Linux network stack—directly in the NIC driver, before socket buffer allocation. We selected XDP over alternative rate limiting mechanisms because, in our KVM/virtio-net setup, it processes packets in $\approx 1\text{--}3\ \mu\text{s}$ (vs. $\approx 2\text{--}5\ \mu\text{s}$ for kernel-based mechanisms), dropped packets bypass the networking stack, minimizing CPU usage, and eBPF maps enable userspace to update rate limits without reloading the program.

The XDP rate limiter implements the token bucket algorithm described in Section 3.7. The program maintains per-slice state in eBPF maps (*rate_limits*, *token_state*) shown in Algorithm 2.

Algorithm 2 XDP Token Bucket Rate Limiting

```

procedure PROCESSPACKET(ctx)
  dst_ip  $\leftarrow$  EXTRACTDESTIP(ctx)
  tb  $\leftarrow$  token_state[dst_ip]
  if tb = NULL then return XDP_PASS
   $\Delta t \leftarrow \text{now}() - \text{tb.last\_update}$ 
  tb.tokens  $\leftarrow$  min(tb.tokens + tb.rate  $\times$   $\Delta t$ ,  $B_{max}$ )
  if tb.tokens  $\geq$  |packet| then
    tb.tokens  $\leftarrow$  tb.tokens − |packet|
    return XDP_PASS
  else return XDP_DROP

```

The algorithm operates as follows. The destination IP address extracted from the packet header is used as a per-slice key to look up the token bucket state (*tb*) in the eBPF map. If no entry exists for that destination (the slice is not under active rate limiting), the packet is passed unconditionally via XDP_PASS. Otherwise, tokens are replenished by computing the elapsed time Δt since the last update and adding $\text{tb.rate} \times \Delta t$ tokens, capped at the maximum burst size B_{max} to prevent unlimited accumulation. If the current token count is sufficient to cover the packet size |*packet*|, the tokens are consumed and the packet is forwarded; if not, the packet is dropped via XDP_DROP before it reaches the kernel networking stack. As discussed in Section 3.7, the XDP ingress placement ensures that dropped packets bypass the kernel networking stack, directly reducing CPU consumption in the OVS datapath.

4.5. gRPC Communication Layer

The system uses gRPC for communication between CSRM and Data Plane Agents. Alternative transport mechanisms such as REST/HTTP, WebSocket, raw sockets, or message queues (e.g., MQTT or ZeroMQ) are viable. In this work, gRPC was selected primarily due to its native support for server-side streaming, which avoids application-level polling, its efficient Protocol Buffers serialization, and its strongly typed service interface that

facilitates modular system evolution. Given our 500 ms control loop interval, any transport supporting push-based or streaming semantics would likely be sufficient to meet the timing requirements. We did not conduct a comparative evaluation of transport-layer latency and overhead; this choice reflects a design rationale rather than an empirically validated performance comparison. Raw sockets offer low latency but lack built-in serialization and require custom protocol implementation. Message queues add deployment complexity and potential single points of failure. gRPC was selected because server-side streaming enables continuous telemetry export without polling overhead, Protocol Buffers provide compact binary encoding, and the service contract defined in .proto files ensures type safety and consistency between distributed components. We do not claim that REST/HTTP is unsuitable for sub-second control loops; quantifying transport-layer effects on end-to-end detection-to-action latency is left to future work.

The gRPC interface defines three methods: `StreamMetrics` for continuous CPU metrics streaming from agent to CSRM, `SetRateLimit` for commands from CSRM to update slice rate limits, and `RunCalibration` to trigger the calibration sequence for coefficient measurement.

4.6. PID Controller Implementation

The PID controller described in Section 3.8 is implemented in `pid_controller.py`. The control loop executes every 500 ms following Algorithm 3.

Algorithm 3 PID Control Loop

```

procedure CONTROLLOOP( $CPU_{target}, \Delta t$ )
   $CPU_{measured} \leftarrow \text{READMETRICS}()$ 
   $e \leftarrow CPU_{target} - CPU_{measured}$ 
  if  $|e| < 0.5\%$  then return // dead zone
   $P \leftarrow K_p \times e$ 
   $integral \leftarrow integral + e \times \Delta t$ 
   $I \leftarrow K_i \times integral$ 
   $D \leftarrow K_d \times (e - e_{prev}) / \Delta t$ 
   $u \leftarrow P + I + D$ 
   $R_{limit} \leftarrow R_{limit} + u$ 
  UPDATERATELIMIT( $R_{limit}$ )
   $e_{prev} \leftarrow e$ 

```

The algorithm operates as follows. At each 500 ms tick, `READMETRICS` queries the gRPC server for the latest CPU utilization reported by the eBPF sensor. The error $e = CPU_{target} - CPU_{measured}$ is computed; a dead zone of $|e| < 0.5\%$ suppresses corrections when the system is already within an acceptable band, preventing limit oscillation caused by measurement noise. Outside the dead zone, the controller evaluates the three PID terms as defined in Section 3.8: $P = K_p \cdot e(n)$, $I = K_i \cdot \sum e(j)\Delta t$, $D = K_d \cdot (e(n) - e(n-1)) / \Delta t$. The controller adds the control output $u = P + I + D$ to the current rate limit R_{limit} , producing an incremental (velocity-form) update, and writes the updated limit to the XDP eBPF map via `UPDATERATELIMIT`, taking effect at the NIC for the next arriving packet. Finally, e_{prev} is saved for the derivative computation in the following cycle. The Data Plane Agent writes the new values to the XDP eBPF maps for immediate enforcement.

5. Test Environment

This section describes the experimental platform, traffic generation methodology, and evaluation scenarios used to validate the proposed framework.

5.1. Hardware and Software Platform

The experimental platform is deployed on a virtualized infrastructure using Proxmox VE with KVM hypervisor. Table 4 summarizes the hardware and software specifications.

Table 4. Experimental platform specifications.

Component	Specification
Hypervisor	Proxmox VE 8.x (KVM)
vCPU	2 cores (Intel Xeon, nested virtualisation)
RAM	4 GB DDR4
Network Interface	virtio-net driver
Uplink Interface	virtio-net paravirtualised NIC (configured to avoid link-layer bottleneck)
Operating System	Ubuntu 22.04 LTS (kernel 5.15+)
OVS Version	2.17
BPF Toolchain	BCC (BPF Compiler Collection) 0.24+

Although the virtual uplink interface was provisioned with sufficient capacity to avoid becoming the primary bottleneck, the evaluated KVM/virtio-net datapath saturated near 1.4 Gbps aggregate throughput due to software packet-processing limitations. The dominant bottlenecks are virtualization overheads—particularly virtio-net virtqueue depth/parallelism and KVM interrupt/VM-exit handling—rather than the line rate of the underlying host. Accordingly, the reported results should be interpreted as comparative measurements within a constrained virtualized environment, not as absolute hardware throughput limits. Both XDP and OVS policing were evaluated under identical conditions, preserving the internal validity of the comparative analysis; on bare-metal or with enhanced I/O (e.g., more queues/RSS or SR-IOV), the absolute ceiling would be higher, which we leave for future work. The virtualized infrastructure reflects real-world cloud deployment scenarios where network functions are commonly deployed as virtual machines. The virtio-net driver provides paravirtualized network I/O with near-native performance.

The 2-core configuration with 2 RSS (Receive Side Scaling) queues represents a resource-constrained environment typical of edge computing deployments. This configuration amplifies the effects of CPU contention, making it suitable for evaluating the effectiveness of CPU-aware traffic control.

