The Brewing Storm in 5G's Data Plane: Design and Evaluation of a High-Performance eBPF/XDP-Based User Plane Function

Franck Messaoudi*, Luhan Wang†, Abdelkader Mekrache*, Adlen Ksentini*, Bingxuan Li†, Jialei Su†, Sofiane Messaoudi*, Salim El Ghalbzouri*
*Eurecom Institute, Sophia Antipolis, France
†Beijing University of Posts and Telecommunications (BUPT), Beijing, China
Email: *{franck.messaoudi, abdelkader.mekrache, adlen.ksentini, sofiane.messaoudi, salim.elghalb}@eurecom.fr
†{wluhan, chandlerbing, sjl_rick}@bupt.edu.cn

***Abstract*—This paper presents the design and implementation of a novel 5G UPF leveraging eBPF technology to meet the stringent performance and programmability requirements of emerging 6G systems. Traditional UPF implementations often struggle to balance performance, flexibility, and resource efficiency-challenges particularly critical in CPU- and I/O-constrained edge environments. The proposed eBPF-based UPF architecture mitigates these limitations by embedding core functionalities, such as packet classification, forwarding, and QoS enforcement, directly within the Linux kernel via eBPF programs attached through XDP and tc hook points. Performance evaluation using TRex demonstrates that the proposed solution achieves competitive throughput, low packet loss, and efficient CPU utilization across traffic profiles. Moreover, it maintains full compliance with 5G Core Network standards. Comparative analysis with well-established open-source UPF implementations further underscores its advantages. This work highlights the potential of eBPF as a foundational technology for building next-generation, programmable UPFs optimized for edge cloud deployments in the 6G era.**

***Index Terms*—5^{th} Generation Mobile Networks (5G), User Plane Function (UPF), QoS Enforcement Rule (QER), Quality of Service (QoS), extended Berkeley Packet Filter (eBPF), eXpress Data Path (XDP), Traffic Control (tc), Queuing Discipline (qdisc).**

I. Introduction

The evolution of mobile networks toward 5G and emerging 6G technologies introduces unprecedented performance demands, driven by next-generation services, which require not only ultra-high data rates but also consistent, ultra-low latency, reliability, and adaptability [1]. To meet these expectations, the data plane—the component responsible for handling user traffic—must evolve significantly. This challenge is amplified by modern Radio Access Network (RAN) systems capable of transmitting gigabytes of data per second [2], which place enormous pressure on the data plane to efficiently aggregate, process, and forward massive volumes of traffic while maintaining strict Quality of Service (QoS) guarantees.

In response to these challenges, the 3rd Generation Partnership Project (3GPP) redesigned the 5G Core Network (CN) architecture around the Software-Defined Networking (SDN) principles, separating the control and data planes to enable centralized management, dynamic programmability, and streamlined network orchestration [3]. This shift enhances scalability, accelerates service deployment, and improves resource efficiency across infrastructures. Aligned with this vision, 3GPP introduced the Service-Based Architecture (SBA) framework, an architectural paradigm splitting the CN into modular, virtualized, and self-contained network functions. These functions interact through service-based interfaces, exposing and consuming services in a discoverable manner that enables flexibility, reusability, and easier integration [4].

Within this architecture, the Session Management Function (SMF) [5] is the control-plane function responsible for managing and orchestrating data plane sessions, playing a role analogous to an SDN controller. From the data plane perspective, 3GPP introduced the User Plane Function (UPF) [4], which interconnects the RAN with external networks, referred to as Data Network (DN), and manages user traffic across both ends. The UPF acts like an SDN switch with capabilities such as QoS enforcement, buffering to support mobility, and usage reporting for charging. Instead of adopting OpenFlow, 3GPP defined the Packet Forwarding Control Protocol (PFCP) [6] to control the UPF, supporting these and additional features. This architecture also mirrors the European Telecommunications Standards Institute (ETSI) Network Function Virtualization (NFV) Management and Orchestration (MANO) framework. The SMF assumes a role analogous to the NFV Orchestrator (NFVO) and Virtualized Network Function (VNF) Manager (VNFM), governing the UPF's lifecycle and behavior, while the UPF acts as a VNF exposing data-path capabilities via the PFCP southbound interface- analogous to MANO's *Ve-Vnfm* and *Or-Vi* reference points. This architectural parallel underscores how 3GPP has internalized SDN/NFV principles to achieve flexible, software-driven network operation.

The specification also envisions a distributed data plane architecture, where multiple UPFs are interconnected to prevent bottlenecks and reduce latency. Typically, a centralized UPF connects the external DN, while local UPFs, positioned closer to the RAN, enable edge breakout for latency-sensitive applications and reduce CN load [4].

The UPF has thus emerged as the most performance-critical element within the 5G CN, as it operates under intense constraints and assumes user plane responsibilities. It must sustain high-throughput, low-latency, enforce QoS and policies, adapt to user mobility, and collect traffic metrics for charging and analytics [4]. These demanding tasks impose sig-

nificant computational and architectural pressure, making the UPF a potential bottleneck—or enabler—for the performance, scalability, and efficiency of the entire CN.

Traditionally, UPFs have been deployed as proprietary, hardware-based appliances developed by major telecom vendors—often costly and inflexible. In response, the research community has explored diverse architectural strategies to enable open, efficient, and programmable alternatives. Baseline solutions like Open5GS [7] and free5GC [8] rely on the conventional Linux networking stack based on sockets and kernel Input/Output (I/O). While they prioritize modularity and 3GPP compliance, their throughput remains constrained due to reliance on socket-based I/O and kernel overhead. To overcome these limitations, the open-source community has pursued alternative architectures, each offering different trade-offs between performance, flexibility, and system integration.

(i) *Kernel bypass*, which enables user-space packet processing. Frameworks like Data Plane Development Kit (DPDK) [9] provide direct access to Network Interface Controllers (NICs), eliminating context switches and interrupt overhead, yielding high throughput and low latency. This comes at the cost of Central Processing Unit (CPU) monopolization through busy-polling, which consumes nearly 100%, and poor multi-tenant resource sharing—a key constraint in edge [10]. A representative example is the UPG-VPP [11], a User Plane Gateway (UPG) leveraging Vector Packet Processing (VPP)'s DPDK-based architecture to deliver high-speed forwarding and General Packet Radio Service (GPRS) Tunnelling Protocol (GTP) User Plane (GTP-U) encapsulation.

(ii) *High-speed in-kernel processing* targets fast-path packet handling within the kernel to eliminate the overhead of context switches and user-kernel transitions. Technologies like extended Berkeley Packet Filter (eBPF) [12], [13], FlexNIC [14], NetFPGA [15], and smart NICs [16] embody this model, enabling high-throughput, low-latency under extreme traffic conditions of $10\,\mathrm{Gbps}$+ links [17], [18], while maintaining portability and integration. eUPF [19] exemplifies this approach by leveraging eBPF to implement a lightweight UPF with high observability and deployability. However, its focus on minimalism and observability comes at the cost of limited feature completeness and scalability. Notably, it supports only one Forwarding Action Rule (FAR), QoS Enforcement Rule (QER), and Service Data Flow (SDF) filter per Packet Detection Rule (PDR) or tunnel, lacks dynamic peer discovery, and requires manual GTP configuration. Beyond eUPF, academic work has explored eBPF for the UPF, yet each addresses only a narrow slice of the problem. Cable [20] uses eBPF/eXpress Data Path (XDP) to reduce per-packet processing time in a single UPF but does not tackle QoS enforcement. Polycube [21] provides a general framework for eBPF network functions with a 5G mobile gateway prototype, yet remains tied to its own runtime rather than the plain Linux kernel. Broader surveys of eBPF for 5G and beyond [22] highlight observability and security use cases but leave the UPF data path untreated. Other academic prototypes have evaluated XDP-based UPF designs in containerized environments [23], confirming the viability of kernel-native data paths for cloud-native deployments. A full, carrier-grade UPF that combines high-performance forwarding with fine-grained QoS enforcement within the kernel is still missing.

(iii) *Programmable data plane*, allowing switches, routers, or NICs customization for specialized packet processing. Enabled by SDN and NFV, it supports dynamic, software-driven behavior. Programming the data plane uses languages like Programming Protocol-Independent Packet Processors (P4), OpenFlow, or Click to define rules and policies governing packet processing, forwarding, and modification. This approach enables line-rate performance with deterministic latency, through hardware offloading, but suffers from limited flexibility, hardware dependency, and a steep learning curve. Software-Defined Fabric (SD-Fabric) [24] and UPF-Evolved Packet Core (UPF-EPC) [25] explore this approach. The former deploys a P4-based UPF on programmable switch Application-Specific Integrated Circuits (ASICs), achieving hardware-level performance. The latter is a 3GPP-compliant UPF featuring two data paths: UP4 [26], leveraging Open Network Operating System (ONOS) and P4-programmable switches, and Berkeley Extensible Software Switch (BESS) [27], a software-based pipeline. Beyond these projects, recent academic work has pushed the boundaries of hardware-assisted UPF design. HiP4-UPF [28] shows that a comprehensive UPF, covering most 3GPP-mandated features, can be deployed entirely on commodity P4 switches, while heterogeneous integration frameworks [29] unify DPDK- and SmartNIC-based data planes under a common PFCP control plane. These designs achieve high throughput but inherit the hardware-dependency and portability limitations characteristic of the category, reinforcing the need for a software-based solution that does not rely on specialized silicon.

In this work, we rethink the UPF architecture based on the emerging eBPF, an in-kernel technology that reshapes how fast-path data processing can be done in Linux. Unlike kernel-bypass architectures that trade performance for portability and resource contention, our design embraces eBPF to combine line-rate performance with safe, fine-grained programmability in the kernel. We introduce a modular UPF architecture that splits responsibilities between a Management Layer—handling PFCP signaling—and a Data-Path Layer built from eBPF programs of types XDP and Traffic Control (tc), to implement parsing, classification, forwarding, and QoS enforcement.

To validate our approach, we extensively evaluated our UPF on a high-performance testbed using TRex [30], a widely adopted traffic generator for data plane performance testing. We benchmarked our design against leading open-source UPFs, namely Open5GS, free5GC, and eUPF on the 100 Gbps testbed, and additionally against UPG-VPP in a separate virtualized environment. The results demonstrate that our solution matches or exceeds the performance of existing implementations, while remaining kernel-native, portable, and resource-efficient-key properties for sustainable deployment at the network edge.

Our contributions are summarized as follows:

- A carrier-grade UPF implementation based on the emerging eBPF technology, supporting high-speed data forwarding and QoS. A 3GPP-conformant UPF over the N3, N4 and N6 interfaces that outperforms the evaluated

open-source software implementations in CPU efficiency and packet loss, at comparable throughput.
- A complete open-source solution integrated with the OpenAirInterface (OAI) CN.
- A portability analysis across the Linux NIC ecosystem, characterizing how XDP attachment modes, buffer management, queue scalability, and driver maturity govern the deployability of the proposed UPF beyond the evaluation testbed.

The remaining sections of this paper are structured as follows: Section II overviews the 3GPP Core, focusing on the UPF and QoS mechanisms. Section III presents the eBPF framework, including XDP and tc hooks for high-performance packet processing. Section IV describes the Linux tc subsystem—qdiscs, classes, filters, policing, and shaping—that forms flexible traffic-handling pipelines. Section V details the design of the proposed UPF. Section VI describes the methodology and testbed. Section VII presents and discusses the results. Section VIII analyzes the portability of the proposed UPF across the Linux NIC ecosystem, discussing how XDP support, buffer management, queue scalability, and driver maturity influence its deployment beyond our testbed. Finally, Section IX concludes the paper.

II. 5G Core in 3GPP Specifications: An Overview of User Plane and QoS Mechanisms

A. UPF

The UPF serves as the data plane anchor of the 5G CN, bridging User Equipment (UE) traffic to the DN and handling routing, forwarding, inspection, and QoS enforcement. It is controlled by the SMF via the PFCP protocol over the N4 interface, while interfacing with the gNodeB (gNB) over N3, with the DN over N6, and possibly with other UPFs over N9. The connection direction toward the gNB is known as ACCESS, while toward the Internet is known as CORE. To establish a UE-DN communication, the SMF sends a Packet Data Unit (PDU) session request to the UPF, setting up a bi-directional tunnel. Each PDU session comprises one or more Data Radio Bearers (DRBs) between UE and gNB, and a single *N3 GTP-U Tunnel* between gNB and UPF [31]. The session includes a Data Network Name (DNN) (e.g., *Internet* (for public access), *IP Multimedia Subsystem (IMS)* (for Voice over Long Term Evolution (LTE) (VoLTE)/Voice over New Radio (NR) (VoNR) services), or *Enterprise* (for private networks)) defining the IP connectivity service requested by the UE. Upon setup, session context is shared across UE, gNB, and UPF. For UPF, this includes PDRs, QERs, FARs, and Usage Reporting Rules (URRs). The N3 GTP-U tunnel encapsulates traffic in a GTP header carrying user data identifiers like the QoS Flow Identifier (QFI) (see Figure 2) [32].

Within the UPF, packet processing begins by inspecting incoming packets on the N3, N6, or N9 interfaces to identify the corresponding PFCP session. This is achieved by extracting key header fields—primarily the UE's IP address, which uniquely associates the UE with a PFCP session. The retrieved PFCP Session typically contains multiple PDRs, which are *prioritized* by *precedence*. The UPF sequentially matches the Packet Detection Information (PDI) from each PDR to the corresponding header fields, starting with the highest priority PDR, until a match is found. The relevant rules in the matching PDR are applied to the packet. These rules include the mandatory FAR, along with any optional QER, URR, and Buffer Action Rule (BAR) if applicable [6].

B. QoS Management

1) QoS: The concept of QoS encompasses both quantitative and qualitative parameters required to fulfill specific system performance objectives, expressed as tuples of (parameter, value/range) such as latency, bandwidth, and jitter. In packet-switched networks, QoS refers to a set of techniques and mechanisms that ensure reliable, efficient, and predictable data delivery while meeting predefined performance levels. These mechanisms encompass traffic classification and prioritization, bandwidth allocation, traffic shaping and policing, congestion control, as well as queue and buffer management [33], [34]. Furthermore, QoS can be enforced through dedicated protocols and architectural models, such as the Integrated Services (IntServ) and Differentiated Services (DiffServ), as well as Multi-Protocol Label Switching (MPLS) architectures.

2) 5G QoS Enforcement: In 5G, the QoS model relies on *QoS flows*, which define the *finest* level of QoS granularity within a PDU session. Each flow is identified by a unique QFI and classified as *Guaranteed Bit Rate (GBR)*, *Non-GBR* (best-effort), or *Delay-Critical GBR* (GBR with ultra-low latency and high reliability) [4]. A default Non-GBR flow is always established with a PDU session tied to an Internet DNN, ensuring basic UE connectivity [35]. This flow remains active unless explicitly modified or removed. Additional QoS flows can be added as needed to support specific services like Voice over IP (VoIP), Vehicle-to-Everything (V2X), or Internet of Things (IoT) (see Figure 1).

QoS enforcement is managed by the SMF during PDU session establishment or modification, covering both the Access Network (AN) (UE and gNB) and the DN (via the UPF). In the AN, QoS flows are mapped to DRBs; in the DN, to N3 GTP-U tunnels. The SMF collaborates with the Unified Data Management (UDM) and Policy Control Function (PCF), which provide subscription and authorization data. The PCF responds with Policy Charging and Control (PCC) rules that include packet filters, from which the SMF derives QoS Flow Binding Parameters (e.g., 5G QoS Identifier (5QI), Allocation and Retention Priority (ARP), and Maximum Data Burst Volume (MDBV)). If no matching QoS flow exists, the SMF instantiates a new one [5], [35].

Each flow is tied to a QoS profile specifying parameters and characteristics based on flow type: GBR, Non-GBR, or Delay-Critical GBR, defining resource guarantees and expected behavior, summarized in Table I. Binding PCC rules involves linking flows with QoS profiles, QoS rules, and PDRs. Derived from PCC rules, PDRs inherit precedence and form part of the SDF template sent to the UPF. The SMF then distributes: the QoS profile to the AN via Access and Mobility Management Function (AMF) over N2 interface (AMF/N2), QoS rules to the UE (AMF/N1), and PDRs to the UPF (N4) [5]. Each QoS

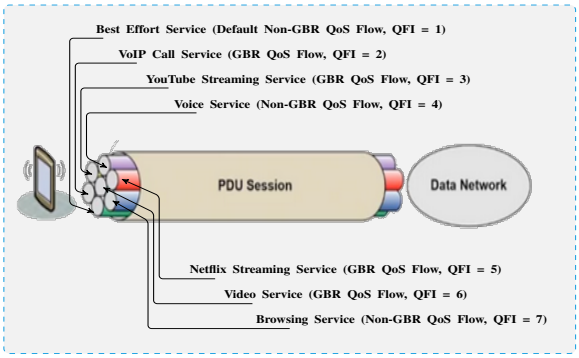

Fig. 1: **PDU Session with 7 QoS Flows (QFI 1–7) for Services like VoIP, Video, and Browsing; Some Use GBR, Others Non-GBR**

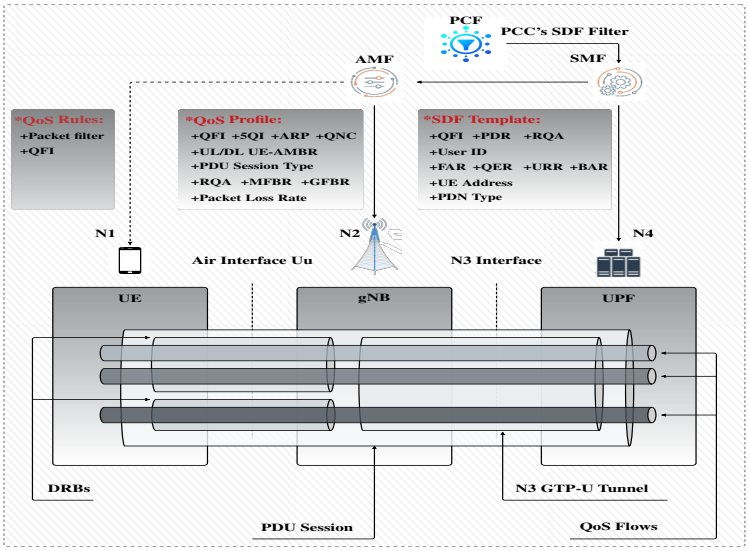

Fig. 2: **5G UPF Data Plane and Control Plane Interaction with QoS Management**

TABLE I: **5G QoS flow parameters**

QoS Flow Type			QoS Flow Parameter
Delay-Critical GBR	GBR	Non-GBR	5G QoS Identifier (5QI)
			QoS Flow Identifier (QFI)
			Allocation and Retention Priority (ARP)
			Reflective QoS Attribute (RQA)
			Aggregate Maximum Bit Rate (AMBR)
			Guaranteed Flow Bit Rate (GFBR)
			Maximum Flow Bit Rate (MFBR)
			Maximum Data Burst Volume (MDBV)
			QoS Notification Control (QNC)
			Averaging Window
			Packet Delay Budget (PDB)
			Packet Error Rate (PER)
			Maximum Packet Loss Rate (MPLR)

flow is uniquely identified by a QFI, and each QoS rule has a session-scoped identifier. This ensures reliable and consistent QoS enforcement across the 5G system (see Figure 2).