5.2. Network Topology

The testbed consists of a central OVS node (VM1) that hosts two isolated network slices, each connected to a dedicated external traffic generator. VM1 runs both the ovs-1 and ovs-2 slices and is connected to the physical network via the ens19 uplink through the br-phy bridge. An external Traffic Generator 1 sends packets to the ovs-1 slice at the IP address 192.168.101.10. Traffic Generator 2 sends packets to the ovs-2 slice at the IP address 192.168.102.10. This setup allows both slices to receive separate traffic streams simultaneously, simulating multiple users.

5.3. Traffic Generation Methodology

Traffic is generated using a custom Python (v. 3.10.12) script (`traffic_generator.py`) [35] that wraps `iperf3` with ramp-up capabilities. The script supports configurable ramp-up where traffic rate increases in defined steps (e.g., 100 Mbps increments), step duration control where each rate level is maintained for a configurable period (e.g., 10 s), synchronized start allowing multiple generators to be coordinated for concurrent slice loading, and statistics collection providing per-step throughput, packet loss, and jitter measurements. Table 5 lists the traffic generation parameters used in experiments.

Table 5. Traffic generation parameters.

Parameter	Value
Traffic generator	iperf3 3.9 (UDP mode)
Packet size	1400 bytes
Ramp-up start	100 Mbps per slice
Ramp-up step	100 Mbps
Step duration	10 s
Maximum rate	1500 Mbps per slice

The choice of UDP traffic eliminates TCP congestion control effects, providing direct observation of the rate-limiting mechanism. The 1400-byte packet size represents typical application payloads while avoiding fragmentation.

5.4. Evaluation Scenarios

Three evaluation scenarios are designed to characterize system behavior across different operating conditions:

5.4.1. Scenario 1—Baseline Performance (No Rate Limiting)

This scenario establishes the system's natural capacity without any rate limiting intervention. Table 6 summarizes the scenario parameters.

Table 6. Scenario 1: Baseline Performance.

Parameter	Description
Objective	Determine maximum sustainable throughput and CPU saturation point
Configuration	XDP rate limiting disabled; traffic ramps from 100 to 1500 Mbps per slice
Metrics	Total CPU utilization, per-slice OVS CPU, packet delivery ratio
Expected outcome	Identification of the linear CPU-throughput region and saturation threshold

This scenario provides the empirical data needed to calculate the CPU-throughput coefficient k and establish operating zone boundaries.

5.4.2. Scenario 2—Low-Load Controlled Operation

This scenario evaluates the PID controller's effectiveness under moderate traffic conditions where the system operates within its stable region. Table 7 summarizes the scenario parameters.

This scenario demonstrates the system's ability to maintain predictable resource allocation when operating below saturation.

Table 7. Scenario 2: Experimental Setup and Evaluation Metrics.

Parameter	Description
Objective	Validate closed-loop CPU regulation and slice isolation
Configuration	CPU targets of 2% and 3% on OVS-1 (OVS-2 unlimited); traffic up to 500 Mbps per slice
Metrics	CPU convergence, packet loss comparison, cross-slice interference
Expected outcome	Effective CPU isolation with limited slice showing higher packet loss

5.4.3. Scenario 3—High-Load Stress Test (Worst Case)

This scenario pushes the system beyond its stable operating region to characterize failure modes and limitations. Table 8 summarizes the scenario parameters.

Table 8. Scenario 3: High-Load Stress Test.

Parameter	Description
Objective	Validate slice isolation under extreme overload conditions
Configuration	5% CPU target on OVS-1 (OVS-2 unlimited); traffic up to 1500 Mbps per slice (3000 Mbps aggregate)
Metrics	Packet loss comparison, XDP drop counts, CPU isolation effectiveness
Expected outcome	Limited slice shows higher packet loss while unlimited slice is not negatively affected

This worst-case scenario reveals the fundamental limitations of software-based rate limiting when kernel resources become the bottleneck.

5.5. Measurement and Data Collection

Three categories of metrics are collected during each experiment. CPU metrics are gathered via eBPF at 100ms intervals, including per-slice CPU time in nanoseconds, total system CPU utilization as a percentage, and per-function CPU breakdown. Traffic metrics are collected via iperf3 at 1-second intervals, capturing achieved throughput in Mbps, packet loss rate as a percentage, and jitter in milliseconds. Control metrics are logged by CSRM at 500ms intervals, recording the PID error signal, computed rate limit adjustments, and actual enforced rate limits.

All metrics are timestamped and logged to CSV files for post-experiment analysis. The variability observed across five independent runs is illustrated by shaded mean $\pm$ standard deviation regions in Section 6. The measurement infrastructure introduces less than 1% additional CPU overhead, as validated through comparison with and without instrumentation.

5.6. PID Controller Parameters

Table 9 lists the PID controller parameters used across all experiments.

Table 9. Parameters of the Proposed PID Controller.

Parameter	Value
Proportional gain (K_p)	5.0
Integral gain (K_i)	0.1
Derivative gain (K_d)	1.0
Sampling interval	0.5 s
Dead zone (ϵ)	$\pm 0.5\%$
Token bucket burst size	65,536 bytes (64 KB)

These parameters were derived through an empirical three-stage procedure on the live testbed, validated by the sensitivity study and stability analysis described below.

5.6.1. Tuning Procedure

Gains were determined in three sequential stages on the live testbed. In the first stage, with $K_i = K_d = 0$, K_p was increased from 1.0 until the first CPU-limit violation appeared; the last compliant value identified the available proportional headroom. In the second

stage, K_i was raised incrementally until the steady-state offset—CPU remaining well below the 3% setpoint during sustained high traffic—was eliminated without inducing oscillation. In the third stage, K_d was tuned to damp overshoot observed on step-up traffic transitions spanning 100 to 500 Mbps ramps.

The Ziegler–Nichols open-loop method was considered but not applied directly, as the plant (CPU usage as a function of packet rate) is nonlinear and time-varying; empirical step-response tuning on the actual testbed was therefore more appropriate. The resulting nominal gains lie at the empirical knee of the responsiveness–stability curve, as confirmed by the sensitivity study below.

5.6.2. Sensitivity Analysis

Each variant was tested with five independent runs under identical synthetic traffic (step increases: 100, 200, 300, 400, 500 Mbps $\times$ 10 s each, constant 1400-byte frames). The OVS-1 CPU limit is 3.0%. The raw per-run CSV logs for all variants are available in the project repository [35]. As shown in Table 10, the nominal PID configuration achieved stable CPU regulation without overshoot.

Table 10. PID gain sensitivity: aggregate metrics (5 runs per variant; OVS-1 CPU limit = 3.0%).

Variant	K_p	K_i	K_d	CPU Mean (%)	CPU Max (%)	Viol. (>3%)	Overshoot (pp)	XDP 400 (Mbps)	XDP 500 (Mbps)
−50%	2.50	0.050	0.50	0.813	3.652	0.4%	+0.65	7.89	37.41
−25%	3.75	0.075	0.75	0.745	2.782	0.0%	—	22.20	57.00
Nominal	5.00	0.100	1.00	0.750	2.077	0.0%	—	20.29	55.46
+25%	6.25	0.125	1.25	0.895	2.285	0.0%	—	4.58	31.79
+50%	7.50	0.150	1.50	0.772	2.607	0.0%	—	7.62	33.63

CPU mean/max computed over active-traffic samples only (NIC rx > 1 Mbps). Violations: % of 1-s samples with `ovs-1 cpu` > 3.0%. XDP 400/500 Mbps: mean Mbps throttled by the XDP program at the respective load step. Table 11 presents per-burst CPU mean (%) averaged over five runs.

Table 11. Per-burst CPU mean (%) under gain variation (5 runs per variant).

Variant	100 Mbps	200 Mbps	300 Mbps	400 Mbps	500 Mbps
−50%	0.365	0.626	0.874	1.068	1.045
−25%	0.388	0.665	0.923	0.909	0.833
Nominal	0.473	0.702	0.831	0.923	0.826
+25%	0.439	0.648	0.900	1.127	1.217
+50%	0.371	0.689	0.827	0.922	0.988

5.6.3. Stability Argument

The empirical results constitute an implicit BIBO (Bounded-Input, Bounded-Output) stability demonstration. Under $\pm 25\%$ gain variation the controller remains fully compliant, recording zero violations with CPU max $\leq 2.78\%$, and the -25% variant deviates by only +1.54 Mbps from nominal at 500 Mbps, while higher-gain variants (+25%, +50%) show reduced steady-state throttling (~ 32 – 34 Mbps vs. 55 Mbps nominal at 500 Mbps), reflecting more aggressive initial control action that holds CPU further below the setpoint, indicating a robust gain margin. Under -50% gain variation a single transient violation occurs—a peak overshoot of +0.65 pp above the setpoint, affecting 0.4% of samples—which confirms a finite gain-margin boundary. Under +50% gain variation no oscillation or instability was

observed, with CPU stdev remaining at 0.385 compared to the nominal 0.383, indicating adequate phase margin.

Formal linearization is infeasible because the plant gain ($\Delta\text{CPU}/\Delta\text{packet-rate}$) is traffic-pattern-dependent; the empirical 5-run replication (coefficient of variation $<5\%$ in CPU max across runs for all compliant variants) confirms reproducibility. The nominal gains therefore represent a conservative, experimentally validated operating point that maintains compliance across the full $\pm 25\%$ gain envelope, with a clear safety margin against the -50% degradation boundary.

Setpoint portability. Setpoints are specified on a total-host basis. To preserve the same absolute CPU budget when changing core count, the setpoint should be scaled inversely with N_{cores} , i.e., $\text{target}_{\text{new}} = \text{target}_{\text{old}} \times (N_{\text{cores,old}} / N_{\text{cores,new}})$. For example, a 3% setpoint on a 2-core host corresponds to approximately 1.5% on a 4-core host.

5.7. Token Bucket Decisions

The token bucket burst size ($B_{\text{max}} = 64 \text{ KB}$) was selected as a pragmatic engineering compromise between transient burst accommodation, control stability, throughput efficiency, and bounded latency effects within the evaluated virtualized OVS/XDP dataplane. Moderate burst capacities are commonly employed in practical token bucket and Linux traffic shaping deployments to balance short-term burst tolerance against queuing delay and jitter rather than maximize peak burst throughput [RFC 2697; RFC 2698]. In this context, the selected 64 KB configuration allows short-lived microbursts to pass without immediate packet drops, thereby improving throughput stability and preventing excessively aggressive controller behavior during transient traffic fluctuations. At the same time, substantially larger burst capacities would permit longer bursts to accumulate before enforcement, potentially increasing transient queuing delay, jitter, and short-term latency variation—effects that may be undesirable for latency-sensitive 5G/6G network slices such as V2X communication, industrial control, or real-time smart city services. Conversely, excessively small burst capacities would impose overly strict shaping, increasing unnecessary packet loss and reducing throughput efficiency even for benign short-duration traffic bursts. Accordingly, the selected 64 KB value represents a balanced trade-off appropriate for the evaluated CPU-constrained software dataplane, prioritizing stable CPU regulation while maintaining bounded burst-induced delay. Although the present work does not include a dedicated sensitivity analysis of latency or jitter across alternative burst-size configurations, future work will investigate adaptive or slice-specific burst sizing strategies for SLA-sensitive smart city and 5G/6G deployments.

6. Evaluation and Discussion

This section presents the experimental evaluation of the proposed CPU-aware traffic control framework across the three scenarios defined in Section 5.

6.1. Scenario 1—Baseline Performance

The baseline test established system capacity without rate limiting, identifying the CPU-throughput relationship and saturation thresholds. Figure 6 shows the relationship between aggregate traffic and CPU utilization during the baseline ramp-up test. The total CPU increased from 17% at 200 Mbps to approximately 42% at 1 Gbps (Safe zone), remains near this level through 1.2 Gbps (41.6–41.9%), then rises to 50% at 1.4 Gbps (Caution zone), and continues to 70% at 3 Gbps (Danger zone). The 0.8 Gbps point is absent from the figures due to insufficient stable-state samples at that ramp step. The OVS CPU per-slice line remains relatively flat (2–10%), indicating that the OVS datapath itself consumes only a

small fraction of total CPU. All CPU percentages reported in this paper are normalized to the total host capacity (N_{cores}). For the 2-vCPU testbed, 100% denotes two fully utilized vCPUs.

Figure 7 provides a detailed view of the per-slice OVS CPU consumption. Both OVS-1 and OVS-2 showed nearly identical CPU usage, ranging from 2% at 200 Mbps to 10% at 3 Gbps aggregate. This confirmed fair resource sharing between slices in the baseline scenario without rate limiting.

The significant gap between Total System CPU (17–70%) and OVS CPU per-slice (2–10%) reflects the layered nature of packet processing in virtualized environments. The combined OVS datapath (both slices) accounts for approximately 20–29% of total CPU consumption, while the remaining 70–80% is consumed by two components: (1) kernel networking overhead including the shared network stack, bridge processing, veth pair traversal, and interrupt handling, and (2) the iperf3 server processes running inside the namespaces that receive and process the test traffic. This overhead is not directly attributable to any single slice because it occurs in shared kernel code paths and application-level processing before and after packets reach slice-specific OVS processing.

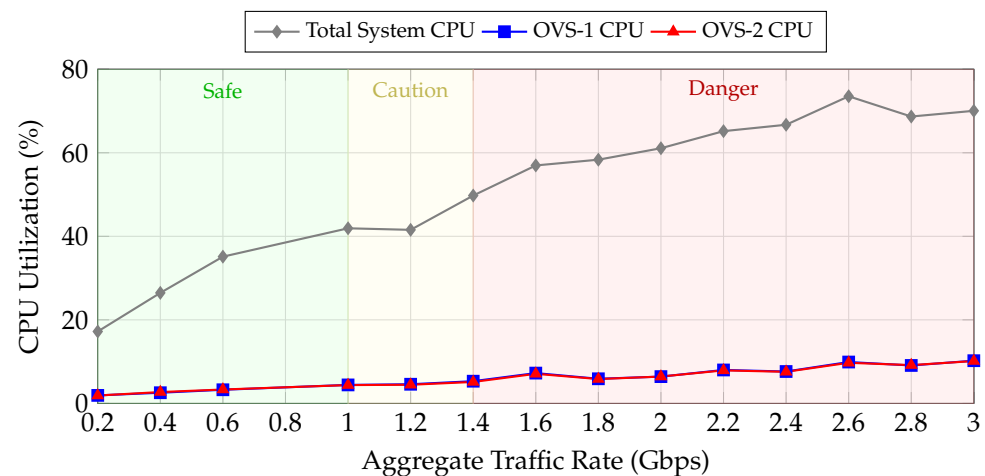


Figure 6. Total system CPU as a function of aggregate traffic during baseline test. CPU rose from 17% at 200 Mbps to 70% at 3 Gbps aggregate. CPU% values are normalized to total host capacity across 2 cores.