III. eBPF-Based Networking: An Overview of XDP and TC Hooks

A. eBPF

eBPF is a Linux kernel feature providing a software-based execution environment that runs sandboxed programs in privileged mode to *safely* and *efficiently* extend kernel capabilities with custom code injected at run-time, without modifying kernel sources or loading modules. eBPF programs are *event-driven*: they are triggered when execution reaches a *hook point*— a predefined location in user space, kernel space, or hardware, where the eBPF program is dynamically attached to intercept, analyze, or modify data in real time. Built-in hook points cover system calls, kernel functions, tracepoints, and network events, to name a few. When such hooks are insufficient, *kprobes* (dynamic probes for tracing kernel functions) and *uprobes* (for tracing user-space application functions) allow the creation of custom hooks to instrument any part of the system [36]–[39].

- *Execution Pipeline*. An eBPF program is deployed through a four-stage pipeline ensuring safety, portability, and efficiency. The program, written in a *restricted form of C*, is *(i) compiled* by the Clang/LLVM toolchain into bytecode and stored in an Executable and Linkable Format (ELF) object file. The BPF library then *(ii) loads* the ELF file into the target hook point via system calls. Once loaded, the kernel *(iii) verifies* against the program safety constraints, such as bounded loops, limited instruction sets, and restricted memory access to eBPF maps (structured key-value stores such as arrays, hashes, and queues shared between kernel and user space). Finally, the bytecode is *(iv) Just-In-Time (JIT) compiled* into machine-specific instructions for native execution [40].

B. XDP

XDP is a category of eBPF programs designed for high-performance packet processing. XDP hook is a callback function placed in the NIC driver's *ingress* path, inside the New Application Programming Interface (API) (NAPI) *poll()* method [41], and runs *before* Socket Kernel Buffer (SKB) allocation (see Figure 3). NAPI is a polling-based kernel feature widely supported by modern drivers. Triggered on every packet reception, the XDP program can parse, classify, and modify the packet—field rewrites, encapsulation/decapsulation, or other operations normally performed by the TCP/IP stack—before returning a verdict: drop (`XDP_DROP`), pass (`XDP_PASS`), transmit (`XDP_TX`), or redirect (`XDP_REDIRECT`) [36].

- *Attachment Modes*. The hook point depends on how the XDP program is attached. We distinguish the *generic*, *native*, and *offloaded* XDP. Generic XDP is loaded into the kernel as part of the regular network path, making it suitable for

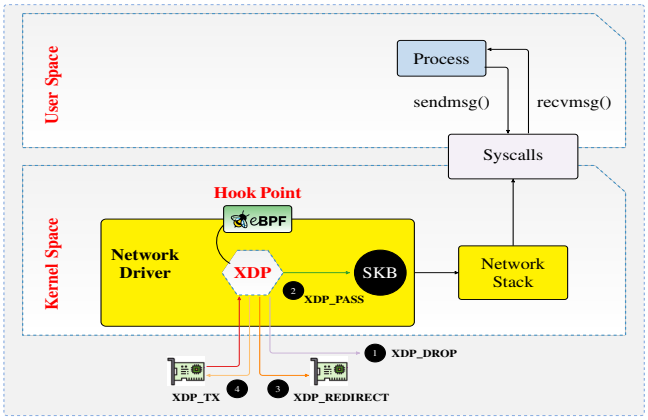

Fig. 3: **eBPF XDP with Possible Actions**

testing. Native XDP is loaded within the network card driver, providing better performance but requiring driver support. Offloaded XDP runs directly within smart embedded NICs supporting eBPF. Network device drivers may not support XDP, in which case the generic model is utilized. In Linux 4.18 and later, supported drivers include Veth, Virtio, Tap, Tun, Thunder, Ixgbe, I40e, and Mlx5 [42].

C. tc

tc, from eBPF perspective, is the second eBPF program class relevant to this work, targeting traffic scheduling and shaping. tc programs attach at the *ingress* or *egress* hook of the Queuing Discipline (qdisc) layer in the network stack, on a per-interface basis, *after* SKB allocation, granting access to higher-level metadata (SKB priorities, class IDs, netfilter marks) unavailable to XDP. This makes tc-bpf well suited to programmable QoS and scheduling in software switches and user-plane functions: programs can shape, police, and prioritize traffic via constructs such as Hierarchical Token Bucket (HTB) or Fair Queuing (FQ), and parse, modify, or redirect SKBs while preserving the eBPF runtime's safety guarantees.

IV. Linux Networking Framework: An Overview of Traffic Control (tc)

The tc subsystem in the Linux kernel is built around five main conceptual components: qdiscs, classes, filters, policing, and shaping. Together they form flexible and powerful packet handling pipelines [43]. Each network device is associated with a qdisc managing its packet queues. Basic qdiscs follow First-In, First-Out (FIFO) behavior, while advanced ones use *filters* to classify packets into traffic classes with priority levels, enabling QoS enforcement on dequeue and transmission.

A. Queueing Discipline

A qdisc is the foundational *scheduler* of Linux traffic control, ordering and prioritizing packet transmissions within a queue according to configured policies. Linux assigns one by default per interface (*fq_codel*, formerly *pfifo_fast*). Each interface exposes two *hook* points: the *egress* (output) and the *ingress* (input) hooks. The egress hook hosts the *egress* qdisc, carries outbound traffic and supports the full range of queuing disciplines with hierarchical classes and filters—and is therefore the primary target of most tc configurations. In contrast, the ingress hook hosts the *ingress* qdisc, which carries inbound traffic but cannot host child classes and serves mainly as an attachment point for filters used in policing.

1) Categories: qdiscs fall into two behavioral categories: *classful* and *classless*. *(i)* Classful qdiscs build a recursive hierarchy tree rooted at the *root* qdisc, where each *parent* class can host multiple *child* classes that may themselves act as parents, enabling nested bandwidth sharing. Within this hierarchy, classful qdiscs organize traffic into multiple classes and attach filters for fine-grained control, rearranging packets between input and output as well as within each class. Packets are enqueued and dequeued *exclusively* at *leaf* classes, each backed by a fundamental classless qdisc responsible for packet queue management. Typical examples include HTB and Class-Based Queuing (CBQ) [44]. *(ii)* In contrast, classless qdiscs are rigid, self-contained schedulers: they support neither classes nor filters, and do not share bandwidth with others. Each manages its own queue according to a built-in algorithm. Examples include Token Bucket Filter (TBF), and Stochastic Fair Queuing (SFQ).

2) qdisc Implementations: There are various qdisc implementations within Linux kernel, each tailored to specific network requirements and challenges influencing the QoS (e.g., latency, throughput, and packet loss rate). Here, we focus on TBF and HTB, central to our research [45]. *(i) TBF* shapes and limits traffic based on a *token rate* and *bucket size*. Tokens represent permission to send packets, and the bucket determines the maximum *burst* size. TBF regulates packet dequeuing speed; packets are transmitted if sufficient tokens are available. Otherwise, deferred. *(ii) HTB* extends TBF to support hierarchical traffic management. HTB classifies traffic into classes based on criteria, each with its own token bucket filter, rate limits, burst sizes, and priorities. These classes are organized hierarchically, where parents control bandwidth distribution among child classes. HTB offers flexible management of Classes of Traffic (CoT), ensuring critical traffic receives necessary bandwidth while protecting the network from congestion or overuse by lower-priority flows.

B. Classes

In the context of Linux tc, a *class* represents a subdivision within a classful *qdisc* that manages a subset of traffic according to specific rules and resource allocations. While a qdisc defines the overall queuing and scheduling behavior for an interface or a branch of the hierarchy, a class serves as an organizational unit within a qdisc to further differentiate and manage traffic flows. Each class is identified by a unique `classid`, a 32-bit (*u32*) identifier typically expressed in the format `major:minor`. The *major* part corresponds to the identifier of the parent qdisc, while the *minor* part uniquely identifies the class under that qdisc. Through classes, administrators can apply different queuing disciplines, priorities, or rate limits within a single qdisc, enabling highly granular and hierarchical traffic management [43].

C. Filters

In the tc subsystem, filters are key components for packet classification, serving as the *decision-making* logic that categorizes packets based on criteria such as IP addresses, ports, and protocol. Typically attached to *classful* qdiscs, filters inspect incoming packet headers and match them against predefined criteria to determine in which class packets will be enqueued. This enables traffic flow differentiation and the enforcement of targeted policies. When a filter identifies a match, it triggers an associated action, such as packet redirection, dropping, or forwarding. The filter framework is extensible, supporting custom logic for efficient, policy-driven packet processing. The Linux kernel offers various filters: u32_filter and *tc_bpf* are particularly relevant to our work [45].

- *tc-bpf*. A powerful and flexible filtering mechanism extending the *tc* subsystem by integrating *eBPF programmability* into the networking stack at the Link layer. Attached via the `cls_bpf` classifier—a tc mechanism applying eBPF programs at qdisc ingress or egress. tc-bpf operates directly on the `sk_buff` structure to access packet metadata (including Virtual Local Area Network (VLAN) tags, traffic classes, and custom fields). This enables advanced operations such as header modification, redirection, and traffic marking via eBPF programs. The `direct-action` mode in tc-bpf further improves efficiency by combining classification and action: programs classify packets and directly return action codes to forward, drop, or reclassify them. Importantly, tc-bpf programs can be updated at runtime without disrupting traffic, enabling dynamic and efficient traffic management. This makes tc-bpf well suited for high-throughput environments and implementing complex QoS policies based on custom logic.

D. Policing

Policers are traffic control mechanisms that enforce specific *limits* on traffic flows, defined by parameters such as *rate* (allowed traffic rate), *ceil* (maximum tolerable burst rate), and *burst* size (maximum data allowed over a short interval). When traffic conforms to these thresholds, it is handled normally; otherwise, a policy is applied. These *policies* define actions such as *dropping* packets, *reclassifying* them to a different priority, *redirecting* traffic to another path, or *remarking* packet headers (e.g., adjusting Type of Service (ToS) values) to signal congestion. Operationally, a policer performs a binary decision per packet: if the traffic rate is within the threshold, the packet is accepted; otherwise, the policy is enforced. Internally, policers use *token bucket* algorithms to track traffic volume, but unlike shapers, they cannot buffer or delay packets [43]. Policing can be applied at multiple stages of the packet path, including filters (traffic-based decisions), enqueue operations (where packets may be refused), within queuing disciplines (where excess traffic leads to drops), and during packet admission to ensure compliance with constraints, thereby contributing to overall network performance and fairness.

E. Shaping

Refers to the process of *deliberately delaying* packets to regulate the rate at which traffic leaves a network interface or queue. By temporarily holding packets in an output queue, shapers absorb bursts and gradually release traffic at the desired transmission rate. Unlike policing, which enforces rate limits by dropping or reclassifying excess packets, shaping smooths bursts by buffering and pacing transmissions, avoiding loss. A shaper ensures traffic conforms to bandwidth constraints—such as average rate, peak rate, and burst size—using token bucket algorithms. When packets arrive faster than allowed, they are *temporarily* queued and released *gradually* according to shaping parameters, preventing spikes causing downstream congestion. Shaping is usually applied at the egress path playing a key role for QoS guarantees, smoothing jitter-sensitive flows and avoiding packet loss on congested links. Classful queuing disciplines such as HTB and CBQ integrate shaping per class, while classless qdiscs like TBF focus entirely on shaping single flows [43].

V. Proposed Solution

In this section, we present our proposed 5G (and beyond) UPF design, built around aspects shown in the aforementioned sections. We provide a global overview of the solution before delving into the detailed design and implementation. The proposal highlights key contributions, combining eBPF for high-performance and programmable packet processing, as well as tc for fine-grained QoS enforcement.

A. UPF Architecture: Global View

We begin with an overview of the proposed UPF architecture, which builds upon the ETSI/3GPP definitions of the data-plane summarized in Section II-A. The design is organized into two complementary layers: the *Management Layer*, which provides configuration and control capabilities, and the *Data-Path Layer*, which ensures high-performance packet forwarding and enforces QoS policies.

1) *Management Layer*: Enclosed within the user space, it operates as a core library responsible for configuring the user Data-Path. It is designed as follows:
   a) *YAML Config Validator*: This component loads configuration from a YAML file to set up the UPF, delineate interfaces (N3, N4, N6, N9), and define DNN-slice mappings. It configures key Data-Path settings such as packet forwarding and QoS enforcement, and supports eBPF acceleration and QoS policies.
   b) *PFCP Session Manager*: Responsible for managing PFCP sessions, it receives packet processing rules from the SMF via N4, allowing fine-tuning of the Data-Path for efficient packet forwarding. It handles establishing, modifying, and terminating PFCP requests.
   c) *eBPF Program Manager*: This central entity manages the lifecycle of eBPF programs in the UPF Data-Path. It dynamically loads, unloads, and modifies programs through *eBPF skeleton* interfaces, which invoke *bpf() system calls*. By preserving PFCP session context, it enables real-time configuration, customized packet filtering, traffic shaping, and manages eBPF Maps.
2) *Data-Path Layer*: This layer processes the user traffic. It is a Service Function Chaining (SFC), comprising

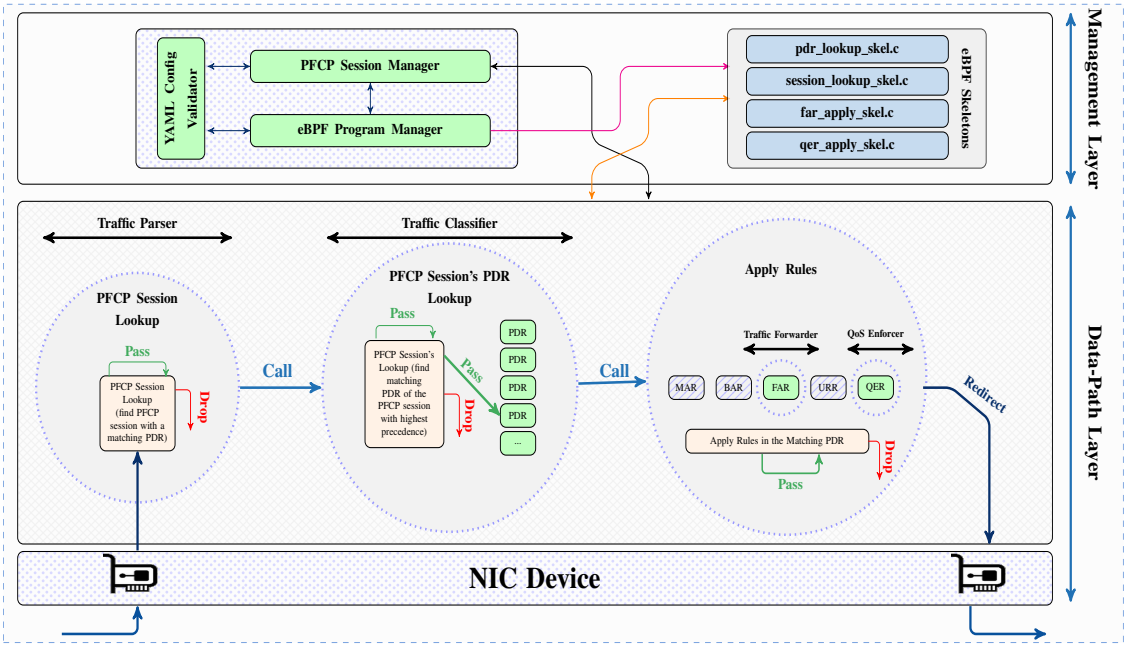

Fig. 4: **UPF Architecture (Global View)**

the four following components. Each of these forms a pipeline with multiple stages. At every stage, a decision is made on the packet's fate: move to the next stage, be discarded under certain conditions, or be redirected.

a) *Traffic Parser*: Processes incoming user traffic, extracting relevant headers, and performing the PFCP Session Lookup at the entry point to the Data-Path Layer.
b) *Traffic Classifier*: Classifies traffic according to various criteria into CoTs based on PDRs and matching PDIs, serving as the PFCP Session's PDR Lookup.
c) *Traffic Forwarder*: Redirects traffic by routing to the CORE (i.e., DN) via N6 with GTP header removal, or to the ACCESS (i.e., RAN) via N3 with appropriate GTP encapsulation. It represents the FAR.
d) *QoS Enforcer*: Guarantees traffic complies with predefined QoS standards through shaping, prioritization, and bandwidth allocation. It continuously monitors traffic and dynamically adjusts rates and priorities in real time. It represents the QER.

Figure 4 provides a global view of the full UPF architecture. The upper half shows the Management Layer (YAML Config Validator, PFCP Session Manager, and eBPF Program Manager) residing in user space, while the lower half depicts the Data-Path Layer as a chain of four components—Traffic Parser, Traffic Classifier, Traffic Forwarder, and QoS Enforcer—executing in kernel space. Arrows show the control flow from the Management Layer down to the NIC device.

B. UPF Management Layer: PFCP Data Model

The SMF coordinates packet processing within the UPF by supervising two main actions: managing PFCP Session contexts and provisioning rules. PFCP Session context management is accomplished through PFCP signaling procedures, including *PFCP Establishment*, *Modification*, and *Deletion Request*, allowing the SMF to respectively establish, modify, and delete PFCP Session contexts corresponding to individual PDU sessions. Rules provisioning involves the addition, modification, or deletion of packet rules such as PDRs, FARs, QERs, and URRs within a PFCP Session context [6]. Figure 16 (in Appendix B) provides a non-exhaustive overview of the PFCP protocol design, highlighting the relevant Information Elements (IEs) and groups of IEs, along with the corresponding description sections from the 3GPP standard.

C. UPF Data-Path Layer: Deployment Modes

The Data-Path Layer supports three distinct deployment modes: *Simple-Switch*, *eBPF-XDP*, and *eBPF-XDP-TC*. The deployment mode determines whether the Data-Path operates in user space or kernel space. As Simple-Switch, the Data-Path is running within the user space. In contrast, When eBPF-XDP or eBPF-XDP-TC modes are used (i.e., eBPF accelerations are enabled), the Data-Path is running within the kernel space.

1) **Simple-Switch:** This mode operates as a Linux-based router, relying on the in-kernel TCP/IP stack for traditional packet processing. Packets traverse the kernel network stack and undergo essential operations such as GTP encapsulation and decapsulation, routing, and other core user-plane processing.
2) **eBPF-XDP:** This mode leverages eBPF programs attached via XDP hooks. Unlike Simple-Switch, operations occur immediately after packet reception by the NIC driver. eBPF-XDP enables programmable packet inspection, filtering, and modification without traversing TCP/IP stack, reducing latency and CPU overhead. It supports dynamic handling of PDRs and FARs via eBPF maps.
3) **eBPF-XDP-TC:** Building upon eBPF-XDP, this mode adds QoS enforcement (i.e., QERs) into the kernel packet

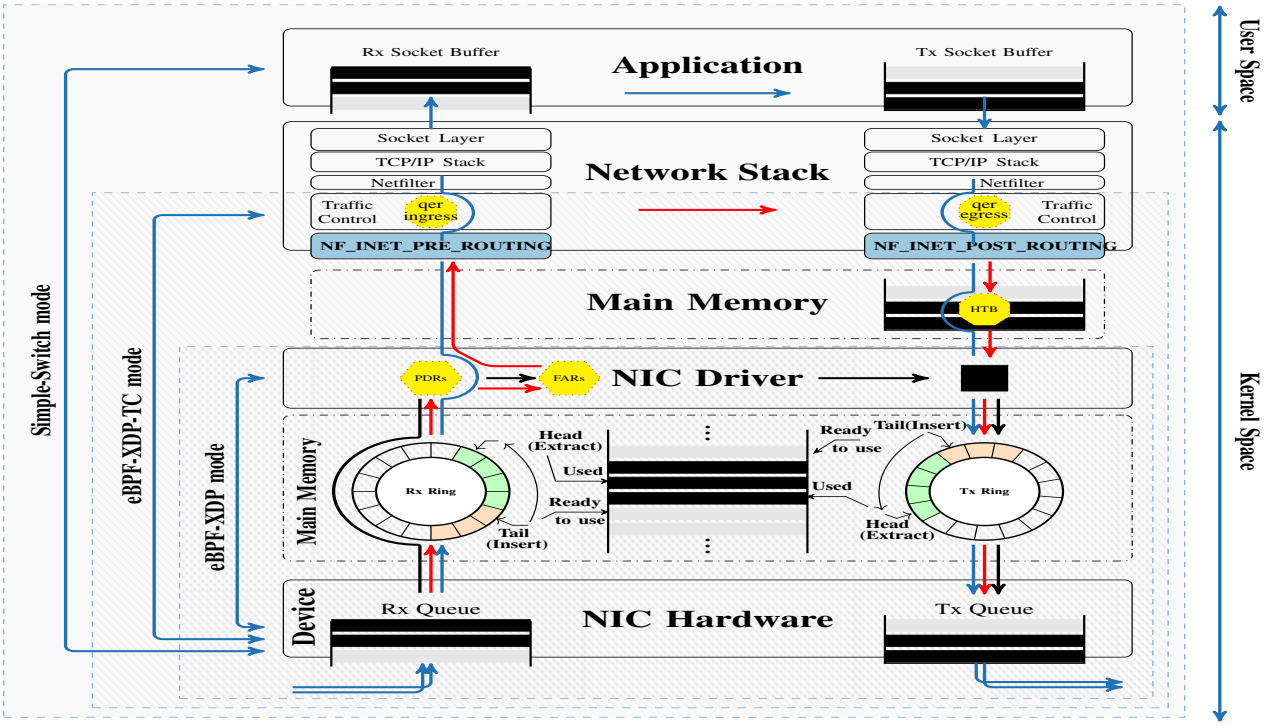

Fig. 5: **UPF Data-Path Deployment Modes**

path. While eBPF-XDP handles programmable inspection, filtering, and forwarding, eBPF-XDP-TC extends this with traffic shaping, policing, and prioritization at the tc layer. Unlike Simple-Switch, which traverses the full TCP/IP stack, and eBPF-XDP, which bypasses it, eBPF-XDP-TC combines early XDP acceleration with traversal of the TCP/IP stack up to the tc layer, enabling kernel-level QoS enforcement.

Figure 5 illustrates the packet journey within the UPF implementations, highlighting the interaction of the aforementioned deployment modes with the Linux kernel's networking stack, with a particular focus on how the eBPF-accelerated modes differ from the traditional in-kernel processing path (i.e., Simple-Switch), which serves as the baseline reference. Therefore, understanding the packet journey within the Simple-Switch scenario, provides the context needed to appreciate how the eBPF-based modes alter and accelerate packet handling.

*a) **Simple-Switch Scenario**:* Prior to packet reception, the NIC driver allocates an *sk_buff*, configures the Rx Ring Buffer, and notifies the NIC of its *head* and *tail* pointers. The NIC then fetches available descriptors via Direct Memory Access (DMA) and awaits incoming traffic.

Upon arrival, the packet is captured in the NIC's Rx queue and DMA-copied [46] into the next available sk_buff. The buffer address is recorded at the Ring Buffer's tail, marking the descriptor as *used*. This marks the packet's first copy. Next, a Hardware Interrupt Request (IRQ) (HardIRQ) is raised, prompting the CPU to run the NIC's Interrupt Service Routine (ISR), which moves the packet to the FIFO poll queue [47] and triggers a Software IRQ (SoftIRQ). This transitions processing to the kernel, where *net_rx_action()* forwards the packet to the network stack for protocol-specific processing [48], traversing the tc and Netfilter layers. tc handles shaping, prioritization, and qdisc (if needed); Netfilter applies, if needed, filtering and Network Address Translation (NAT) [49]. If accepted by the IP layer, the packet undergoes header sanity and checksum verification, then proceeds to TCP/UDP [50] for L4 checks and reassembly. The Transport layer passes payloads to the socket layer via *sock_queue_rcv_skb()*, which notifies any waiting syscall. The data is copied to an *iovec struct* and delivered to user space. Replies follow the reverse path back to the NIC.

*b) **eBPF-XDP Scenario**:* In the beginning, the UPF starts with the eBPF-XDP configuration mode, and a PDU session is established after being requested by a UE. This initial state leads to the creation of an XDP-based SFC (i.e., Traffic Parser, Classifier, and Forwarder) within the NIC Driver. With the use of the XDP program, packets are intercepted as soon as they enter the NIC RX queue, before interacting with the main Linux kernel's networking stack, allocating per-packet sk_buff data structures, or being forwarded to user space.

Packet processing begins when the NIC triggers a HardIRQ, invoking the IRQ handler. This handler activates the NAPI subsystem via a SoftIRQ, starting processing via driver's poll function, which executes the XDP hook. To ensure uninterrupted processing, IRQs are disabled by the NIC driver. The XDP program triggered by the XDP hook, marks the entry point for the created XDP SFC. Starting execution, the XDP program accesses a context metadata, encapsulated within the optimized *xdp_md struct* [37] and retrieves the different packet's headers to parse. In this regard, XDP programs access persistent data structures (*eBPF maps*) via helper functions such as *bpf_map_lookup_elem()* or *bpf_map_update_elem()*. A final decision on the packet's handling (drop, retransmit, pass to kernel, or redirect) is issued. Once packet processing

is completed, the NAPI subsystem is deactivated, and IRQs from the NIC device are re-enabled.

*c) **eBPF-XDP-TC Scenario**:* In this scenario, the UPF starts with eBPF-XDP-TC mode configuration and an already established PDU session. This initial state leads to the creation of an XDP- and tc-based SFC (i.e., Traffic Parser, Classifier, Forwarder, and QoS Enforcer).

In-kernel processing begins like eBPF-XDP: the packet traverses the XDP-based SFC until the Classifier, which short-circuits the chain, issues a verdict, and returns *XDP_PASS*, forcing entry into the kernel instead of tail-calling the Forwarder. This marks the end of the XDP pipeline. Next, the NAPI subsystem is deactivated and IRQs re-enabled. The packet is DMA-copied to the next available `sk_buff`, with its address recorded at the tail of the Rx Ring Buffer, marking the corresponding buffer descriptor as *used*. From there, the packet follows the standard kernel path to the network stack (as in the Simple-Switch scenario). At the bottom, it reaches the tc layer, where the eBPF tc pipeline begins. The tc layer abstracts queuing logic as qdisc and provides *enqueue/dequeue* callbacks where the eBPF tc hooks are invoked. When the packet is enqueued into the qdisc, the enqueue callback is invoked, triggering the Ingress eBPF tc hook. This hook implements a default classifier that inspects the packet and redirects it to the outgoing interface via *bpf_redirect_neigh()*. On egress, the enqueue operation again activates the Egress tc hook, which runs sequential tc-bpf filters to classify packets; a match directs the packet to the corresponding class for further qdisc processing. The tc framework's classifier-action pipeline performs the final stage of classification and forwards the packet out of the kernel to the NIC for transmission, in accordance with the configured QERs and SDFs.

D. UPF Data-Path Layer: eBPF-XDP

We now focus on the eBPF-XDP scenario. Since only PDRs and FARs are involved, the Data-Path is entirely handled by XDP programs seeking performance. The resulting SFC includes a Traffic Parser, Classifier, and Forwarder—each implemented as a multi-stage XDP program capable of issuing early verdicts. Together, they form a staged processing XDP pipeline as shown in Figure 6. To further optimize processing, we distinguish between uplink and downlink traffic: uplink packets encapsulated in GTP and downlink packets in plain IP. Accordingly, the Data-Path is split into two specialized pipelines—one dedicated to uplink (attached to N3 interface), called `xdp_handle_uplink`, and the other to downlink (N6 interface), named `xdp_handle_downlink`. This separation eliminates repeated conditional checks within a unified pipeline, significantly reducing the instruction footprint per packet and lowering CPU cycles per path.

After the XDP hook on N3 (N6, respectively) is triggered, the corresponding uplink (downlink, respectively) pipeline begins at the Traffic Parser. The parser extracts the packet from `xdp_md`, inspects the L3 protocol, and passes non-IP packets to the kernel. For IP packets, and in case of uplink, the transport header is checked-only UDP with destination port 2152 (GTP) continues in the pipeline. The GTP header is parsed to confirm it carries a G-PDU (`GTPU_G_PDU`), after which the GTP extension header is used to extract the `QFI`, followed by the inner IP header. The source IP (ip_src) is retrieved, and used as a key in the `session_mappings` map, which returns the session ID `SEID` and tunnel ID `TEID`. In the downlink case, the packet is plain IP (no UDP or GTP headers), so only the destination IP is extracted and used in the same map. If no match is found, the packet is dropped. The Classifier then accesses to `session_pdrs` map and retrieves the PDRs sorted by precedence. Each PDR's PDI-composed of UE IP, TEID, QFI, and interface—is matched against values extracted from the packet: in the uplink, ip_src, TEID, QFI, and ACCESS interface; in the downlink, destination IP and CORE interface. The first match determines the active PDR; if none is found, the packet is dropped. Once a PDR is matched, the Forwarder fetches its corresponding FAR from the map `rules_match_pdr`. The FAR dictates the forwarding action: removing the GTP header and redirecting to the CORE interface (N6) in uplink, or GTP-encapsulating the packet and redirecting to the ACCESS interface (N3) in downlink. Before executing `bpf_redirect_map`, the MAC addresses are updated to ensure correct delivery.

E. UPF Data-Path Layer: eBPF-XDP-TC

This section describes the UPF architecture under the eBPF-XDP-TC scenario, an extension of the eBPF-XDP framework that enables QoS enforcement via the tc layer. Unlike the baseline XDP—only pipeline, which redirects packets directly to N3 or N6 after FAR processing, the tc-enabled design hands selected packets to the kernel's traffic control stack for shaping. To support this, a new stage -QoS Enforcer- is introduced into the SFC (Parser, Classifier, Forwarder), enabling fine-grained policy enforcement. Figure 7 illustrates the full Data-Path.

While 3GPP specifications support QoS enforcement in both directions, uplink QoS enforcement within the UPF typically requires the N9 interface, which is beyond the scope of this work; therefore, our focus is on downlink. To balance performance and traffic shaping on the downlink, the pipeline is split into two branches: `xdp_handle_downlink` described previously, where packets are forwarded immediately after FAR processing using XDP only, and another derived from this latter (called `xdp_handle_shaping`), where packets undergo additional kernel-level processing via tc for hierarchical traffic shaping and QoS enforcement. This design acknowledges the moderate overhead introduced by SKB handling, classification, and queuing in the kernel.

To sum up, the UPF architecture design implements three distinct pipelines: one uplink pipeline attached to the N3 interface (`xdp_handle_uplink`), which remains unchanged across both eBPF-XDP and eBPF-XDP-TC scenarios; and two separate downlink pipelines: `xdp_handle_downlink` for the eBPF-XDP scenario focused on performance, and `xdp_handle_shaping` for the eBPF-XDP-TC scenario enabling QoS enforcement. The choice between the downlink pipelines to be attached to the N6 interface, is made dynamically based on the `enable_qos` flag in the configuration file.

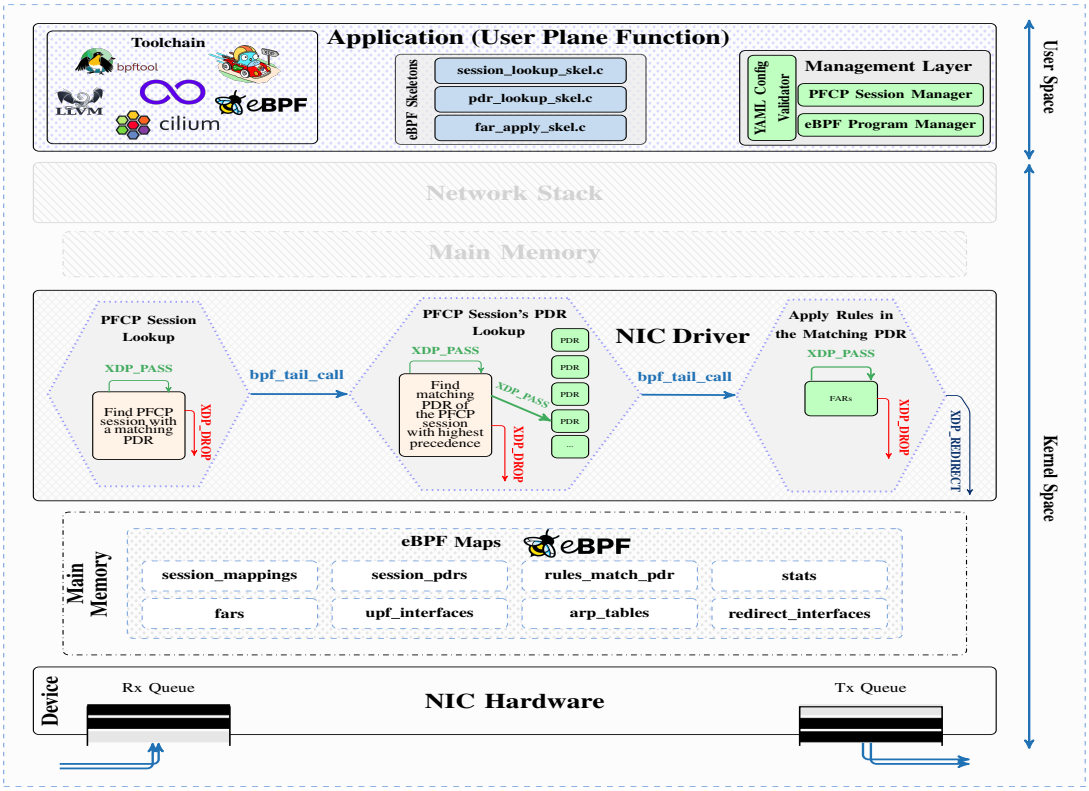

Fig. 6: **UPF Architecture: View on the XDP Pipeline**

When QoS is enabled, packets processed by the Forwarder are tail-called to the QoS Enforcer, which passes them to the tc ingress hook on the N6 interface. From there, traffic is redirected to the N3 egress where HTB-based shaping applies flow-specific QoS policies. Additionally, the framework extends the set of available eBPF maps by introducing new ones specific to the eBPF-XDP-TC mode, supporting the additional requirements of classification and shaping.