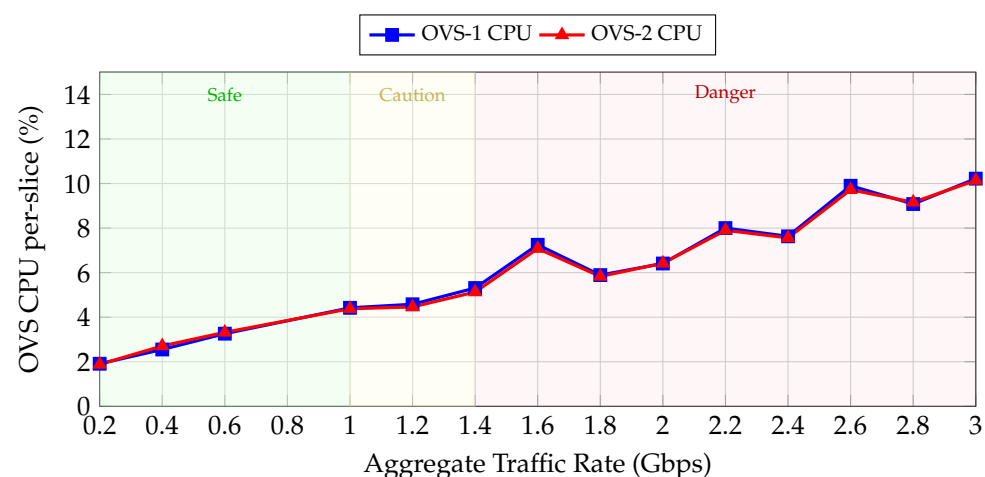


Figure 7. Per-slice OVS CPU utilization (detailed view). Both slices show nearly identical CPU consumption (2–10%), demonstrating fair resource sharing. CPU% values are normalized to total host capacity across 2 cores.

The experimental results revealed three characteristic operating zones with respect to traffic load and packet loss behaviour, as presented in Table 12.

Table 12. Operating zones for baseline performance without rate limiting (pps calculated for 1400 B packets).

Zone	Traffic (Aggregate)	Per-Slice	pps (Aggregate)	Packet Loss
Safe	<1000 Mbps	<500 Mbps	<89 K	<1%
Caution	1000–1400 Mbps	500–700 Mbps	89–125 K	1–5%
Danger	>1400 Mbps	>700 Mbps	>125 K	>30%

Key thresholds: zero-loss throughput up to 600 Mbps aggregate (300 Mbps/slice); loss <1% up to 1000 Mbps aggregate; saturation onset at 1200 Mbps (>2% loss); severe saturation at 1600 Mbps (≈35% loss). pps calculated for 1400-byte frames.

6.2. Scenario 2—Low-Load Controlled Operation

This scenario validated the PID controller’s effectiveness under controlled conditions. Based on Scenario 1 findings, we limited the maximum input traffic to 500 Mbps per slice (1000 Mbps aggregate), operating at the upper boundary of the Safe zone, avoiding the Caution and Danger zones where kernel-level saturation dominates. We first established baseline performance without CPU limiting, then applied 3% and 2% CPU targets to OVS-1 while leaving OVS-2 unlimited.

Table 13 presents the complete results showing CPU utilization and packet loss for all three configurations.

Table 13. Low-load test results ($N = 5$ independent runs): CPU utilization and XDP-enforced drop rate for baseline, 3% limit, and 2% limit configurations. The PID controller successfully maintains OVS-1 CPU below both targets across all runs; OVS-2 (unlimited) has negligible XDP drops. In the 100–300 Mbps range, the limited slice operates in a supply-limited regime, where the offered load remained below the admissible rate computed by the controller. As a result, the measured per-slice OVS CPU stays below the configured setpoint without requiring substantial active throttling.

Mbps	Baseline		3% Limit		2% Limit	
	OVS-1	OVS-2	OVS-1	OVS-2	OVS-1	OVS-2
CPU Utilization (%), mean $\pm$ std ($N = 5$ runs)						
100	1.47 $\pm$ 0.56	1.60 $\pm$ 0.56	0.47 $\pm$ 0.14	0.40 $\pm$ 0.07	0.47 $\pm$ 0.08	0.40 $\pm$ 0.08
200	1.64 $\pm$ 0.04	1.74 $\pm$ 0.15	0.63 $\pm$ 0.04	0.60 $\pm$ 0.06	0.76 $\pm$ 0.21	0.71 $\pm$ 0.13
300	2.42 $\pm$ 0.19	2.43 $\pm$ 0.25	0.88 $\pm$ 0.02	0.84 $\pm$ 0.05	0.77 $\pm$ 0.21	0.94 $\pm$ 0.16
400	3.03 $\pm$ 0.18	3.08 $\pm$ 0.26	0.93 $\pm$ 0.07	1.13 $\pm$ 0.06	0.73 $\pm$ 0.19	1.23 $\pm$ 0.16
500	3.53 $\pm$ 0.22	3.58 $\pm$ 0.29	1.02 $\pm$ 0.11	1.43 $\pm$ 0.12	0.60 $\pm$ 0.12	1.34 $\pm$ 0.10
XDP-Enforced Drop Rate (%), mean $\pm$ std ($N = 5$ runs)						
100	0.00 $\pm$ 0.00	—	0.00 $\pm$ 0.00	0.00 $\pm$ 0.00	0.00 $\pm$ 0.00	0.00 $\pm$ 0.00
200	0.00 $\pm$ 0.00	—	0.00 $\pm$ 0.00	0.00 $\pm$ 0.00	1.15 $\pm$ 1.46	0.00 $\pm$ 0.00
300	0.00 $\pm$ 0.00	—	2.09 $\pm$ 2.26	0.00 $\pm$ 0.00	33.58 $\pm$ 4.76	0.00 $\pm$ 0.00
400	0.00 $\pm$ 0.00	—	22.14 $\pm$ 3.25	0.01 $\pm$ 0.00	53.15 $\pm$ 1.42	0.01 $\pm$ 0.01
500	0.00 $\pm$ 0.00	—	40.94 $\pm$ 5.15	0.01 $\pm$ 0.01	64.01 $\pm$ 0.74	0.01 $\pm$ 0.01

Figure 8 visualizes the packet loss comparison between configurations, with shaded mean $\pm$ standard deviation regions indicating measurement dispersion across all five independent runs.

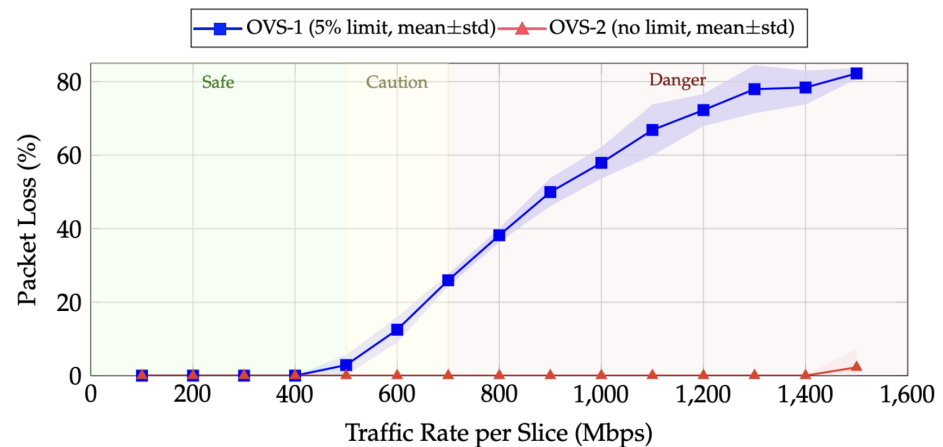


Figure 8. XDP-enforced drop rate under low load across $N = 5$ independent runs. Solid/dashed curves show mean packet-loss values for 3% and 2% CPU targets, while shaded regions indicate mean $\pm$ standard deviation. OVS-1 (limited slice) experiences progressively stronger XDP enforcement as CPU targets tighten, whereas OVS-2 (unlimited slice) maintains negligible packet loss ($<0.02\%$), confirming effective slice isolation in the safe operating zone.