Embarking on its journey through the Data-Path, the packet arrives at the Classifier stage—its path so far mirroring the eBPF-XDP process, with the Parser and Classifier having already classified traffic and retrieved session context. Now focused on the downlink, the Classifier accesses the `session_pdrs` map and retrieves PDRs sorted by precedence. Each PDR's PDI—comprising the UE IP, TEID, QFI, and interface—is matched against the packet's destination IP and the CORE interface. If `enable_qos` is set, an additional check is performed: the SDF field in the PDI is used to fetch a flow description from the `sdf_filter` map, which is then matched against the packet's five-tuple (protocol, source and destination IP addresses along with source and destination ports). The first PDR satisfying all these criteria is selected; if no such match is found, the packet is dropped. Once the active PDR is determined, the packet continues its journey to the Forwarder, which retrieves the associated FAR from the `rules_match_pdr` map to determine the forwarding action-typically GTP encapsulation in the downlink. Rather than redirecting the packet for transmission, the Forwarder invokes the QoS Enforcer, which fetches the corresponding QER from the same `rules_match_pdr` map. The QoS Enforcer spans both the XDP and tc layers, each contributing to complementary aspects of QoS control. At the XDP level, it is represented by the QER application logic, which enforces gate status, retrieves some metadata, and applies packet marking early in the pipeline. At the tc layer, it encompasses the full shaping subsystem: a lightweight *ingress* stage on the N6 interface and a more elaborate HTB-based configuration on the N3 *egress*. This dual-level enforcement ensures that admission control and precise traffic shaping are applied coherently across the processing chain: early filtering and gate control take place at the XDP layer, after which packets are handed off to the tc subsystem for final shaping and transmission, achieving end-to-end QoS enforcement.

F. UPF Data-Path Layer: HTB & qdisc Implementation

Building upon our previous treatment of the UPF downlink Data-Path (i.e., eBPF-XDP-TC), where the packet embarks on its initial journey through the XDP eBPF program for early classification and metadata tagging, we now delve deeper into the next stage of the packet's quest: the QoS enforcement subsystem within the tc layer.

Having passed the first gate, the packet is steered toward the kernel's traffic control layer, where a dual-subsystem tc framework—strategically attached at distinct hook points—governs

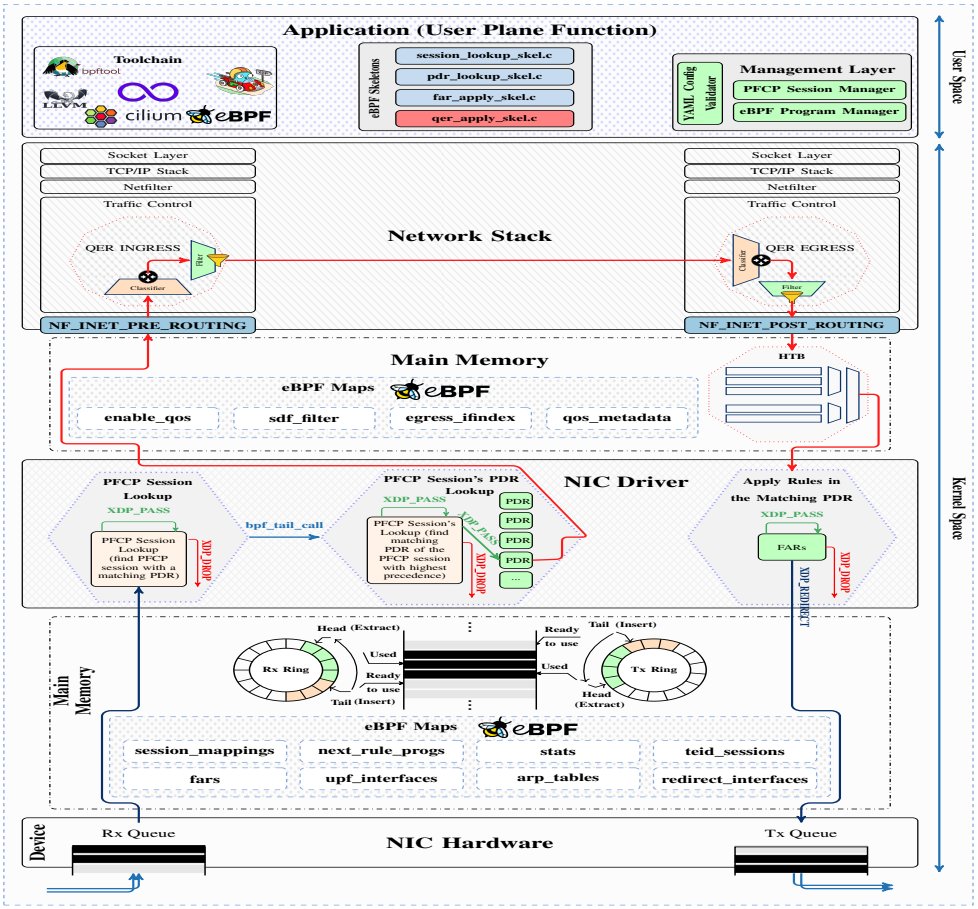

Fig. 7: **UPF Architecture: View on the XDP-TC Pipelines**

the next stage of its journey. The first subsystem resides at the ingress hook of the N6 interface, while the second is anchored at the egress hook of the N3 interface. Together, they support the enforcement of downlink QoS policies. The ingress side, designed for minimal overhead, leverages a simple `pfifo_fast` qdisc and a lightweight classifier. Its role is limited to filtering and redirecting eligible packets (the IP packets embedding QoS metadata created at the XDP layer) toward the egress subsystem on N3 interface. Serving a different role, the egress subsystem on N3 handles the actual shaping logic using HTB QDISC and organized hierarchically as follows: For each new PDU session, a dedicated QDISC is instantiated as a child of the root. Within this QDISC, each QoS flow is represented as a distinct child class. At the leaves of the hierarchy, a simple PFIFO QDISC is typically used. However, for the default QoS flow, a nested HTB QDISC is inserted within the corresponding class, and a PFIFO QDISC is attached at its leaf. This structure allows us to constrain the bandwidth available to the default flow by setting rate and ceil on the intermediate class, thereby limiting how much unused capacity it may borrow from its parent. Figure 8 illustrates this hierarchy. The rate and ceil parameters of each class reflect the GBR and Maximum Bit Rate (MBR) of the associated QoS flow, while class priority encodes the PDR's precedence. The burst and ceil burst (cburst) values are tuned based on the QoS flow type (Non-GBR, GBR, or Delay-Critical GBR). To direct traffic accurately within this hierarchical structure, a *tc-bpf* classifier is attached to the HTB QDISC as a filter. Its role is to intercept GTP-U packets and assign them to the correct class or nested QDISC using the QoS metadata injected earlier at the XDP stage. This metadata includes the Session ID (SEID) and the QoS Flow ID (QFI), which together form a unique key used to identify the appropriate node in the class hierarchy. This organization is captured in the Data Model (see Schema 1), which outlines the complete structure of the tc framework for the UPF. It builds upon the foundational concepts introduced in Section IV.

Schema 1 formalizes the complete `tc` configuration as a context-free grammar. Rule 1 defines the top-level `tc` object, composed of the five building blocks introduced in Section IV. Rules 2–7 specify qdisc types: a *root qdisc* (rule 4) anchors the hierarchy at a given interface with `handle 1:0`, while *leaf qdiscs* (rule 5) are attached at the leaves of the class tree. Rules 8–9 define classes, each identified by a `major:minor` classid; the major part is inherited from the parent qdisc (SEID), and the minor part encodes the QoS Flow ID (QFI). Rule 10 defines filters that steer GTP-U packets to the appropriate class using QoS metadata injected at the XDP stage. Rules 11–12 encode policing and shaping parameters, directly mapping 3GPP QoS attributes: `rate` ← GBR,

`ceil` ← MBR, and `prio` ← |255 − PDR precedence|. Rule 13 constrains all numeric types to ensure compatibility with the Linux `tc` API.

Schema 1 tc Representation/Grammar

```
1: tc: {
       qdiscs, classes, filters, policing, shaping
   }
2: qdiscs: qdisc [, qdiscs]
3: qdisc: root_qdisc | leaf_qdisc
4: root_qdisc : {
       role: root
       dev: interface
       handle: 1:0
       type: classful_qdisc | classless_qdisc
       default: default_class_id
       shaping: r2q
   }
5: leaf_qdisc : {
       role: leaf
       parent: classid
       type: classless_qdisc
       (optional) parameters
   }
6: classful_qdisc : {
       scheduler: HTB | CBQ,
       hook: egress
   }
7: classless_qdisc : {
       scheduler: PFIFO | FQ_CODEL| PFIFO_FAST | TBF | SFQ,
       hook: ingress | egress
   }
8: classes: class [, classes]
9: class: {
       role: root | parent | child | leaf
       classid: major:minor
       parent: root | another_class
       qdisc: (optional) attached qdisc at leaf,
       shaping: rate, ceil, burst, cburst, prio
   }
10: filters : {
       protocol: ipv4 | ipv6 | all
       match: selectors (e.g., IP src, dst, port, DSCP, etc.)
       action: redirect | drop | reclassify | police | remark,
       target: classid (to redirect matched packets)
   }
11: policing: {
       rate: policer_rate
       burst: policer_burst
       action: drop | remark | reclassify
   }
12: shaping: {
       rate: gbr,
       ceil: mbr,
       burst: ceil/rate,
       cburst: ceil + burst,
       r2q: 1 (recommended),
       prio: |255 − pdr_precedence|
   }
13: parameters: {
       gbr: const (u32) in kbps,
       mbr: const (u32) in kbps,
       precedence: const (u16),
       policer_rate: const (u32),
       policer_burst: const (u32)
   }
```

VI. METHODOLOGY AND TESTBED

Now, we present the comprehensive methodology and testbed infrastructure used in our work. We describe the implementation, the hosts configuration, the test setup, and the deployment scenarios.

A. Implementation and Artifact Availability

The OAI-UPF is developed in C++ for the Management Layer and in restricted C for the Data-Path Layer. The build process uses CMake [51] and Make [52], with version control managed through Git [53]. The project is open source and publicly accessible on the EURECOM GitLab [54], mirrored on GitHub [55], licensed under the Collaborative Software Source License (CSSL) v1.0 [56], and released in successive versions each adding specific features. A Docker [57] image (`oaisoftwarealliance/oai-upf`) is also published on Docker Hub [58] with several tags, to ease adoption and reproduction. The Data-Path Layer leverages a toolchain comprising libbpf [59], bpftool [60], libnl [61], Clang ≥ 3.4 [62], LLVM ≥ 3.7 [63], kernel headers ≥ 5.4 [64], and libelf [65], and supports both native and generic XDP attachment. The End-to-End (E2E) tests are scripted in Python.

The measurement dataset, the analysis scripts, the complete test configurations, and the traces of the control experiment of Section VII are archived on Zenodo [66], guaranteeing reproducibility.

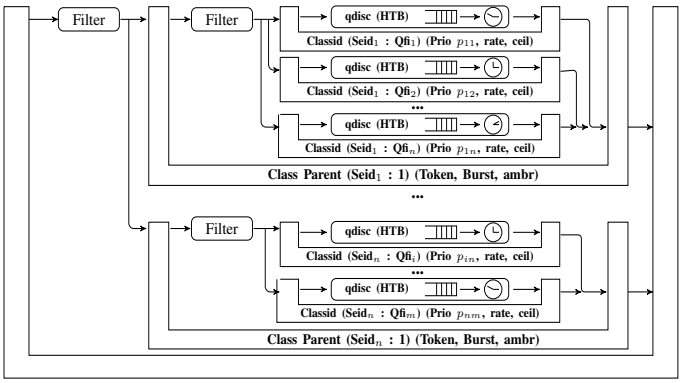

Fig. 8: **UPF Architecture: HTB qdisc Implementation**

B. Experimental Setup

We are interested in subjecting our solution to rigorous stress testing to measure its resilience, using RFC 2544-like tests [67]. These tests are designed to evaluate the throughput, latency, and overall performance of the solution under various conditions. The System Under Test (SUT) consists of a *loopback* between two devices: one hosting a traffic generator such as TRex, and the other, the Device Under Test (DUT), hosting the solution to test, in our case OAI-UPF. TRex [30] is an open-source realistic traffic generator powered by DPDK, used to measure the maximum sustainable throughput of a DUT. To contextualize the performance of our solution, we also compare it against other state-of-the-art open-source UPF implementations, including free5GC, Open5GS, UPG-VPP, and eUPF. Each UPF that could be instantiated on the testbed is evaluated under identical test conditions to ensure fairness and reproducibility; UPG-VPP, which could not be operated with the available NICs, is compared separately in a virtualized environment.

C. Experimental Platform

To evaluate the performance of each UPF, we employed a testbed comprising two identical physical hosts. One host was dedicated to running the TRex traffic generator, while the other

hosted the UPF under test-including the OAI-UPF and the aforementioned open-source implementations for comparative evaluation. Both machines operated on Ubuntu 22.04 with Linux kernel version 6.2, selected for its compatibility with eBPF and DPDK technologies. Each system was equipped with an AMD EPYC 9534 64-core processor with two threads per core, running at a base clock of $2.45\,\mathrm{GHz}$ with an all-core boost of $3.55\,\mathrm{GHz}$, and supported by $256\,\mathrm{MB}$ of L3 cache [68]. For high-speed networking, the original on-board Intel 1 GbE adapters were upgraded with high-performance NICs: the TRex host was tuned with dual-port Mellanox ConnectX-5 Ex (MT27800 family) NICs, each port providing up to $100\,\mathrm{Gbps}$, while the UPF host was fitted with dual Intel E810 NICs supporting up to $100\,\mathrm{Gbps}$ per port.

D. Hardware Limitations

The Intel E810 NICs used on the UPF host are known to exhibit an XDP TX buffer leak under sustained load, originating in the `ice` driver rather than in user code or in the UPF implementations under test. The issue is documented by Red Hat [69] and acknowledged in the Intel community forum [70], and primarily affects small-packet workloads (64 and 104 bytes), where the per-queue packet rate is highest and TX buffer pressure most pronounced. At a fixed line rate, smaller packets translate into a proportionally larger number of packets per second distributed across RX/TX queues; conversely, larger packets (e.g., $1460/1500\,\mathrm{bytes}$) lower the per-queue packet rate, relieving pressure on both the NIC and the CPU cores. This is why drops remain visible at large packet sizes but are markedly less severe than at small ones. We caution the reader that anomalies observed at small packet sizes in Section VII should be interpreted in this light, rather than as limitations of the UPF implementations themselves. To isolate the effect, we ran a control experiment in which the UPF pipeline was bypassed entirely via a bare `XDP_REDIRECT` between N6 and N3; the same drop pattern persisted, confirming the driver-level origin.

E. Test Cases

Our test framework mimics the behavior of the PDU session establishment procedures and generates corresponding uplink and downlink traffic. This approach ensures that the DUT (e.g., OAI-UPF) handles realistic session management and data forwarding scenarios as it would in a live 5G network. We scripted Python test cases for both uplink and downlink scenarios, leveraging TRex APIs [71].

- *Uplink*: We generate UDP traffic in TRex, encapsulated within GTP headers, to simulate a UE sending data via one of the dual-port Mellanox adapters to the DUT. On the DUT, the selected UPF processes the traffic, *decapsulates* the GTP headers, and forwards it to the N6 interface. The traffic is looped back to TRex on the second port, simulating a DN.
- *Downlink*: In the downlink scenario, the simulated DN, represented by TRex, generates UDP traffic and sends it to the DUT (i.e., UPF) via the N6 interface. The DUT *encapsulates* the traffic within GTP and forwards the resulting packets to the end-user (i.e., TRex) over the N3 interface through the other port of the Mellanox NIC, completing the E2E delivery process.
- *QoS Enforcement*: We simulate several UEs, each with a single PDU session containing multiple PDRs, FARs, and QERs, creating a variety of QoS flows (Non-GBR, GBR, and Delay-Critical GBR). We use tools like `iperf3` and `ping` to generate various traffic types across the different UEs and DN endpoints. This test case demonstrates the UPF's ability to *properly* enforce the QoS.

F. Measurement Protocol

- *Forwarding-performance campaign*. Each configuration of the parameter grid runs for 600 seconds, with throughput, CPU utilization and Packet Loss Rate (PLR) captured once per second; the first $60\,\mathrm{s}$ are discarded as warm-up and the reported value is the mean over the remaining 540 samples. This is an order of magnitude above the $60\,\mathrm{s}$ minimum recommended by RFC 2544 [67], and the margin is deliberate: the `ice`-driver TX buffer leak of Section VI-D, the thermal and Dynamic Voltage and Frequency Scaling (DVFS) transient of the EPYC 9534, and the ten-second `irqbalance` rebalancing cycle [72], [73] all evolve on timescales that a shorter trial would either miss or sample in transient. The campaign comprises 1408 configurations — 896 for throughput and PLR (7 packet sizes $\times$ 16 queue counts $\times$ 4 implementations $\times$ 2 directions) and 512 for the CPU-load heatmaps — corresponding to more than 230 hours of continuously generated traffic.

- *QoS-enforcement campaign*. The two QoS scenarios require a different protocol, since the quantities of interest are per-flow rather than aggregate. Throughput is measured with per-flow `iperf3` sessions and latency with `ping` probes, both issued from the external data network at their default one-second intervals over runs of approximately $1000\,\mathrm{s}$. These samples are not reduced to a single mean: throughput is reported as a time series, since the object is MBR compliance throughout the run, and Round-Trip Time (RTT) as an empirical distribution. In the inter-UE scenario the duration is imposed by the population cycle: ten reference UEs over the first $100\,\mathrm{s}$, forty background UEs attaching in batches of five at $10\,\mathrm{s}$ intervals, a plateau at fifty, and a symmetric staggered departure.

VII. Performance Evaluation

We now present the results of our measurement campaign, focusing on throughput, CPU utilization, and PLR. The evaluation covers a wide range of configurations by systematically varying three key parameters: (i) the packet injection rate f_i, (ii) the packet size s_p, and (iii) the number of Rx queues c_j. Specifically, f_i (in Mpps; millions of packets per second) takes values from $\{0.1, 0.2, ..., 0.9\} \cup \{1, 2, ..., 9\} \cup \{10, 20, 30, ..., 140\}$; s_p (in bytes) takes values in $\{64, 104\} \cup \{250, 500, ..., 1250\} \cup \{1460, 1500\}$; and c_j takes values from $\{2, 4, 6, ..., 30, 32\}$. We used the `irqbalance` service to distribute incoming traffic evenly across the active Rx queues, with each Rx queue pinned to a single CPU core to optimize performance. Note that while

the figures in this paper are expressed in terms of CPU cores, the underlying XDP performance primarily depends on the number of active Rx queues, since each queue is directly mapped to a CPU core.