The results demonstrated three key findings across all five runs. First, the PID controller successfully maintained OVS-1 CPU well below both targets in every run: mean $1.02 \pm 0.11\%$ vs. the 3% target and mean $0.60 \pm 0.12\%$ vs. the 2% target at 500 Mbps. The consistently low standard deviations confirmed stable controller behavior across repeated experiments. The controller is intentionally conservative, preferring to enforce XDP drops rather than risk violating CPU constraints.

Second, the unlimited slice (OVS-2) experienced negligible XDP-enforced drops ($<0.02\%$) regardless of the limit applied to OVS-1, confirming robust slice isolation. The near-zero variance bands for OVS-2 across all traffic levels further validate that rate limiting on one slice does not introduce measurable cross-slice interference under Safe-zone conditions.

Third, more aggressive limiting (2% vs. 3%) provides stronger CPU isolation ($0.60 \pm 0.12\%$ vs. $1.02 \pm 0.11\%$) but at the cost of substantially higher XDP drop rates ($64.01 \pm 0.74\%$ vs. $40.94 \pm 5.15\%$ at 500 Mbps). The widening variance bands for OVS-1 at higher throughput reflect increasing controller intervention as traffic approaches the configured CPU ceiling.

Overall, these results confirmed that XDP-based PID-controlled rate limiting successfully enforced predictable CPU isolation within the Safe zone while preserving near-perfect service quality for unlimited slices. Across all five runs, the unlimited slice remains effectively unaffected, demonstrating that the proposed framework achieves strong per-slice isolation without inducing kernel-level contention under controlled traffic conditions.

6.3. Scenario 3—High-Load Stress Test

This scenario validated slice isolation under extreme load conditions where total traffic exceeded system capacity. We applied a 5% CPU limit to OVS-1 while leaving OVS-2 unlimited, then ramp both slices simultaneously from 100 to 1500 Mbps (3000 Mbps aggregate) using UDP traffic. Table 14 presents the complete results.

Figure 9 visualizes the packet loss comparison between the limited and unlimited slices, with shaded mean $\pm$ standard deviation regions representing measurement dispersion across all five independent runs.

Table 14. High-load stress test with 5% CPU limit on OVS-1 ($N = 5$ independent runs). OVS-1 CPU stays below the 5% target across all runs; XDP enforcement begins at 500 Mbps and grows monotonically to $315,690 \pm 48,471$ drops/interval at 1500 Mbps. Under saturation (>700 Mbps), shared kernel overhead dominates and isolation degrades.

Traffic (Mbps)	CPU (%), Mean $\pm$ Std		XDP Drop Rate (%), Mean $\pm$ Std		XDP Drops Mean $\pm$ Std
	OVS-1	OVS-2	OVS-1	OVS-2	
N = 5 independent runs					
100	0.44 $\pm$ 0.09	0.41 $\pm$ 0.10	0.00 $\pm$ 0.00	0.00 $\pm$ 0.00	0 $\pm$ 0
200	0.69 $\pm$ 0.11	0.64 $\pm$ 0.08	0.00 $\pm$ 0.00	0.00 $\pm$ 0.00	0 $\pm$ 0
300	1.00 $\pm$ 0.19	0.92 $\pm$ 0.21	0.00 $\pm$ 0.00	0.00 $\pm$ 0.00	0 $\pm$ 0
400	1.14 $\pm$ 0.13	1.10 $\pm$ 0.15	0.00 $\pm$ 0.00	0.00 $\pm$ 0.00	0 $\pm$ 0
500	1.33 $\pm$ 0.13	1.27 $\pm$ 0.22	2.87 $\pm$ 2.76	0.00 $\pm$ 0.00	3208 $\pm$ 3052
600	1.55 $\pm$ 0.18	1.56 $\pm$ 0.11	12.50 $\pm$ 3.44	0.00 $\pm$ 0.00	16,691 $\pm$ 3949
700	1.36 $\pm$ 0.10	1.70 $\pm$ 0.12	25.95 $\pm$ 1.73	0.00 $\pm$ 0.01	45,750 $\pm$ 6100
800	1.41 $\pm$ 0.16	1.93 $\pm$ 0.21	38.20 $\pm$ 1.89	0.00 $\pm$ 0.00	69,925 $\pm$ 20,255
900	1.19 $\pm$ 0.16	2.10 $\pm$ 0.20	49.95 $\pm$ 3.81	0.01 $\pm$ 0.01	103,032 $\pm$ 10,908
1000	1.15 $\pm$ 0.06	2.46 $\pm$ 0.18	57.91 $\pm$ 4.29	0.01 $\pm$ 0.01	126,634 $\pm$ 13,562
1100	1.09 $\pm$ 0.16	2.83 $\pm$ 0.28	66.85 $\pm$ 6.94	0.02 $\pm$ 0.02	154,909 $\pm$ 27,852
1200	0.99 $\pm$ 0.08	3.24 $\pm$ 0.42	72.26 $\pm$ 4.38	0.01 $\pm$ 0.01	198,773 $\pm$ 23,010
1300	1.70 $\pm$ 1.66	3.71 $\pm$ 0.75	77.96 $\pm$ 6.54	0.01 $\pm$ 0.01	218,564 $\pm$ 21,045
1400	1.37 $\pm$ 0.45	4.56 $\pm$ 1.31	78.40 $\pm$ 4.61	0.01 $\pm$ 0.02	306,502 $\pm$ 62,651
1500	1.34 $\pm$ 0.81	4.73 $\pm$ 1.37	82.24 $\pm$ 1.46	2.26 $\pm$ 5.03	315,690 $\pm$ 48,471

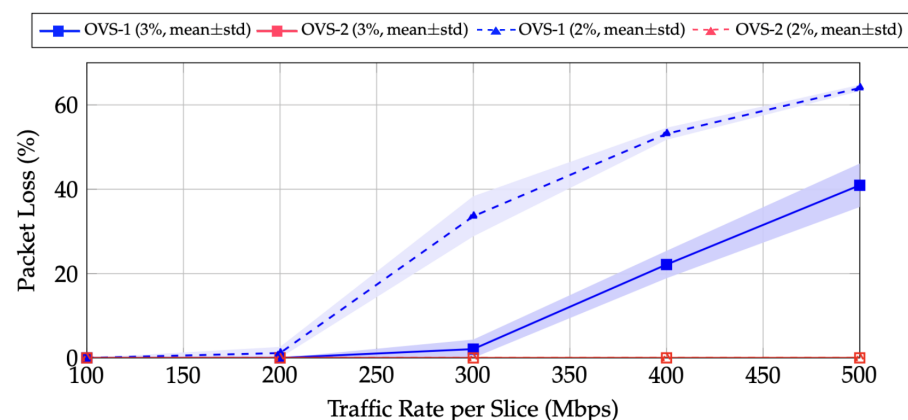


Figure 9. Packet loss under high load across $N = 5$ independent runs. Solid lines show mean packet-loss values, while shaded regions represent mean $\pm$ standard deviation. In the saturated regime, isolation degrades: XDP enforces substantial drops on the limited slice, while the unlimited slice also experiences loss due to shared kernel bottlenecks.