Since UPG-VPP cannot operate with the Mellanox and Intel E810 NICs in our setup, comparisons are limited to the proposed solution, eUPF, free5GC UPF, and Open5GS UPF under the $100\,\mathrm{Gbps}$ configuration. In a separate virtualized environment, the proposed solution outperforms UPG-VPP, achieving $0.938\,\mathrm{Mpps}$ versus $0.640\,\mathrm{Mpps}$ on uplink and $0.952\,\mathrm{Mpps}$ versus $0.632\,\mathrm{Mpps}$ on the downlink for UDP traffic with $1400-\mathrm{byte}$ packets. The following analysis thus focuses on the $100\,\mathrm{Gbps}$ NIC results.

A. Throughput

Figure 9 reports the *peak* throughput (in Mpps) for the downlink and uplink scenarios as a function of the number of Rx queues c_j and packet size s_p. The main measurements use packet sizes from 250 to 1250 bytes in steps of $250\,\mathrm{bytes}$ for both uplink and downlink. Figure 10 depicts the edge cases with packet sizes of $64/104\,\mathrm{bytes}$ (minimum) and $1460/1500\,\mathrm{bytes}$ (maximum) for downlink/uplink, with uplink sizes accounting for the $40-\mathrm{byte}$ GTP header.

From Figures 9 and 10 it can be observed the following:

a) Uplink versus Downlink: Across all configurations, uplink and downlink exhibit broadly similar throughput trends, with performance steadily increasing until stabilizing close to the $100\,\mathrm{Gbps}$ theoretical ceiling. This behavior is consistent for OAI-UPF and eUPF. In contrast, free5GC shows a severe throughput degradation in the uplink ($\simeq 0.36\,\mathrm{Mpps}$), likely due to limitations in its Data-Path implementation and less efficient handling of encapsulated traffic. Open5GS stands out differently: its throughput remains almost flat in both directions ($\simeq 0.17\,\mathrm{Mpps}$), suggesting structural bottlenecks independent of traffic direction. While both directions ultimately converge toward the same throughput, uplink tends to scale more slowly than downlink. This is partly intuitive—uplink packets carry the extra $40-\mathrm{byte}$ GTP header, reducing the number of packets per second that can be transmitted at line rate. But beyond header overhead, uplink requires operations similar in complexity to downlink: multiple map lookups (for GTP tunnel validation, and FAR retrieval), along with a crucial step of packet pointer redirection during decapsulation. These steps, while not heavier than their downlink counterparts, introduce enough per-packet cost that uplink only "catches up" to downlink at larger packet sizes, where header overhead becomes proportionally smaller.

b) Packet Size Impact: The theoretical throughput or packet rate, expressed in Mpps, decreases with increasing packet size, as described in Equation 1:

$$Throughput\,[\mathrm{Mpps}] = \frac{Bandwidth\,[\mathrm{Gbps}] \times 10^3}{Packet\,size\,[\mathrm{bytes}] \times 8} \qquad (1)$$

This equation 1 provides the *theoretical maximum* packet rate achievable for a given bandwidth and packet size, ignoring practical limitations such as system overhead, NIC behavior, or kernel processing.

With a fixed $100\,\mathrm{Gbps}$ line rate, smaller packets lead to higher packet rates (Mpps), while larger packets reduce the number of packets per second (according to Equation 1), even though the link remains fully saturated. This inverse proportionality is clearly observed for OAI-UPF and eUPF. In contrast, Open5GS and free5GC are largely unaffected: their throughput is very low ($\leq 0.17\,\mathrm{Mpps}$ and ≤ 5 Mpps, respectively), so any dependence on packet size is negligible or not visible, as the packet rates are too small to show a measurable impact.

Across the main experimental range (250–$1250\,\mathrm{bytes}$), OAI-UPF and eUPF consistently follow this pattern, with packet rates decreasing as packet size increases. At the lower edge cases ($64/104\,\mathrm{bytes}$), OAI-UPF and eUPF—both based on XDP—should in principle deliver significantly higher throughput, but the measured rates fall short. As discussed in Section VI-D (Hardware Limitations), this shortfall is attributable to the Intel E810 `ice`-driver TX buffer leak, not to the UPF implementations themselves.

c) Rx Queues / CPU Cores Impact: Except for Open5GS, which is single-threaded and thus insensitive to parallelism, all other UPFs benefit from additional Rx queues. With each queue pinned to a dedicated CPU core, efficient kernel- or user-space fast paths enable multi-core scaling. OAI-UPF and eUPF exploit XDP, enabling early packet processing in the NIC driver and distributing workloads across cores with minimal overhead. free5GC, relying on the gtp5g kernel module, also gains from multiple queues but scales less efficiently than DPDK or XDP. By contrast, Open5GS supports neither these technologies nor multi-threading, explaining its flat performance. Among scalable UPFs, OAI-UPF is the most sensitive to queue parallelism, followed by eUPF, while free5GC shows more modest improvements.

B. CPU Load

Figure 11 shows the average CPU usage (in %) as a function of the packet injection rate f_i (in Mpps) and the number of Rx queues c_j across the UPF implementations.

The expected, idealized behavior of CPU load can be approximated as proportional to the packet-rate-per-worker:

$$L(f_i, c_j) \propto \frac{f_i}{c_j}, \qquad (2)$$

where $L(f_i, c_j)$ is the CPU load, f_i the packet injection rate, and c_j the number of Rx queues (or worker CPUs). Thus, for fixed f_i, CPU load should decrease roughly inversely with c_j. Conversely, for fixed c_j, increasing f_i should increase L. Along diagonals where f_i and c_j grow proportionally, the ratio f_i/c_j remains roughly constant, so L should stay nearly unchanged. This simple scaling law provides a baseline expectation for CPU usage. In the heatmap, this relationship translates into clear color transitions: *horizontally*, from dark on the left (few queues, high load) to light on the right (many queues, low load); *vertically*, from light at the bottom (low injection rates) to dark at the top (high injection rates); and *diagonally*, into iso-load contours of nearly uniform color where f_i/c_j is nearly constant. Deviations from these expected

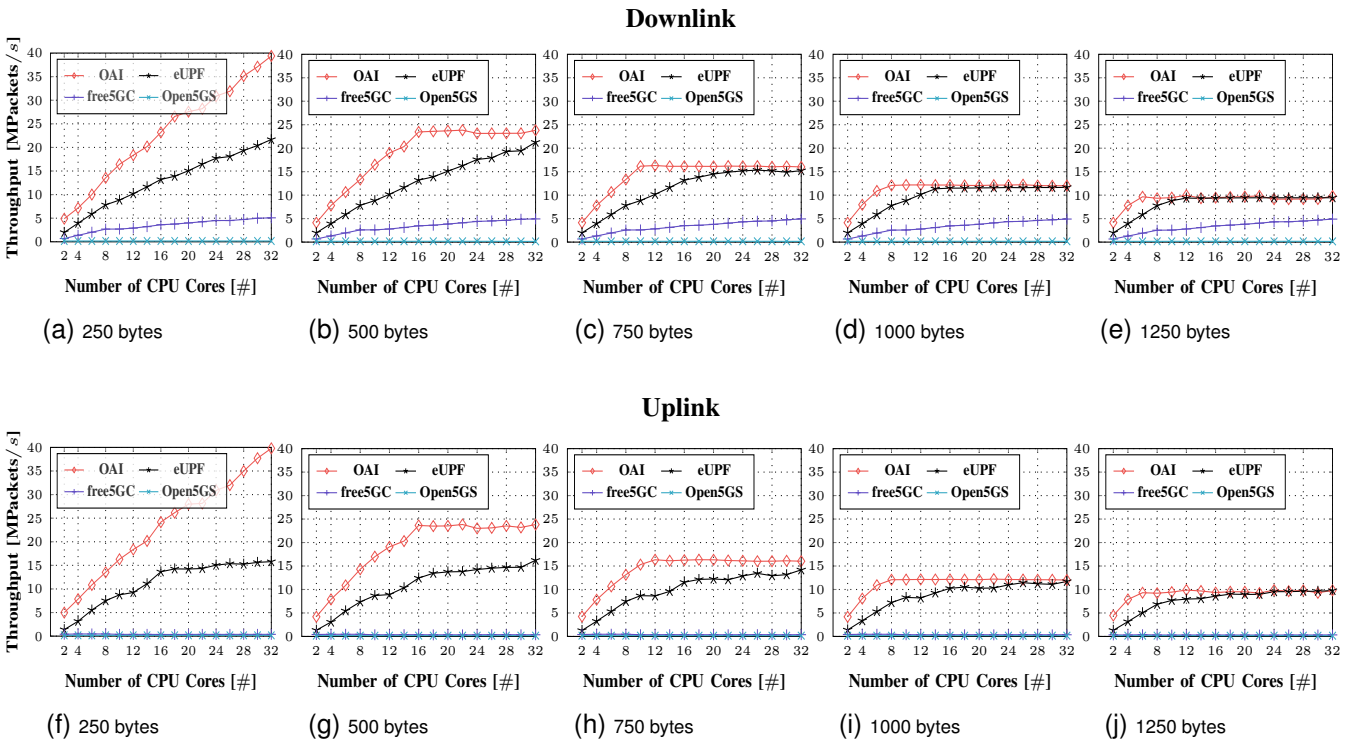

Fig. 9: **Throughput Obtained on the Downlink and Uplink for each UPF, as a Function of Packet Size and CPU Cores Number**

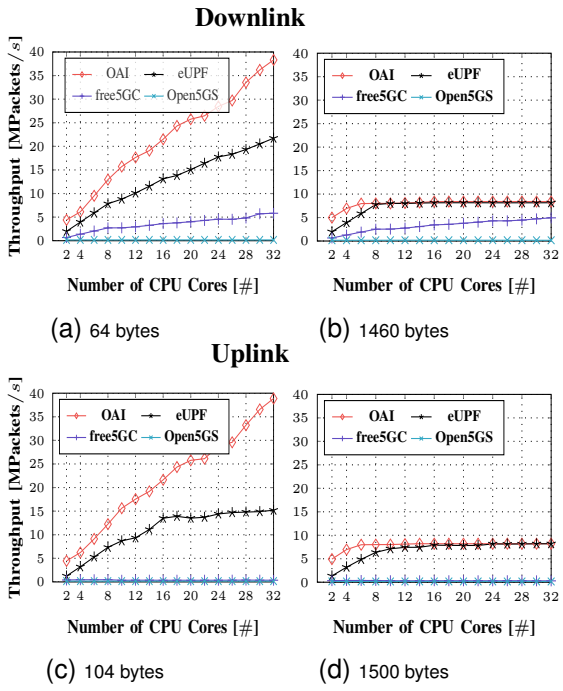

Fig. 10: **Throughput Obtained on the Downlink and Uplink for each UPF, as a Function of Packet Size and CPU Cores Number (Edge Cases)**

patterns, such as dark regions or disrupted diagonals, point to implementation inefficiencies or NIC/driver bottlenecks.

Turning to the results, OAI-UPF stands out: its heatmap contains large white or near-white regions, indicating CPU usage below 40% even at high injection rates. This is especially clear in the subdomains:

$$S_1 = \int_{f_i=1}^{14} \int_{c_j=4}^{32} dc_j\, df_i, \quad S_2 = \int_{f_i=12}^{40} \int_{c_j=16}^{32} dc_j\, df_i,$$

which remain nearly white throughout (CPU load $< 40\%$). Since the full figure spans 40×32, these two domains (i.e., S_1 and S_2) alone cover a substantial fraction of the total area, implying that more than half of the OAI-UPF heatmap corresponds to low CPU load. This highlights OAI-UPF's efficiency and scalability in contrast to the other implementations.

By comparison, the other UPFs show *predominantly* dark regions, pointing to persistent CPU bottlenecks. eUPF follows a pattern similar to OAI-UPF, but more pronounced: CPU load decreases with c_j and increases with f_i, converging faster to lower utilization. Some anomalies remain, attributable to the Intel E810 `ice`-driver issue (Section VI-D); they are visible in eUPF and, to a lesser extent, in OAI-UPF, which handles more traffic and converges more gracefully. This stems from OAI's design, which separates uplink from downlink paths—reducing CPU footprint—while eUPF relies on ARP lookups, involving costly packet copies and kernel interactions. Open5GS, in contrast, exhibits a measured flat profile: 100% of a single core, independent of traffic, reflecting its lack of multithreading or acceleration. free5GC is more nuanced but inefficient: for $f_i < 1\,\text{Mpps}$, load drops from nearly 100% with one or two CPUs to below 20% as c_j grows. Yet as f_i increases (2–6 Mpps), load climbs steeply, saturating around $6\,\text{Mpps}$ even with more CPUs. Theoretically, the heatmap should display smooth gradients-horizontal, vertical, and diagonal iso-load contours; however, free5GC breaks this pattern, revealing architectural inefficiencies and poor parallelization. UPG-VPP is not included in Figure 11. As noted

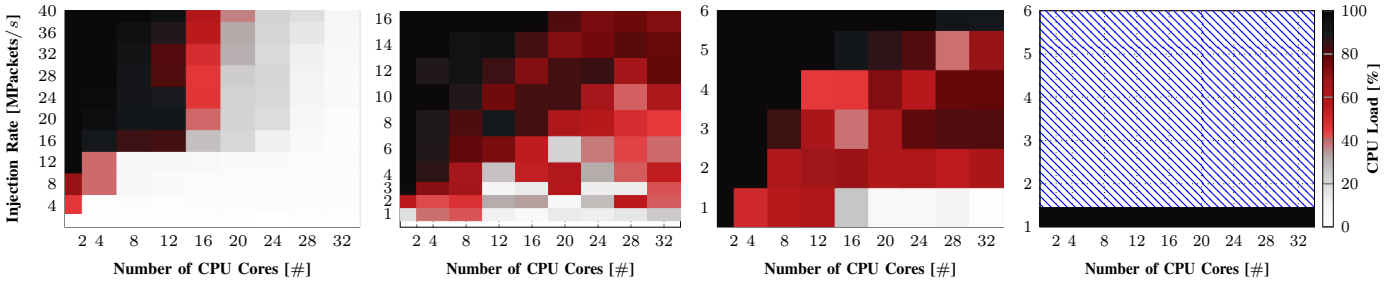

Fig. 11: **Heatmap CPU Usage (Downlink, UDP: 250 bytes)**

in Section VI-B, it could not be brought up on our 100 Gbps testbed, so no CPU-load samples were collected for it. Its behavior is nonetheless well established in literature and worth stating here for contrast: DPDK-based data planes run their workers as poll-mode drivers that busy-spin on the NIC receive queues, so each assigned core is fully occupied whether or not packets are present. CPU utilization is therefore pinned near 100% and invariant in both f_i and c_j [9], [10], [18]. We report this as a known property of the DPDK execution model rather than as an experimental result of this work; it is the structural counterpoint to the kernel-native designs evaluated above, which consume CPU in proportion to offered load.

C. Packet Loss Rate

Figure 12 presents the PLR (in percentage) as a function of packet size (s_p) and Rx queue count (c_j) for the downlink scenario, where the system's full bandwidth ($100\,\text{Gbps}$) is utilized. For each s_p (7 in total), 16 PLR values corresponding to 16 different c_j values are plotted, providing s_p-dependent and c_j-dependent global measures for each UPF implementation. To sum up, we obtain 16×7 values per UPF, providing a rich comparison that is analyzed in the following observations.

- *(i)* For OAI-UPF, eUPF, and free5G (the latter appearing exaggerated due to the y-axis starting at 95% for better visualization), the PLR values span the figure surface, indicating that both packet size and Rx queue count significantly influence PLR. In contrast, Open5GS shows all c_j-dependent values collapsed and s_p-dependent values aligned along $y = 99.86\%$, indicating that neither packet size nor queue count impacts PLR.
- *(ii)* free5GC exhibits moderate variations in PLR across both c_j and s_p, though the range is bounded between 95% and 100%. The distribution of points appears relatively uniform across both axes, suggesting an evenly distributed sensitivity to packet size and queue count.
- *(iii)* OAI and eUPF show high variability, with PLR spanning the full range from 0% to 100%, showing strong sensitivity to both c_j and s_p. A closer look indicates that OAI samples tend to converge faster toward lower PLR values, highlighting better performance compared to eUPF. This trend is particularly evident for intermediate packet sizes ($500 - 1250\,\text{bytes}$).
- *(iv)* Comparing the PLR distribution across intervals provides further insight: for OAI, a large portion of values (48 out of 102) fall below 5% PLR, while very few values exceed 80% (11 in total; 8 correspond to packet size $64\,\text{bytes}$, 2 to $250\,\text{bytes}$, and 1 to $500\,\text{bytes}$). In contrast, eUPF shows a more spread-out distribution with a smaller fraction of very low PLR values (only 9 out of 102 below 5%) and more samples in the higher PLR ranges (13 values each in the $[80,\ 90\%]$ and $[90,\ 100\%]$). This indicates that OAI achieves consistently lower PLR for most packet and queue configurations, whereas eUPF exhibits more variable performance with a higher PLR.
- *(v)* For small packet sizes (64 and $250\,\text{bytes}$), PLR is higher than expected for both OAI and eUPF. As discussed in Section VI-D (Hardware Limitations), this elevated PLR is largely attributable to the Intel E810 ice-driver TX buffer leak rather than to the UPFs themselves; the bare `XDP_REDIRECT` control experiment described therein confirmed this driver-level origin even at very low injection rates.

D. Fine-Grained QoS Performance Evaluation

Having previously analyzed the baseline forwarding capacity of the UPF, we now turn our attention to its fine-grained behavior under policy-driven traffic differentiation. In this section, we investigate how the UPF enforces heterogeneous QoS rules when multiple flows compete for resources. We evaluate two complementary scenarios designed to probe different aspects of 5G QoS enforcement in a realistic deployment. Together, they exercise *intra-UE traffic prioritization with multi-flow QoS*, the impact of user *mobility on QoS stability*, and *inter-UE traffic isolation under mixed service demands and resource budgets*. Each scenario captures dynamic behaviors, flow prioritization, and QoS isolation under contention, providing insights into UPF's enforcement of Service-Level Agreement (SLA) in multi-tenant environments.

Throughout this section we adopt a common notation: a QoS flow f_i is represented as a tuple $f_i\{\text{gbr}_\text{i}, \text{mbr}_\text{i}\} \cup \{sdf, port_i\}$, where gbr_i and mbr_i are the downlink guaranteed and maximum bit rates (in Mbps), sdf is the service-data-flow filter, and $port_i$ is the destination port used by the UPF to classify the flow f_i via its transport-layer header. The subnet $12.1.1.0/24$ is allocated to the UEs. Throughput is measured with per-flow `iperf3` sessions and latency with `ping` probes, both issued from the external data network

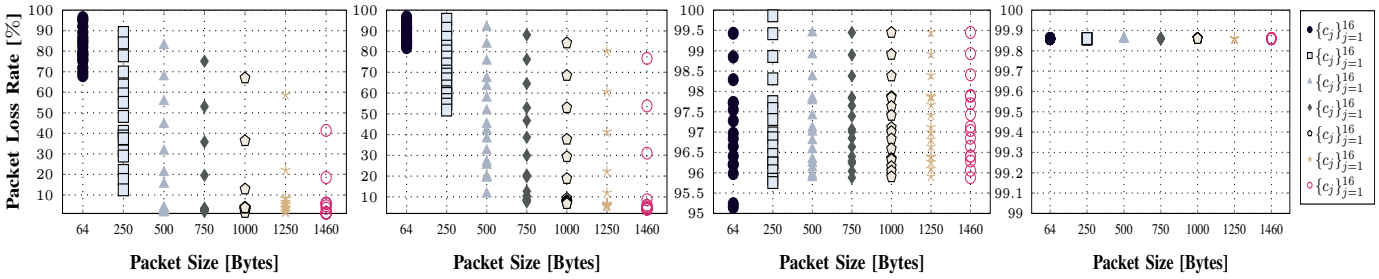

Fig. 12: **PLR per Packet Size and Number of Rx Queues**

container `oai-ext-dn`. Concretely, Scenario 1 isolates *intra-UE* behavior: a single UE simultaneously runs several flows with very different profiles, and we observe whether the UPF shapes each of them independently. Scenario 2 isolates *inter-UE* behavior under a controlled population dynamic: 50 UEs join the cell sequentially at fixed 10-second intervals, then progressively leave in the same staggered fashion. Among them, 10 reference UEs carry the measurements while the remaining 40 act as background load. We observe whether the per-UE guarantees of the reference UEs are preserved as the surrounding population builds up to full load and then drains.

1) Scenario 1: Intra-UE QoS Differentiation with Multi-QoS Flows: This scenario evaluates the UPF's ability to enforce fine-grained QoS policies when a UE simultaneously generates diverse classes of traffic. It emulates a complex user behavior—such as concurrent video streaming, file uploads, voice communication, and background synchronization—by instantiating a single UE with multiple parallel flows. Specifically, the UE runs six QoS flows: one Non-GBR flow for latency tracking (Internet Control Message Protocol (ICMP)) and five GBR flows, f_1 to f_5 with increasing bandwidth budgets, each mapped to a distinct QoS profile: $f_1\{0.512, 1\}$, $f_2\{1, 1\}$, $f_3\{9, 10\}$, $f_4\{20, 45\}$, $f_5\{50, 70\}$, all sharing the same filter sdf = `permit out tcp from any to 12.1.1.0/24`, and destination ports `5200`, `5201`, `5202`, `5203`, `5204`, respectively. Figure 13 reports per-flow throughput (Fig. 13a) and the ICMP RTT distribution (Fig. 13b).

As reported in Figure 13a, the UPF enforces accurate traffic shaping across all GBR flows (f_1–f_5), each configured with distinct bandwidth profiles. Flows f_1 and f_2 are assigned low bitrates (below 1 Mbps), with f_2 featuring equal GBR and MBR, representing bandwidth-sensitive applications with strict guarantees. Flow f_3 targets higher sustained throughput (9/10 Mbps), while f_4 and f_5 are provisioned with significantly larger budgets (20/45 Mbps and 50/70 Mbps, respectively), reflecting resource-intensive traffic. The UPF maintains per-flow compliance with specified MBRs throughout the experiment, validating its shaping logic and demonstrating precise enforcement across heterogeneous traffic. Additionally, the absence of leakage confirms strong isolation and fairness guarantees-essential for QoS assurance in multi-tenant environments.

Figure 13b presents the distribution of ICMP round-trip times (RTTs) for the Non-GBR flow under concurrent multi-flow conditions, rendered as a violin plot: the filled body is a Gaussian kernel density estimate of the measured RTT samples, and the interior carries a compact box-and-whisker annotation in which the thin vertical line indicates the observed range, the filled box delimits the IQR, and the white marker locates the median. This representation combines the full shape of the empirical distribution with the conventional five-number summary in a single, compact visual. The measurements indicate a median RTT around 0.51 ms, with minimum and maximum observed values of 0.18 ms and 1.26 ms, respectively. The Inter-Quartile Range (IQR) is 0.35 ms (spanning [0.38, 0.73] ms), and the full range covers 1.07 ms. Beyond these aggregate statistics, the density body reveals that the bulk of the probability mass concentrates inside the IQR, with a thin upper tail that decays rapidly toward the observed maximum - confirming that high-latency excursions are rare rather than merely bounded. While this flow is treated as best-effort traffic with no guaranteed quality, these values demonstrate low latency with bounded variation, indicating that the UPF can handle background traffic efficiently, even under concurrent GBR load, without excessive delay or jitter. Although Non-GBR flows are not subject to guaranteed bandwidth, in our setup the default HTB class was configured with a rate of 1024 kbit and a ceil of 2048 kbit to shape best-effort traffic.

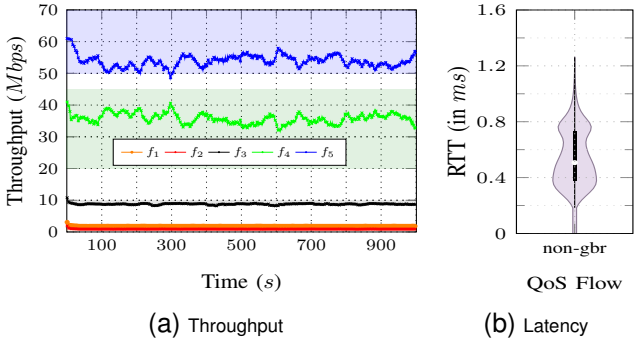

Fig. 13: **Performance Metrics: Throughput and Latency**

2) Scenario 2: Inter-UE Isolation under Population Dynamics: This scenario evaluates the UPF's ability to preserve per-UE QoS guarantees as the attached population varies. Fifty UEs join the cell sequentially at 10-second intervals and then leave in the same

staggered fashion: background UEs depart in batches of five every $10\,s$. Each UE carries two flows: a GBR flow $f\{10, 20\}$ and a non-GBR ICMP flow for latency tracking, all sharing the same filter sdf = `permit out ip from any to 12.1.1.0/24`. Figure 14 reports the per-UE downlink throughput trajectories of the ten reference UEs as small multiples (Figure 14a to Figure 14j), and Figure 15 presents an aggregate view of the same data (mean and min–max envelope)(see Figure 15(a)) overlaid on the active-UE population (see Figure 15(b))

Across both views, the dominant observation is that the per-UE service rate is invariant under population churn. In Figure 15(a), the mean reference-UE throughput remains pinned to the $20\,\mathrm{Mbps}$ MBR target with deviations of only a few kbps, and the min–max envelope stays compressed to a band narrower than $\pm 15\,\mathrm{kbps}$. The active-UE population in Figure 15(b) makes four operating regimes immediately distinguishable: a plateau at 10 active UEs ($t \in [0, 100]\,s$) corresponding to the reference UEs operating alone, a staircase ramp-up as the 40 background UEs progressively attach in batches of five every $10\,s$, a plateau at 50 active UEs corresponding to full load, and a symmetric staircase ramp-down as background UEs detach. The boundaries between these regimes are clearly visible on the population curve in panel (b), yet they leave no observable trace on the throughput curve in panel (a). The transitions between phases are clearly visible on the population curve in panel (b), yet they leave no observable trace on the throughput curve in panel (a). In other words, the addition or removal of background UEs is invisible to the reference UEs, which is the property Scenario 2 was designed to test.

The small-multiples view in Figure 14 confirms that this stability is not an artifact of averaging across UEs. Each of the ten reference UEs individually tracks its $20\,\mathrm{Mbps}$ MBR target with sub-percent variability, and the ten panels are visually indistinguishable: no UE is starved while another holds capacity, and no UE exhibits a step change at the regime transitions identified above. This per-UE consistency follows directly from the way the UPF maps PFCP state onto the Linux traffic-control hierarchy. Each UE owns a dedicated HTB leaf class parameterized by its own GBR/MBR pair, attached to the root class of the N3 interface. When new UEs attach during the arrival ramp-up, the UPF instantiates additional leaves beneath the same root without renegotiating the budgets already allocated to the existing leaves; conversely, when UEs detach, their leaves are removed without disturbing the rest of the hierarchy. The shaping decisions taken at full load (50 active UEs) are therefore identical to those taken in the reference-only regime (10 active UEs), as both figures show.

Two secondary observations are worth noting. First, the min–max envelope in Figure 15(a) does not widen during the high-churn portions of the experiment: the spread between the slowest and the fastest reference UE remains essentially constant whether the cell hosts 10 or 50 active UEs. This indicates that the UPF's per-UE isolation is not statistical (i.e., the result of averaging fluctuations across many flows) but *structural*: the HTB hierarchy enforces the same per-leaf rate regardless of how many sibling leaves coexist beneath the root. Second, the GBR floor of $10\,\mathrm{Mbps}$ is never approached: all UEs sustain their MBR throughout the experiment, including during the full-load plateau. This is consistent with the baseline forwarding capacity reported in Section VII-A (Figure 10b), where the UPF sustains $C_{pps} \approx 10\,\mathrm{MPackets/s}$ on downlink mode for MTU-bound packets of $L = 1460$ B (matching the 1500-B TCP payload generated by `iperf3`, fragmented at N6 interface). Each UE in Scenario 2 carries a single GBR flow capped at $r = 20\,\mathrm{Mbps}$, so the per-UE offered load is $R_{ue}^{(1)} = r = 20\,\mathrm{Mbps}$ and the UPF saturates around

$$N_{\max}^{(1)} = \frac{C_{pps} \cdot L}{R_{ue}^{(1)}} \approx \frac{10^7 \cdot 12 \cdot 10^3}{20 \cdot 10^6} \approx 6\,000 \text{ UEs.} \quad (3)$$

The 50 UEs of Scenario 2 therefore exercise less than 1% of the UPF Data-Path capacity in this configuration. A more demanding regime is reached when each UE multiplexes the maximum number of QoS flows allowed in commercial deployments. 3GPP TS 23.501 [4] permits up to 64 QoS flows per PDU session, and operators typically provision $q \in [8, 16]$ flows per UE [35], [74]. Taking the upper bound $q = 16$ at the same per-flow rate r, the maximum offered load per UE rises to $R_{ue}^{(16)} = q \cdot r = 320\,\mathrm{Mbps}$, and the saturation point drops to

$$N_{\max}^{(16)} = \frac{C_{pps} \cdot L}{R_{ue}^{(16)}} \approx 375 \text{ UEs.} \quad (4)$$

However, in practice the bit rates effectively provisioned per flow are far *below* our experimental setup: standardized GBR flows whose bit rates are governed by their associated 5QI value (see Section II-B). 5QIs cover bit rates from $64\,\mathrm{kbps}$ for VoNR voice (5QI 1) to a few Mbps for conversational and streamed video (5QI 2, 4) [4], while best-effort traffic is bounded by the Session-AMBR rather than per-flow limits. Aggregating a realistic operator-grade subscriber profile (one VoNR call, one conversational video, one buffered video, and a $100\,\mathrm{Mbps}$ Session-AMBR for non-GBR traffic) yields a per-UE offered load of approximately $R_{ue}^{op} \approx 115\,\mathrm{Mbps}$, and the corresponding saturation point is

$$N_{\max}^{op} = \frac{C_{pps} \cdot L}{R_{ue}^{op}} \approx 1\,000 \text{ UEs.} \quad (5)$$

What we observe in Figs. 14 and 15 is therefore the *QoS-enforcement regime* of the UPF, not its capacity ceiling. The three bounds frame the operating envelope: the per-UE invariance reported here is expected to hold up to several hundred UEs even under aggressive 16-flow worst-case provisioning, up to roughly one thousand UEs in a realistic operator deployment, and well beyond several thousand in the single-flow configuration of this scenario.

VIII. Portability and NIC Dependency

The performance results reported in Section VII depend not only on the UPF design but also on the underlying NIC and its driver, since eBPF/XDP hooks the packet path inside the driver before `skb` allocation [36]. We therefore characterize portability through four mechanisms that govern packet processing

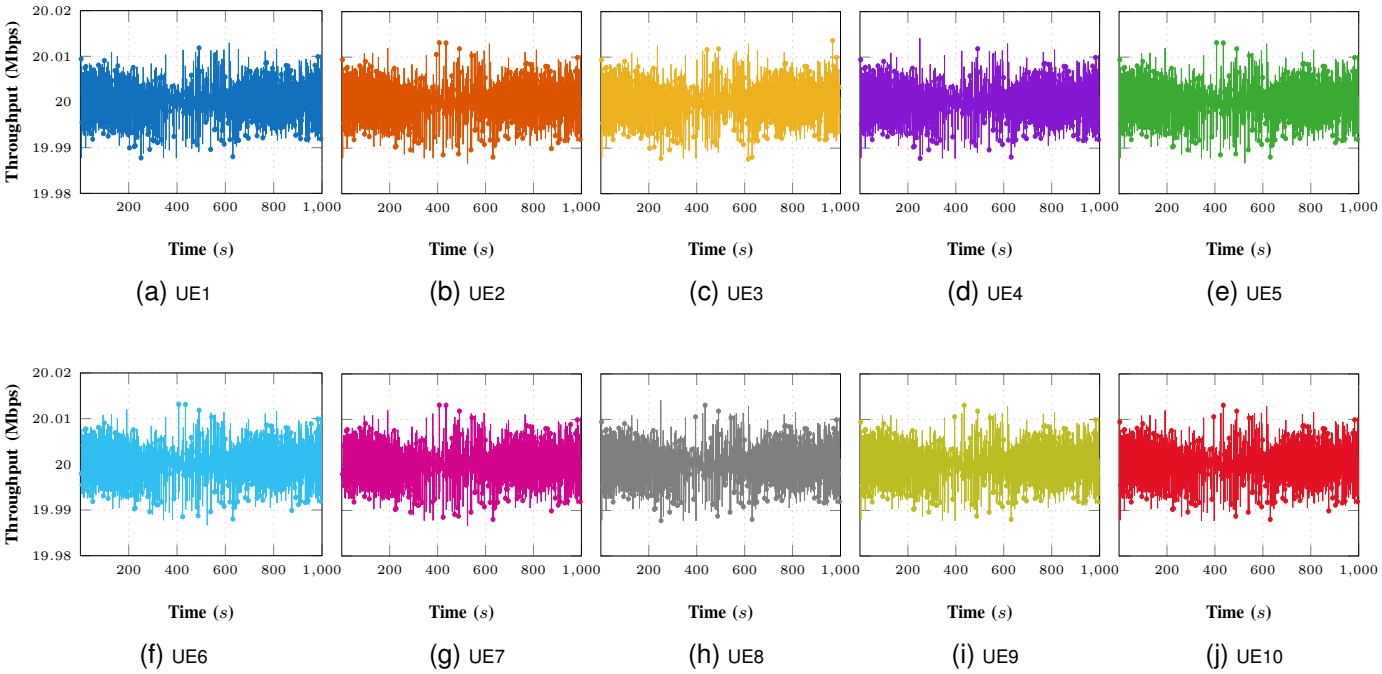

Fig. 14: **Per-UE throughput stability across the** 10 **reference UEs under population dynamics.** Each panel reports the downlink throughput of one reference UE over the entire experiment; all UEs converge to their MBR of 20 Mbps regardless of the background-UE arrival/departure activity.

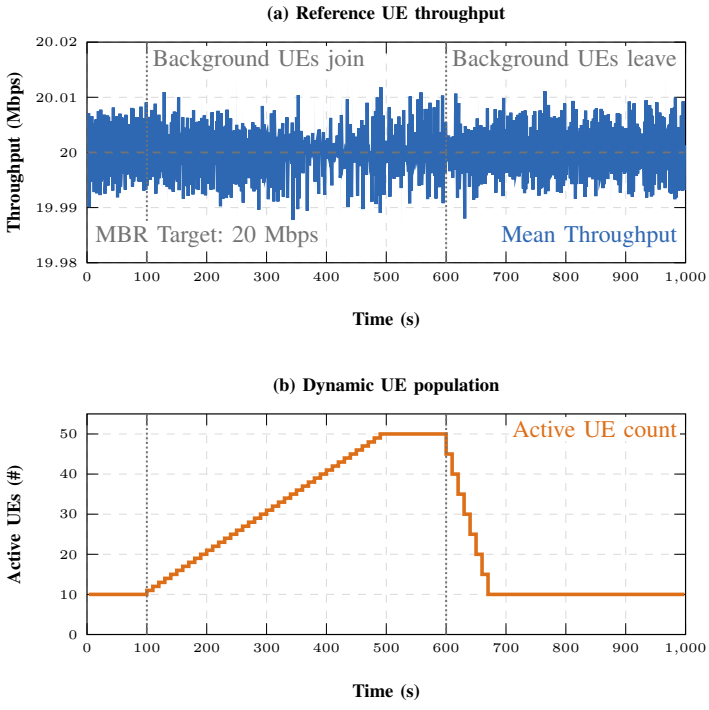

Fig. 15: **Per-UE throughput stability under population dynamics.** (a) Aggregate downlink throughput across the 10 reference UEs. (b) Number of active UEs over time. The staircase shape of the population curve makes the four operating regimes of the experiment self-evident: reference-only, arrival ramp-up, full load at 50 UEs, and symmetric departure ramp-down.

performance across the Linux NIC ecosystem, illustrated with the most relevant NIC families. Table II summarizes the corresponding support matrix.