The results across all five runs demonstrated three key findings. First, the controller maintained OVS-1 CPU below the 5% target in every run: mean $1.33 \pm 0.13\%$ at 500 Mbps and $1.34 \pm 0.81\%$ at 1500 Mbps; OVS-2 reached $4.73 \pm 1.37\%$ at 1500 Mbps. Second, XDP enforcement began at 500–600 Mbps (3208 ± 3052 drops/interval at 500 Mbps) and grew monotonically to $315,690 \pm 48,471$ at 1500 Mbps; the widening shaded variance bands at high traffic rates confirmed increasing run-to-run dispersion as kernel saturation emerged. Third, XDP drops on OVS-1 began at 500 Mbps, coinciding with the transition into the Caution zone; below this threshold all five runs operated without active rate limiting.

Overall, under saturation (>700 Mbps per slice), OVS-1 experiences severe XDP-enforced drops (25.95% at 700 Mbps, rising to 82.24% at 1500 Mbps), while OVS-2 remained largely protected until the most extreme load (2.26% at 1500 Mbps with high variance),

confirming that shared kernel bottlenecks begin to dominate at extreme throughput and that XDP enforcement alone cannot guarantee perfect isolation at line rate.

6.4. Discussion

The experimental results demonstrated that CPU-aware traffic control via XDP was effective in a multi-tenant OVS dataplane environment when operating within system capacity. In the Safe zone (below 1000 Mbps aggregate, or 500 Mbps per slice), the baseline (no rate limiting) operates without any XDP-enforced drops, and the closed-loop approach adapted to varying traffic patterns automatically across all five independent runs. When CPU limits were applied, the controller enforced XDP drops to keep per-slice CPU below the target: with a 3% CPU target, OVS-1 maintained a mean CPU of $1.02 \pm 0.11\%$ (vs. 3% target) at the cost of $40.94 \pm 5.15\%$ XDP-enforced drop rate at 500 Mbps, while the unlimited OVS-2 slice experienced negligible drops ($<0.02\%$).

The primary limitation is cross-slice interference under saturation. In the Danger zone (above 1400 Mbps aggregate), kernel-level packet drops occur before XDP processing, causing both slices to experience severe packet loss ($>30\%$) regardless of XDP configuration. The root cause is the shared kernel networking stack, which accounts for approximately 70–80% of total CPU consumption, while the combined OVS datapath accounts for only 20–29%.

The per-function eBPF telemetry collected during Scenario 3 indicates that the dominant contributors to this shared overhead are (1) `NET_RX_SOFTIRQ` and `NET_TX_SOFTIRQ` deferred interrupt processing, which scales super-linearly with packet rate due to per-packet `skb` allocation and freeing; (2) Linux bridge forwarding (`br_forward/br_nf_pre_routing`) in the host-level `br-phy` bridge shared by all slices; and (3) `virtio-net` driver interrupt handling inherent to the KVM/`virtio-net` setup, which cannot be attributed to a single slice. These subsystems are architecturally shared and lie outside the scope of the per-slice XDP filter.

Several kernel-level tuning strategies could partially mitigate this bottleneck. Receive Packet Steering (RPS) and Receive Flow Steering (RFS) distribute `softirq` processing across all available vCPUs rather than concentrating it on a single core, which can defer saturation to a higher aggregate rate. CPU affinity pinning (`isolcpus, taskset`) can dedicate one vCPU exclusively to `softirq` and bridge processing, isolating this cost from OVS vCPUs. Interrupt coalescing (`ethtool-C`) reduces the per-packet interrupt rate at the `virtio-net` interface. For workloads that must operate reliably beyond 1400 Mbps aggregate, a kernel-bypass approach such as `AF_XDP` zero-copy mode or DPDK-based forwarding eliminates the `softirq` and `skb` overhead, though at the cost of a substantial implementation change. Investigating these strategies under realistic 5G traffic profiles is left as future work.

Based on our findings, we recommend proactive capacity planning by keeping aggregate traffic below 1000 Mbps (Safe zone), using 3% CPU target per-slice for a balance between isolation and throughput (or 2% for stricter isolation at the cost of higher packet loss), and considering hardware offload via SmartNIC-based rate limiting for strict isolation requirements when operating in the Danger zone (above 1400 Mbps aggregate).

A detailed comparison with related work is provided in Appendix A. Table A1 summarizes the key differences between our approach and existing solutions in terms of monitoring methodology and actuation capabilities. Table A2 presents quantitative metrics including detection-to-action time, CPU overhead, and isolation effectiveness.

Impact on Latency-Sensitive Slices and Mitigations

The under-target behavior observed in Table 13 is attributed to three main factors: a supply-limited operating regime at low offered load, conservative per-slice CPU accounting that underestimates OVS datapath activity due to dominant fixed kernel costs, and stability-oriented PID tuning, including low integral gain, a dead zone, coarse update intervals,

and an additional safety margin. While these design choices improve stability and prevent oscillations, they reduce effective utilization headroom for latency-sensitive slices during transient conditions. To mitigate this effect, an enhanced performance profile can be applied, incorporating feed-forward initialization of the admissible rate from offline calibration, tighter control parameters with shorter update intervals and reduced dead zone, adjusted integral gain with anti-windup, and the use of a minimum committed rate to guarantee baseline bandwidth.

7. Conclusions

This paper presented a closed-loop, CPU-aware traffic control framework for SDN dataplanes built on OVS. The system integrates eBPF-based CPU telemetry, a PID feedback controller for adaptive bandwidth regulation, and XDP-based early packet filtering to enable real-time traffic management in software-defined switching environments.

The framework architecture separates concerns between the Data Plane Agent (running on OVS hosts) and the Central Slice Resource Manager (CSRM), connected via gRPC for efficient bidirectional communication. This design enables local enforcement with centralized intelligence, supporting scalable multi-host deployments.

Experimental evaluation in a multi-tenant OVS dataplane environment demonstrated several key results. The PID controller effectively maintained CPU utilization below configured targets, with the 3% CPU target providing a good balance between isolation and throughput (OVS-1 maintaining $1.02 \pm 0.11\%$ CPU at the cost of $40.94 \pm 5.15\%$ XDP-enforced drop rate at 500 Mbps). Within the Safe operating zone (below 1000 Mbps aggregate, or 500 Mbps per slice), baseline kernel-level packet loss remained below 1%, and the unlimited slice (OVS-2) experienced negligible XDP-enforced drops ($<0.02\%$) regardless of the limit applied to OVS-1.

The study also identified fundamental limitations of software-based rate limiting. Effective isolation between slices was observed only within the Safe zone (below 1 Gbps aggregate), as discussed in Section 6.4. As detailed in Section 6.4, shared kernel overhead (70–80% of total CPU) dominated over OVS datapath consumption, defeating XDP-based rate limiting under saturation. Practical deployment should remain within the Safe zone and consider SmartNIC offload for strict isolation requirements.

The proposed framework is applicable to smart city network infrastructure, where multi-tenant SDN environments must provide fair resource sharing among diverse city services while maintaining cost efficiency through commodity-hardware deployment.

Future work could be directed in several ways, including multi-queue NIC support via RSS for improved slice isolation, machine-learning-based adaptive PID tuning for varying workloads, hybrid hardware–software control integration with SmartNIC offload, and IPv6 support for dual-stack deployments.