- *Execution path.* The primary portability factor is the XDP attachment mode. When native (driver-level) XDP is available, packets are processed before `skb` allocation, eliminating a major source of per-packet overhead; performance is then bounded by queue parallelism and memory bandwidth. NVIDIA/Mellanox `mlx5` (ConnectX-5/6/7), Intel `ice` (E810, 100 GbE), Intel `ixgbe` (82599/X550), and recent Intel `igc` (i225/i226, since Linux 6.14) all expose this fast path [75], [76]. When only generic XDP is available—typical of consumer-grade adapters and some virtualized environments—the program runs *after* `skb` allocation through the `xdpgeneric` fallback [77], which incurs a throughput penalty (typically 30–60 % on commodity 10 GbE setups [37], [78]) without affecting correctness or QoS enforcement.

- *Buffer management.* A second factor is the efficiency of packet buffer handling, in particular for `XDP_REDIRECT` and AF_XDP zero-copy paths. Zero-copy eliminates data copies between kernel and user space but requires tight driver integration, and its benefit is largest at small packet sizes where per-packet overhead dominates. `mlx5` provides the most mature implementation, including multi-buffer support and striding RQ on 100 GbE [75]; Broadcom `bnxt` (NetXtreme-E) and Intel `ice` also document zero-copy support [79]. Intel `i40e` (700 Series: X710, XL710, X722) is more constrained: zero-copy has been removed from the out-of-tree driver and remains unstable in combination with multi-buffer [80].

- *Queue scalability.* Throughput scaling across CPU cores depends on the number of hardware RX/TX queues exposed by the NIC and their mapping to cores. NICs with few queues may constrain parallelism even when native XDP is supported, an effect that is independent of the eBPF program and reflects hardware design trade-offs. The NICs listed above all expose multi-queue configurations sufficient for the 100 Gbps regime

TABLE II: **XDP/eBPF support matrix for major NIC families. "Native" refers to the driver-mode XDP hook (before `skb` allocation); "ZC" denotes AF_XDP zero-copy. ✓ = full support, ✗ = not supported, ❍ = partial or unstable. Status is based on upstream Linux $\geq$ 6.2 and vendor documentation as of 2025–2026.**

Driver	Adapter family	Native	Redirect	ZC	Offload	Notes
Mature native XDP with full `XDP_REDIRECT` and AF_XDP zero-copy						
`mlx5`	NVIDIA ConnectX-5/6/7	✓	✓	✓	✗	Reference XDP implementation [75]; the only mature stack supporting multi-buffer + ZC + striding RQ on 100 GbE.
`ixgbe`	Intel 82599/X550	✓	✓	✓	✗	Long-standing support since Linux 4.10 [77].
`igb`	Intel Gigabit (I210/I350)	✓	✓	✓	✗	ZC since Linux 6.14 [76].
`bnxt`	Broadcom NetXtreme-E	✓	✓	✓	✗	Documented support for both XDP and AF_XDP [79].
Native XDP with caveats						
`ice`	Intel E810 (100 GbE)	✓	✓	✓	✗	AF_XDP zero-copy supported per Intel feature matrix [82] and kernel documentation [83]; stable under workloads $\leq$ 30 Gbps; watchdog and TX-leak issues reported beyond that [69].
`i40e`	Intel 700 Series	✓	✓	❍	✗	ZC removed from out-of-tree driver [80]; multi-buffer + ZC currently unstable.
Hardware offload and virtualized guests						
`nfp`	Netronome Agilio	✓	✓	✓	✓	The only upstream driver supporting hardware offload [81].
`virtio_net`	KVM/QEMU guest	✓	✓	✗	✗	Native XDP only; ZC available via `vhost-net` on the host.

Generic mode (`xdpgeneric`, after `skb` allocation): functional fallback for any Linux NIC, lower performance.

considered in this paper; lower-end adapters trade queue count for cost and would saturate at correspondingly lower rates.

- *Driver maturity*. Beyond nominal feature support, driver maturity affects stability under sustained load. Reported issues on Intel `ice` (E810) include NETDEV WATCHDOG transmit-queue timeouts and TX buffer leaks under sustained XDP traffic [69], [70]—the same `XDP_TX` leak we encountered in Section VI-D, which is specific to `ice` and does not affect `mlx5`. Such limitations do not affect correctness but may reduce achievable throughput under stress; we recommend kernel and driver versions explicitly acknowledged as fixed by the vendor.

- *Implications for this work*. The proposed UPF relies only on standard XDP capabilities and `XDP_REDIRECT`, with no dependency on NIC-specific features. It is therefore expected to operate correctly on any Linux-supported NIC. On platforms with mature native XDP and efficient zero-copy, performance should match or exceed the results reported here; on systems limited to generic XDP or with fewer queues, functionality is preserved at a predictable throughput cost. We deliberately avoid hardware offload (`XDP_OFFLOAD`), which would yield marginal additional gains but remains restricted to Netronome SmartNICs in upstream Linux [37], [81] and would tie the UPF to a single vendor.

IX. Conclusion and Perspectives

This work highlights the potential of eBPF as a foundational technology for next-generation mobile core data planes. Our implementation demonstrates that eBPF can deliver both high performance and fine-grained programmability while preserving compatibility with 3GPP specifications—qualities increasingly critical in edge-centric and cloud-native 5G deployments. Beyond its current role, we see eBPF enabling more dynamic, per-flow control and tighter integration with telemetry, service function chaining, and AI-driven optimization. As the mobile ecosystem moves toward 6G, embracing micro-UPFs, in-network computing, and zero-touch automation, eBPF stands out as a key enabler for evolving the user plane into a more intelligent, adaptive, and efficient substrate.

The convergence with AI/ML is particularly promising. The fine-grained observability natively exposed by eBPF maps—per-flow counters, latency histograms, queue occupancy—provides a low-overhead telemetry substrate that can feed online learning pipelines without instrumenting the Data-Path externally. Conversely, ML-derived policies such as traffic classification, anomaly detection, or adaptive QoS profiles can be pushed back into the kernel as updated eBPF programs or map entries at runtime, closing the loop between inference and enforcement at packet-processing timescales. This tight coupling is especially relevant for 6G use cases such as intent-based networking, predictive QoS for Vehicle-to-Everything (V2X) and Extended Reality (XR) services, and autonomous slice management, where control decisions must keep pace with the data plane itself. Future work will explore this direction by extending the proposed UPF with in-kernel telemetry hooks and lightweight ML inference offloaded directly to eBPF programs.

X. Acknowledgement

This work is supported by three research projects: the National Key Research and Development Program of China (Grant 2024YFE0200300), the IPCEI ME/CT project (BPIFrance Grant No. DOS0239248/00), and the Franco-German STIC5G project (BPIFrance Grant No. DOS0209312/00).

References

[1] S. Kar, P. Mishra, and K.-C. Wang, "Efficient resource management using 5G multi-connectivity for high throughput and reliable low latency communication," *EURASIP Journal on Wireless Communications and Networking*, vol. 2025, no. 58, 2025.

[2] T. K. Vu, M. Bennis, M. Debbah, M. Latva-aho, and C. S. Hong, "Ultra-reliable communication in 5G mmWave networks: A risk-sensitive approach," *IEEE Communications Letters*, vol. 22, no. 4, pp. 708–711, Apr. 2018.

[3] 3GPP, "Architecture enhancements for control and user plane separation of EPC nodes; stage 2," 3GPP, Tech. Rep. TS 23.214 V15.1.0, Rel. 15, Dec 2017.

[4] ——, "System architecture for the 5G system (5GS); stage 2," 3GPP, Tech. Rep. TS 23.501 V17.9.0, Rel. 17, Mar 2023.

[5] ——, "Procedures for the 5G system (5GS)," 3GPP, Tech. Rep. TS 23.502 V16.7.0, Rel. 16, Jan 2021.

[6] ——, "Interface between the control plane and the user plane nodes," 3GPP, Tech. Rep. TS 29.244 V16.5.0, Rel. 16, Nov 2020.

[7] Open5GS Community, "Open Source Implementation for 5G Core and EPC," https://open5gs.org, accessed: Jul. 2026.

[8] free5GC, a Linux Foundation Project, "Open Source Core Network Implementation," https://free5gc.org, accessed: Jul. 2026.

[9] DPDK Project and The Linux Foundation, "Data Plane Development Kit (DPDK)," https://www.dpdk.org, accessed: Jul. 2026.

[10] J. Hwang, K. K. Ramakrishnan, and T. Wood, "NetVM: High performance and flexible networking using virtualization on commodity platforms," in *Proc. 11th USENIX Symp. on Networked Systems Design and Implementation (NSDI)*, Seattle, WA, USA, Apr. 2014, pp. 445–458.

[11] Travelping, "UPG: User Plane Gateway Based on VPP," https://github.com/travelping/upg-vpp, accessed: Jul. 2026.

[12] Tigera, "eBPF Explained: Use Cases, Concepts, and Architecture," https://www.tigera.io/learn/guides/ebpf/, accessed: Jul. 2026.

[13] J. Gallego-Madrid, R. Sanchez-Iborra, and A. Skarmeta Gomez, "eBPF and XDP technologies as enablers for ultra-fast and programmable next-gen network infrastructures," in *Resource Management in Distributed Systems*, ser. Studies in Big Data. Springer, 2024, vol. 151.

[14] A. Kaufmann, S. Peter, N. K. Sharma, T. Anderson, and A. Krishnamurthy, "High performance packet processing with FlexNIC," *SIGARCH Computer Architecture News*, vol. 44, no. 2, pp. 67–81, Mar. 2016.

[15] NetFPGA Project, "NetFPGA Platform," https://netfpga.org/, accessed: Jul. 2026.

[16] E. Kfoury, S. Choueiri, A. Mazloum, A. AlSabeh, J. Gomez, and J. Crichigno, "A comprehensive survey on SmartNICs: Architectures, development models, applications, and research directions," *IEEE Access*, vol. 12, pp. 107 297–107 336, 2024.

[17] D. Firestone, A. Putnam, S. Mundkur, D. Chiou, A. Dabagh, M. Andrewartha, H. Angepat, V. Bhanu, A. Caulfield, E. Chung, H. K. Chandrappa, S. Chaturmohta, M. Humphrey, J. Lavier, N. Lam, F. Liu, K. Ovtcharov, J. Padhye, G. Popuri, S. Raindel, T. Sapre, M. Shaw, G. Silva, M. Sivakumar, N. Srivastava, A. Verma, Q. Zuhair, D. Bansal, D. Burger, K. Vaid, D. A. Maltz, and A. Greenberg, "Azure accelerated networking: SmartNICs in the public cloud," in *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*. Renton, WA: USENIX Association, Apr. 2018, pp. 51–66.

[18] D. Cerović, V. D. Piccolo, A. Amamou, K. Haddadou, and G. Pujolle, "Fast packet processing: A survey," *IEEE Communications Surveys & Tutorials*, vol. 20, no. 4, pp. 3645–3676, 2018.

[19] Edgecom LLC, "eUPF," https://github.com/edgecomllc/eupf, accessed: Jul. 2026.

[20] J. Zhou, Z. Ma, W. Tu, X. Qiu, J. Duan, Z. Li, Q. Li, X. Zhang, and W. Li, "Cable: A framework for accelerating 5G UPF based on eBPF," *Computer Networks*, vol. 222, p. 109535, 2023.

[21] S. Miano, F. Risso, M. Vásquez Bernal, M. Bertrone, and Y. Lu, "A framework for eBPF-based network functions in an era of microservices," *IEEE Transactions on Network and Service Management*, vol. 18, no. 1, pp. 133–151, Mar. 2021.

[22] D. Soldani, P. Nahi, H. Bour, S. Jafarizadeh, M. F. Soliman, L. Di Giovanna, F. Monaco, G. Ognibene, and F. Risso, "eBPF: A new approach to cloud-native observability, networking and security for current (5G) and future mobile networks (6G and beyond)," *IEEE Access*, vol. 11, pp. 57 174–57 202, 2023.

[23] C. Scheich, M. Corici, H. Buhr, and T. Magedanz, "Performance analysis of a 5G user plane function accelerated with eXpress Data Path in Docker containers," in *Proc. IEEE Future Networks World Forum (FNWF)*, 11 2023, pp. 1–6.

[24] Open Networking Foundation, "SD-Fabric Documentation," https://docs.sd-fabric.org/master/index.html, accessed: Jul. 2026.

[25] OMEC Project, "OMEC User Plane Function," https://github.com/omec-project/upf, accessed: Jul. 2026.

[26] ——, "UP4," https://github.com/omec-project/up4, accessed: Jul. 2026.

[27] ——, "Berkeley Extensible Software Switch (BESS)," https://github.com/omec-project/bess, accessed: Jul. 2026.

[28] Z. Wen and G. Yan, "HiP4-UPF: Towards high-performance comprehensive 5G user plane function on P4 programmable switches," in *Proceedings of the 2024 USENIX Annual Technical Conference (ATC)*. Santa Clara, CA, USA: USENIX Association, 2024, pp. 303–320. [Online]. Available: https://www.usenix.org/conference/atc24/presentation/wen

[29] T.-H. Wang, M.-C. Hu, L.-H. Yen, and C.-C. Tseng, "Heterogeneous UPF integration framework and 5G user plane acceleration," in *2023 24th Asia-Pacific Network Operations and Management Symposium (APNOMS)*, 2023, pp. 148–153.

[30] Cisco, "TRex Traffic Generator," https://trex-tgn.cisco.com/, accessed: Jul. 2026.

[31] ETSI, "GPRS tunnelling protocol (GTP) across the Gn and Gp interface," European Telecommunications Standards Institute, Tech. Rep. 3GPP TS 29.060 version 16.0.0 Release 16, 2020.

[32] 3GPP, "General packet radio system (GPRS) tunnelling protocol user plane (GTPv1-U)," 3GPP, Tech. Rep. TS 29.281 V16.2.0, Rel. 16, Apr 2021.

[33] A. Binsahaq, T. R. Sheltami, and K. Salah, "A survey on autonomic provisioning and management of QoS in SDN networks," *IEEE Access*, vol. 7, pp. 73 384–73 435, 2019.

[34] M. Karakuş and A. Durresi, "Quality of service (QoS) in software defined networking (SDN): A survey," *Journal of Network and Computer Applications*, vol. 80, pp. 200–218, 2017.

[35] 3GPP, "Policy and charging control framework for the 5G system (5GS); stage 2," 3GPP, Tech. Rep. TS 23.503 V18.10.0, Rel. 18, Mar 2025.

[36] T. Høiland-Jørgensen, J. D. Brouer, D. Borkmann, J. Fastabend, T. Herbert, D. Ahern, and D. Miller, "The eXpress data path: Fast programmable packet processing in the operating system kernel," in *Proc. 14th Int. Conf. Emerging Networking Experiments and Technologies (CoNEXT)*, Heraklion, Greece, Dec. 2018, pp. 54–66.

[37] M. A. M. Vieira, M. S. Castanho, R. D. G. Pacífico, E. R. S. Santos, E. P. M. C. Júnior, and L. F. M. Vieira, "Fast packet processing with eBPF and XDP: Concepts, code, challenges and applications," *ACM Comput. Surv.*, vol. 53, no. 1, pp. 16:1–16:36, 2020.

[38] B. Gregg, "BPF internals," in *Proc. USENIX Ann. Tech. Conf.* USENIX Association, Jun. 2021.

[39] N. Hedam, "eBPF — from a programmer's perspective," EasyChair Preprint 5198, 2025.

[40] B. Gregg, *BPF Performance Tools: Linux System and Application Observability*. Addison-Wesley Professional, 2019.

[41] The Linux Kernel, "NAPI," Linux Kernel Documentation, accessed: Jul. 2026. [Online]. Available: https://docs.kernel.org/networking/napi.html

[42] J. D. Brouer, "XDP — eXpress data path: An in-kernel network fast-path: A technology overview," Presented at Driving-IT, Denmark, Nov. 2017.

[43] W. Almesberger, "Linux network traffic control — implementation overview," EPFL ICA, École Polytechnique Fédérale de Lausanne, Tech. Rep., July 2001, technical Report.

[44] B. Hubert, "Linux advanced routing & traffic control," in *Proceedings of the Linux Symposium / OLS 2002*, 2002, pp. 213–222.

[45] man7.org, "tc(8) — Linux Manual Page," accessed: Jul. 2026. [Online]. Available: https://man7.org/linux/man-pages/man8/tc.8.html

[46] J. Corbet and A. Rubini, *Linux Device Drivers*, 2nd ed. O'Reilly, 2001.

[47] P. Emmerich, D. Raumer, A. Beifuß, L. Erlacher, F. Wohlfart, T. M. Runge, S. Gallenmüller, and G. Carle, "Optimizing latency and CPU load in packet processing systems," in *2015 International Symposium on Performance Evaluation of Computer and Telecommunication Systems (SPECTS)*. IEEE, 2015, pp. 1–8.

[48] R. Chen and G. Sun, "A survey of kernel-bypass techniques in network stack," in *Proceedings of the 2018 2nd International Conference on Computer Science and Artificial Intelligence*, 2018, pp. 474–477.

[49] The Netfilter Project, "The netfilter.org Project," https://www.netfilter.org/, accessed: Jul. 2026.

[50] G. R. Wright and W. R. Stevens, *TCP/IP Illustrated, Volume 2: The Implementation*. Addison-Wesley Professional, 1995.

[51] Kitware, "CMake," https://cmake.org, accessed: Jul. 2026.

[52] Free Software Foundation, "GNU Make," https://www.gnu.org/software/make/, accessed: Jul. 2026.

[53] Git Project, "Git," https://git-scm.com, accessed: Jul. 2026.

[54] OpenAirInterface Software Alliance, "OAI 5G Core Network — User Plane Function," https://gitlab.eurecom.fr/oai/cn5g/oai-cn5g-upf, accessed: Jul. 2026.

[55] ——, "OAI 5G Core Network — User Plane Function (GitHub mirror)," https://github.com/openairinterface/oai-cn5g-upf, accessed: Jul. 2026.

[56] ——, "OAI Community Software Source License," https://openairinterface.org/oai-cssl/, accessed: Jul. 2026.

[57] Docker, Inc., “Docker,” https://www.docker.com, accessed: Jul. 2026.
[58] OpenAirInterface Software Alliance, “oai-upf — Docker Hub Repository,” https://hub.docker.com/r/oaisoftwarealliance/oai-upf, accessed: Jul. 2026.
[59] libbpf Project, “libbpf,” https://github.com/libbpf/libbpf, accessed: Jul. 2026.
[60] ——, “bpftool,” https://github.com/libbpf/bpftool, accessed: Jul. 2026.
[61] T. Graf, “libnl — Netlink Protocol Library Suite,” https://www.infradead.org/~tgr/libnl/, accessed: Jul. 2026.
[62] LLVM Project, “Clang: a C Language Family Frontend for LLVM,” https://clang.llvm.org, accessed: Jul. 2026.
[63] ——, “The LLVM Compiler Infrastructure,” https://llvm.org, accessed: Jul. 2026.
[64] The Linux Kernel Organization, “The Linux Kernel Archives,” https://www.kernel.org, accessed: Jul. 2026.
[65] Sourceware, “elfutils (libelf),” https://sourceware.org/elfutils/, accessed: Jul. 2026.
[66] F. Messaoudi, L. Wang, A. Mekrache, A. Ksentini, B. Li, J. Su, S. Messaoudi, and S. El Ghalbzouri, “5G UPF Measurement Dataset: Comparative Forwarding Performance of OAI-UPF, eUPF, free5GC, and Open5GS, and QoS Enforcement of the eBPF/XDP-Based OAI-UPF,” 2026. [Online]. Available: https://doi.org/10.5281/zenodo.21693514
[67] S. Bradner and J. McQuaid, “Benchmarking methodology for network interconnect devices,” RFC 2544, RFC Editor, Mar. 1999, 31 pages.
[68] AMD, “AMD EPYC™ 9534 — product specifications,” accessed: Jul. 2026. [Online]. Available: https://www.amd.com/en/products/processors/server/epyc/4th-generation-9004-and-8004-series/amd-epyc-9534.html
[69] Red Hat, “XDP issues on Intel E810 NICs and the `ice` driver,” solution 7020411; covers the XDP TX buffer leak, NETDEV watchdog timeouts, and TX-queue breakage. Accessed: Jul. 2026. [Online]. Available: https://access.redhat.com/solutions/7020411
[70] Intel Community Forum, “XDP issues on Intel E810 and the `ice` driver,” thread 1617789. Accessed: Jul. 2026. [Online]. Available: https://community.intel.com/t5/Ethernet-Products/XDP-issues-on-Intel-E810-and-ice-driver/td-p/1617789
[71] Cisco, “TRex Stateless API Reference,” https://trex-tgn.cisco.com/trex/doc/cp_stl_docs/api/index.html, accessed: Jul. 2026.
[72] Ubuntu Manpages, “irqbalance(1) — Distribute Hardware Interrupts across Processors on a Multiprocessor System,” accessed: Jul. 2026. [Online]. Available: https://manpages.ubuntu.com/manpages/jammy/man1/irqbalance.1.html
[73] Oracle Linux Blog, “irqbalance: Design and internals,” Oracle Linux Blog, accessed: Jul. 2026. [Online]. Available: https://blogs.oracle.com/linux/irqbalance-design-and-internals
[74] GSMA, “Generic network slice template,” Permanent Reference Document NG.116 v8.0, 2023.
[75] N. Shankarappa, “Accelerating with XDP over Mellanox ConnectX NICs,” NVIDIA Technical Blog, 2022, accessed: Jul. 2026. [Online]. Available: https://developer.nvidia.com/blog/accelerating-with-xdp-over-mellanox-connectx-nics/
[76] M. Larabel, “Intel gigabit ethernet driver to speed-up with AF_XDP zero-copy for Linux 6.14,” Phoronix, 2025, accessed: Jul. 2026. [Online]. Available: https://www.phoronix.com/news/IntelIGB-AF-XDP-Zero-Copy
[77] J. D. Brouer and T. Høiland-Jørgensen, “Using the eXpress data path (XDP) in Red Hat Enterprise Linux 8,” Red Hat Blog, 2019, accessed: Jul. 2026. [Online]. Available: https://www.redhat.com/en/blog/using-express-data-path-xdp-red-hat-enterprise-linux-8
[78] J. Mostafa, S. Chilingaryan, and A. Kopmann, “Are kernel drivers ready for accelerated packet processing using AF_XDP?” in *2023 IEEE Conf. on Network Function Virtualization and Software Defined Networks (NFV-SDN)*. IEEE, 2023, pp. 117–122.
[79] Broadcom, “XDP and AF_XDP,” Broadcom Ethernet NIC Documentation, accessed: Jul. 2026. [Online]. Available: https://techdocs.broadcom.com/us/en/storage-and-ethernet-connectivity/ethernet-nic-controllers/bcm957xxx/adapters/introduction/features/xdp-and-af_xdp.html
[80] Intel Community Forum, “X710 XDP packet corruption issue (DRV_MODE, zero copy + multi buffer),” message 1725384; Intel-confirmed removal of AF_XDP zero-copy from the out-of-tree `i40e` driver. Accessed: Jul. 2026. [Online]. Available: https://community.intel.com/t5/Ethernet-Products/X710-XDP-Packet-Corruption-Issue-DRV-MODE-Zero-Copy-Multi-Buffer/m-p/1725384
[81] Netronome, “Agilio® smartnics do the work, so your CPUs don’t have to,” Netronome Technical Documentation, accessed: Jul. 2026. [Online]. Available: https://netronome.com/agilio-smartnics/
[82] Intel Corporation, “Intel® Ethernet Controller E810 feature support matrix,” document 630155. Accessed: Jul. 2026. [Online]. Available: https://www.intel.com/content/www/us/en/content-details/630155/
[83] The Linux Kernel, “Linux base driver for the Intel® Ethernet Controller 800 Series (`ice`),” Linux Kernel Documentation, accessed: Jul. 2026. [Online]. Available: https://docs.kernel.org/networking/device_drivers/ethernet/intel/ice.html

APPENDIX A
LIST OF ACRONYMS

Acronym	Expansion
3GPP	3rd Generation Partnership Project
5G	5th Generation Mobile Network
5QI	5G QoS Identifier
6G	6th Generation Mobile Network
AI	Artificial Intelligence
AMBR	Aggregate Maximum Bit Rate
AMF	Access and Mobility Management Function
AN	Access Network
API	Application Programming Interface
ARP	Address Resolution Protocol
ARP	Allocation and Retention Priority
ASIC	Application-Specific Integrated Circuit
BAR	Buffer Action Rule
BESS	Berkeley Extensible Software Switch
CBQ	Class-Based Queuing
CN	Core Network
CoT	Class of Traffic
CPU	Central Processing Unit
CSSL	Collaborative Standards Software License
DMA	Direct Memory Access
DN	Data Network
DNN	Data Network Name
DPDK	Data Plane Development Kit
DRB	Data Radio Bearer
DUT	Device Under Test
DVFS	Dynamic Voltage and Frequency Scaling
E2E	End-to-End
eBPF	extended Berkeley Packet Filter
ELF	Executable and Linkable Format
EPC	Evolved Packet Core
ETSI	European Telecommunications Standards Institute
FAR	Forwarding Action Rule
FIFO	First-In, First-Out
FQ	Fair Queuing
GBR	Guaranteed Bit Rate
GFBR	Guaranteed Flow Bit Rate
gNB	gNodeB
GPRS	General Packet Radio Service
GTP	GPRS Tunnelling Protocol
GTP-U	GTP User Plane
HardIRQ	Hardware Interrupt Request
HTB	Hierarchical Token Bucket
I/O	Input/Output

Acronym	Expansion
ICMP	Internet Control Message Protocol
IE	Information Element
IMS	IP Multimedia Subsystem
IoT	Internet of Things
IQR	Inter-Quartile Range
IRQ	Interrupt Request
ISR	Interrupt Service Routine
JIT	Just-In-Time
LTE	Long Term Evolution
MANO	Management and Orchestration
MBR	Maximum Bit Rate
MDBV	Maximum Data Burst Volume
MFBR	Maximum Flow Bit Rate
ML	Machine Learning
MPLR	Maximum Packet Loss Rate
MPLS	Multi-Protocol Label Switching
MTU	Maximum Transmission Unit
NAPI	New Application Programming Interface
NAT	Network Address Translation
NFV	Network Function Virtualization
NFVO	NFV Orchestrator
NIC	Network Interface Controller
NR	New Radio
OAI	OpenAirInterface
ONOS	Open Network Operating System
P4	Programming Protocol-Independent Packet Processors
PCC	Policy Charging and Control
PCF	Policy Control Function
PDB	Packet Delay Budget
PDI	Packet Detection Information
PDR	Packet Detection Rule
PDU	Packet Data Unit
PER	Packet Error Rate
PFCP	Packet Forwarding Control Protocol
PLR	Packet Loss Rate
qdisc	Queuing Discipline
QER	QoS Enforcement Rule
QFI	QoS Flow Identifier
QNC	QoS Notification Control
QoS	Quality of Service
RAN	Radio Access Network
RQA	Reflective QoS Attribute
RTT	Round-Trip Time
SBA	Service-Based Architecture
SDF	Service Data Flow
SDN	Software-Defined Networking
SEID	Session Endpoint Identifier
SFC	Service Function Chaining
SFQ	Stochastic Fair Queuing
SKB	Socket Kernel Buffer
SLA	Service-Level Agreement
SMF	Session Management Function
SoftIRQ	Software Interrupt Request
SUT	System Under Test
TBF	Token Bucket Filter
tc	Traffic Control
TEID	Tunnel Endpoint Identifier
ToS	Type of Service
UDM	Unified Data Management
UE	User Equipment
UPF	User Plane Function
UPG	User Plane Gateway
URR	Usage Reporting Rule
V2X	Vehicle-to-Everything
VLAN	Virtual Local Area Network
VNF	Virtualized Network Function
VNFM	VNF Manager
VoIP	Voice over IP
VoLTE	Voice over LTE
VoNR	Voice over NR
VPP	Vector Packet Processing
XDP	eXpress Data Path
XR	Extended Reality
YAML	YAML Ain't Markup Language

APPENDIX B
PFCP DATA MODEL

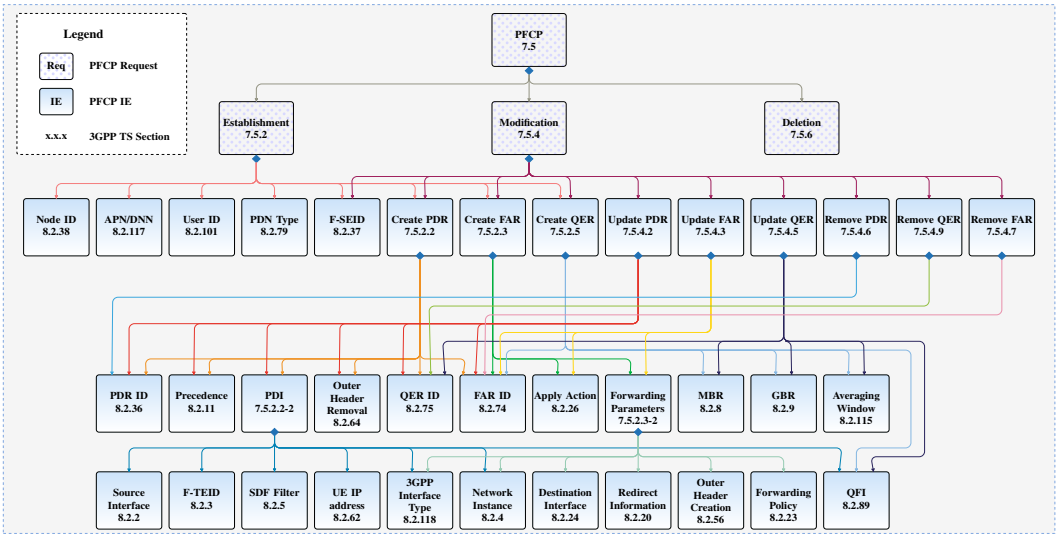

Fig. 16: **PFCP Context Data Model with IEs based on 3GPP Release 16**

Franck Messaoudi received the Engineering degree in computer science, with a specialization in networks and telecommunications, from the École Nationale Supérieure d'Informatique (ESI), Algiers, Algeria, in 2012, the M.Sc. degree in embedded systems and information processing from the University of Paris-Sud (Paris XI), France, in 2014, and the Ph.D. degree in computer science from the University of Rennes 1, France, in 2017.

He is currently a Principal Engineer with EURECOM, where he works on 5G and beyond. He previously held R&D positions with the OpenAirInterface (OAI) Alliance, contributing to the 5G core network, designing and implementing the OAI User Plane Function and its end-to-end QoS enforcement, and maintaining the project's DevOps infrastructure, and with Ekinops and b-com, on 4G and 5G network systems, covering unikernel-based network functions, SD-WAN, SDN/NFV, network slicing orchestration, and satellite-terrestrial integration for non-terrestrial networks. He has contributed to several European collaborative projects, including 5G-Transformer, 5G-EVE, 5Genesis, and SaT5G, and to the Magma and Duranta open-source projects. His research interests center on programmable data planes and in-kernel packet processing with eBPF/XDP, system optimization and performance engineering, NFV, SDN, and edge computing. A growing part of his work applies generative AI and large language models to the orchestration, automation, and management of 5G and beyond networks. He is also active in DevOps and cloud automation.

Luhan Wang is currently an Associate Professor with the Beijing University of Posts and Telecommunications (BUPT), Beijing, China. He received the B.S. degree from Shandong University, Jinan, China, in 2011, and the Ph.D. degree from BUPT in 2017. In 2016, he was a visiting Ph.D. student with EURECOM, Sophia Antipolis, France. He joined BUPT as a Lecturer in 2017 and co-founded the Joint BUPT–EURECOM Open5G Laboratory. His research interests include open-source mobile communications, cloud-native networks, semantic communications, and acceleration technologies for communications and networking. He was selected for the Beijing Nova Program in 2019 and received the Best Paper Award at the IEEE INFOCOM 2023 NG-OPERA Workshop.

Abdelkader Mekrache is a Research Scientist at Simula Metropolitan, Norway. He received the Ph.D. degree in telecommunications from Sorbonne University, France, where he conducted his doctoral research at Eurecom under the supervision of Prof. Adlen Ksentini. He also holds an engineering degree and a Master's degree in computer science from ESI, Algeria. His research interests include next-generation network management frameworks, such as Intent-Based Networking (IBN) and Zero-Touch Network and Service Management (ZSM).

Adlen Ksentini is a full professor in the Communication Systems Department at EURECOM, where he leads the Netsoft group, a team of 23 researchers dedicated to advancing network softwarization for 5G and 6G technologies. His research interests include network softwarization and cloudification, with a particular focus on applying machine learning (ML) and artificial intelligence (AI) to 5G and 6G networks. Professor Ksentini has participated in numerous H2020 and Horizon Europe projects on 5G and beyond, such as 5G!Pagoda, 5GTransformer, 5G!Drones, MonB5G, ImagineB5G, 6GBricks, 6G-Intense, Sunrise-6G, AC3, Flecon-6G, and 6G-DALI. He serves as the technical manager for Flecon-6G, 6G-Intense, and AC3, focusing on zero-touch management of 6G resources, applications, and the Cloud Edge Continuum. His work addresses both system and architectural challenges, as well as algorithmic problems in these areas, leveraging optimization algorithms and machine learning techniques. Professor Ksentini has delivered several tutorials at major IEEE international conferences, including IEEE Globecom 2015, IEEE CCNC (2017, 2018, 2023), IEEE ICC 2017, IEEE/IFIP IM 2017, and IEEE School 2019. He has received several prestigious awards, including Best Paper Awards from IEEE Globecom 2025, IEEE IWCMC 2016, IEEE ICC 2012, and ACM MSWiM 2005. In 2017, he was honored with the IEEE ComSoc Fred W. Ellersick Award for the best paper published in IEEE Communications Magazine.

Bingxuan Li received the B.S. degree in communication engineering from Beijing University of Posts and Telecommunications (BUPT) in 2022. He is currently pursuing the Ph.D. degree at the School of Information and Communication Engineering, BUPT. His current research interests include cloud-native networks, network function virtualization, and network management.

Jialei Su received the B.E. degree in communication engineering from Beijing University of Posts and Telecommunications (BUPT), Beijing, China, in 2026. He is currently pursuing the M.S. degree at the School of Information and Communication Engineering, BUPT. His research interests include 5G/6G core networks, programmable data planes, eBPF/XDP-based packets processing, cloud-native networking.

Sofiane Messaoudi received the Ph.D. degree in Communication Systems from Sorbonne University, Paris, France, with his doctoral research conducted at EURECOM. He is currently a Postdoctoral Researcher with EURECOM, where he works on the design, implementation, and experimental evaluation of next-generation mobile networks. His research interests include 5G and 6G mobile systems, data-plane programmability, software-defined networking (SDN), edge and cloud computing, network automation, and artificial intelligence for communication networks. He has contributed to the development and validation of advanced networking solutions on large-scale experimental platforms, including OpenAirInterface-based 5G infrastructures and programmable network environments. Dr. Messaoudi has been actively involved in several European research projects focusing on sustainable and intelligent communication systems. His recent work addresses programmable data-plane mechanisms, AI-driven network management, and energy-efficient mobile networking for future wireless systems.

Salim El Ghalbzouri has been pursuing the Ph.D. degree with the Communication Systems Department, EURECOM, Sophia Antipolis, France, since 2024. He was a Research Intern with Samsung Research America, Mountain View, CA, USA, in 2025, and received the M.Eng. degree in Smart Information and Communication Technology from the Institut National des Postes et Télécommunications (INPT), Rabat, Morocco, in 2024. His current research interests include network digital twins and AI for next-generation wireless systems. His earlier work addressed reconfigurable intelligent surfaces (RIS) for wireless communications.