Author Contributions: Conceptualization, S.B., P.D.B. and Ž.B.; methodology, S.B.; software, S.B.; validation, S.B., P.D.B. and I.D.; formal analysis, S.B. and I.D.; investigation, S.B.; resources, Ž.B.; data curation, S.B.; writing—original draft preparation, S.B.; writing—review and editing, Ž.B., P.D.B. and I.D.; visualization, S.B.; supervision, P.D.B. and Ž.B.; project administration, P.D.B.; funding acquisition, Ž.B. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Data Availability Statement: The source code for the CSRM framework, including the eBPF/XDP telemetry agent and PID controller implementation, is publicly available at <https://github.com/stefanbisevac028/CSRM> (accessed on 10 April 2026). The raw per-run CSV logs for the PID gain sensitivity analysis (Section 5.6.2) are included in the same repository.

Conflicts of Interest: The authors declare no conflicts of interest.

Abbreviations

The following abbreviations are used in this manuscript:

SDN	Software-Defined Networking
OVS	Open vSwitch
eBPF	extended Berkeley Packet Filter
XDP	eXpress Data Path
PID	Proportional-Integral-Derivative
CPU	Central Processing Unit
NIC	Network Interface Card
RSS	Receive Side Scaling
gRPC	Google Remote Procedure Call
CSRM	Central Slice Resource Manager
VM	Virtual Machine
API	Application Programming Interface

Appendix A. Comparison with Related Work

This appendix provides a detailed comparison between the proposed CPU-aware traffic control framework and related work. Table A1 summarizes the key focus of each referenced work and highlights the differences with our approach. Table A2 presents the quantitative metrics for comparison.

Table A1. Comparison of related work: key focus and proposed solution differences.

Ref.	Key Focus	Proposed Solution–Difference
[11]	Controller-side periodic polling (low frequency), no actuation	In contrast to PayLess, which relies on controller-side polling with seconds-scale latency and lacks actuation capabilities, our approach performs in-band eBPF/XDP telemetry with closed-loop PID actuation directly in the dataplane, achieving sub-second (500 ms) response times.
[14]	Framework for evaluating OpenFlow switches (testing, not runtime control)	While OFLOPS provides a valuable benchmarking framework for offline switch evaluation, our system addresses the complementary challenge of runtime closed-loop control in production environments.
[22]	XDP at line rate, low processing cost	Building upon the XDP programmability demonstrated by BPFabric, we extend the paradigm by applying XDP specifically for CPU-aware rate limiting with PID feedback, enabling adaptive traffic control rather than static packet processing.
[18]	eBPF/XDP monitoring in-kernel	The work by Abranches et al. establishes efficient in-kernel monitoring using eBPF and XDP; our contribution extends this foundation by incorporating actuation through PID-controlled rate limiting with per-slice granularity.
[23]	In-kernel telemetry, scalability and low overhead	The XDP paper establishes the foundation for low-overhead packet processing in the kernel; we leverage this capability and augment it with closed-loop CPU regulation to enable dynamic resource management.
[25]	XDP/BPF in 5G UPF, single-tenant, adaptation	Whereas Tomovic et al. target single-tenant 5G UPF adaptation, our framework extends to multi-tenant environments with per-slice CPU targets and attribution, enabling fine-grained isolation between network slices.
[21]	Programmable dataplane (BPF in SDN)	The comprehensive survey by Vieira et al. catalogues eBPF/XDP capabilities; our work complements this by implementing and experimentally validating a concrete CPU-aware closed-loop system for OVS dataplanes.

Table A1. Cont.

Ref.	Key Focus	Proposed Solution–Difference
[10]	OVS design/performance	The seminal OVS paper by Pfaff et al. describes static policing and metering mechanisms; our contribution introduces dynamic CPU-aware control that adaptively adjusts rate limits based on real-time resource utilization.
[31]	SDN-based slicing (control plane, orchestration)	While Alhussein et al. address control-plane orchestration for network slicing, our work focuses on the complementary dataplane aspect, providing CPU-aware isolation through per-slice telemetry and XDP-based actuation.

Table A2. Quantitative comparison metrics.

Ref.	Metric	Related Work	Our Solution
[11]	Detection-to-action time Monitoring overhead (%CPU) Actuation capability	Seconds (controller polling) Low (adaptive polling) None	~500 ms (control loop) 5–10% (with actuation) PID-based rate limiting
[22]	CPU% at 1 Gbps Drops/pps at ingress Control loop	<5% (XDP forwarding) Line-rate capable None	~42% (total system, with OVS + control) Up to 271 K drops at 3 Gbps Closed-loop PID
[18]	Overhead at 100 ms sampling Per-slice granularity Actuation	<1% (monitoring only) N/A None	5–10% (monitoring + actuation) Supported XDP rate limiting
[23]	Probe overhead Telemetry latency (ms)	<1% <1 ms	5–10% (with PID control) ~500 ms (control interval)
[25]	CPU target error Settling time Per-slice isolation Loss difference (isolation)	Not reported Not reported Single-tenant N/A	1.02% at 3% target <1 s Multi-tenant >2000×
[10]	CPU% at 600 Mbps Packet loss at 500 Mbps CPU target tracking XDP drops at 1500 Mbps	~10% (OVS baseline) 0% (no limiting) N/A (static policing) N/A	~10% (with control) 40.94% (3% target) 1.02% vs 3% target 271 K drops

References

- McKeown, N.; Anderson, T.; Balakrishnan, H.; Parulkar, G.; Peterson, L.; Rexford, J.; Turner, J. OpenFlow: Enabling innovation in campus networks. *ACM SIGCOMM Comput. Commun. Rev.* **2008**, *38*, 69–74. [\[CrossRef\]](#)
- Nunes, B.A.A.; Mendonca, M.; Nguyen, X.N.; Obraczka, K.; Turletti, T. A survey of software-defined networking: Past, present, and future of programmable networks. *IEEE Commun. Surv. Tutor.* **2014**, *16*, 1617–1634. [\[CrossRef\]](#)
- Kreutz, D.; Ramos, F.M.; Verissimo, P.E.; Rothenberg, C.E.; Azodolmolky, S.; Uhlig, S. Software-defined networking: A comprehensive survey. *Proc. IEEE* **2014**, *103*, 14–76. [\[CrossRef\]](#)
- Ebrahimi, S.; Bouali, F.; Haas, O.C. Resource management from single-domain 5G to end-to-end 6G network slicing: A survey. *IEEE Commun. Surv. Tutor.* **2024**, *26*, 2836–2866. [\[CrossRef\]](#)
- Dalgitsis, M.; Cadenelli, N.; Serrano, M.A.; Bartzoudis, N.; Alonso, L.; Antonopoulos, A. Cloud-native orchestration framework for network slice federation across administrative domains in 5G/6G mobile networks. *IEEE Trans. Veh. Technol.* **2024**, *73*, 9306–9319. [\[CrossRef\]](#)
- Heller, B.; Sherwood, R.; McKeown, N. The controller placement problem. *ACM SIGCOMM Comput. Commun. Rev.* **2012**, *42*, 473–478. [\[CrossRef\]](#)
- Tootoonchian, A.; Ganjali, Y. HyperFlow: A distributed control plane for OpenFlow. In Proceedings of the 2010 Internet Network Management Conference on Research on Enterprise Networking, San Jose, CA, USA, 27 April 2010; p. 6.
- Yeganeh, S.H.; Ganjali, Y. Kandoo: A framework for efficient and scalable offloading of control applications. In Proceedings of the First Workshop on Hot Topics in Software Defined Networks, Helsinki, Finland, 13 August 2012; pp. 19–24.
- Dixit, A.; Hao, F.; Mukherjee, S.; Lakshman, T.V.; Kompella, R. Towards an elastic distributed SDN controller. *ACM SIGCOMM Comput. Commun. Rev.* **2013**, *43*, 7–12. [\[CrossRef\]](#)

10. Pfaff, B.; Pettit, J.; Koponen, T.; Jackson, E.; Zhou, A.; Rajahalme, J.; Casado, M. The design and implementation of Open vSwitch. In *Proceedings of the 12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, Oakland, CA, USA, 4–6 May 2015; pp. 117–130.
11. Chowdhury, S.R.; Bari, M.F.; Ahmed, R.; Boutaba, R. PayLess: A low-cost network monitoring framework for SDN. In *Proceedings of the 2014 IEEE Network Operations and Management Symposium (NOMS)*, Krakow, Poland, 5–9 May 2014; IEEE: Piscataway, NJ, USA, 2014.
12. Bannour, F.; Souihi, S.; Mellouk, A. Distributed SDN control: Survey, taxonomy, and challenges. *IEEE Commun. Surv. Tutor.* **2017**, *20*, 333–354. [\[CrossRef\]](#)
13. Karakus, M.; Duresi, A. Control plane scalability issues and approaches in SDN. *Comput. Netw.* **2017**, *112*, 279–293. [\[CrossRef\]](#)
14. Rotsos, C.; Sarrar, N.; Uhlig, S.; Sherwood, R.; Moore, A.W. OFLOPS: An open framework for OpenFlow switch evaluation. In *International Conference on Passive and Active Network Measurement*; Springer: Berlin/Heidelberg, Germany, 2012; pp. 85–95.
15. Jarschel, M.; Oechsner, S.; Schlosser, D.; Pries, R.; Goll, S.; Tran-Gia, P. Modeling and performance evaluation of an OpenFlow architecture. In *Proceedings of the 2011 23rd International Teletraffic Congress (ITC)*, San Francisco, CA, USA, 6–9 September 2011; IEEE: Piscataway, NJ, USA, 2011.
16. Tran, M.N.; Kim, Y. Network performance benchmarking for containerized infrastructure in NFV environment. In *Proceedings of the 2022 IEEE 8th International Conference on Network Softwarization (NetSoft)*, Milan, Italy, 27 June–1 July 2022; IEEE: Piscataway, NJ, USA, 2022; pp. 115–120.
17. Li, G.; Wang, X.; Zhang, Z. SDN-based load balancing scheme for multi-controller deployment. *IEEE Access* **2019**, *7*, 39612–39622. [\[CrossRef\]](#)
18. Abranches, M.; Michel, O.; Keller, E.; Schmid, S. Efficient network monitoring applications in the kernel with eBPF and XDP. In *Proceedings of the 2021 IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN)*, Heraklion, Greece, 9–11 November 2021; IEEE: Piscataway, NJ, USA, 2021.
19. Prometheus Authors. The Prometheus Monitoring System: Documentation and Architecture Overview. 2023. Available online: <https://prometheus.io/docs/introduction/overview/> (accessed on 11 April 2026).
20. Han, S.; Jang, K.; Park, K.; Moon, S. PacketShader: A GPU-accelerated software router. *ACM SIGCOMM Comput. Commun. Rev.* **2010**, *40*, 195–206. [\[CrossRef\]](#)
21. Vieira, M.A.; Castanho, M.S.; Pacifico, R.D.; Santos, E.R.; Júnior, E.P.C.; Vieira, L.F. Fast packet processing with eBPF and XDP. *ACM Comput. Surv.* **2020**, *53*, 16. [\[CrossRef\]](#)
22. Jouet, S.; Pezaros, D.P. BPFabric: Data plane programmability for software defined networks. In *Proceedings of the ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*; IEEE: Piscataway, NJ, USA, 2017; pp. 38–48.
23. Høiland-Jørgensen, T.; Brouer, J.D.; Borkmann, D.; Fastabend, J.; Herbert, T.; Ahern, D.; Miller, D. The express data path: Fast programmable packet processing in the operating system kernel. In *Proceedings of the 14th International Conference on Emerging Networking Experiments and Technologies (CoNEXT)*, Heraklion, Greece, 4–7 December 2018; IEEE: Piscataway, NJ, USA, 2018; pp. 54–66.
24. Shin, J.; Kim, J.; Nam, J. Aquila: Efficient In-Kernel System Call Telemetry for Cloud-Native Environments. *Sensors* **2025**, *25*, 6511. [\[CrossRef\]](#) [\[PubMed\]](#)
25. Tomovic, S.; Radusinovic, I.; Prasad, N.R. Run-Time Adaptive In-Kernel BPF/XDP Solution for 5G UPF. *Electronics* **2022**, *11*, 1022. [\[CrossRef\]](#)
26. Montazerolghaem, A. Energy-efficient Software-defined 5G/6G Multimedia IoV: PID controller-based approach. *IEEE Trans. Veh. Technol.* **2025**, *early access*. [\[CrossRef\]](#)
27. Malavika, R.; Valarmathi, M.L. Adaptive Server Load Balancing in SDN Using PID Neural Network Controller. *Comput. Syst. Sci. Eng.* **2022**, *42*, 229. [\[CrossRef\]](#)
28. Xie, J.; Yu, F.R.; Huang, T.; Xie, R.; Liu, J.; Wang, C.; Liu, Y. A survey of machine learning techniques applied to software defined networking (SDN): Research issues and challenges. *IEEE Commun. Surv. Tutor.* **2018**, *21*, 393–430. [\[CrossRef\]](#)
29. Alwakeel, A.M.; Alnaim, A.K. Unleashing the Potential of 5G for Smart Cities: A Focus on Real-Time Applications. *Smart Cities* **2025**, *8*, 70. [\[CrossRef\]](#)
30. Botez, R.; Pasca, A.G.; Dobrota, V. Efficient Network Slicing with SDN and Heuristic Algorithm for 5G Networks. *Sensors* **2023**, *23*, 6053. [\[CrossRef\]](#) [\[PubMed\]](#)
31. Alhussein, O.; Hadjadj-Aoul, Y.; Outtagarts, A. SDN-Based Network Slicing Mechanism for a Scalable 4G/5G Core Network. *Sensors* **2021**, *21*, 3773. [\[CrossRef\]](#)
32. 3GPP. Service Requirements for Enhanced V2X Scenarios. Technical Report TS 22.186, 3rd Generation Partnership Project. 2024. Available online: https://www.etsi.org/deliver/etsi_ts/122100_122199/122186/18.00.01_60/ts_122186v180001p.pdf (accessed on 9 May 2026).
33. oneM2M. Requirements. Technical Report TS-0002 Version 3.1.2, oneM2M. 2019. Available online: https://www.etsi.org/deliver/etsi_ts/118100_118199/118102/03.01.02_60/ts_118102v030102p.pdf (accessed on 9 May 2026).

34. ETSI. *Multi-Access Edge Computing (MEC); Phase 2: Use Cases and Requirements*; Technical Report GS MEC 002 V2.1.1; European Telecommunications Standards Institute: Sophia Antipolis, France, 2018.
35. Biševac, S. CSRM: CPU-Aware Slice Resource Management Framework. GitHub Repository. 2026. Available online: <https://github.com/stefanbisevac028/CSRM> (accessed on 10 April 2026).

Disclaimer/Publisher’s Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.