On-Demand Slice-Aware Latency Decomposition in Cloud-Native 5G Systems

Yassir Amami
Inria, Université Côte d'Azur
Sophia Antipolis, France
yassir.amami@inria.fr

Chadi Barakat
Inria, Université Côte d'Azur
Sophia Antipolis, France
chadi.barakat@inria.fr

Thierry Turletti
Inria, Université Côte d'Azur
Sophia Antipolis, France
thierry.turletti@inria.fr

***Abstract*—Latency in cloud-native 5G systems emerges from coupled RAN, user-plane, and cloud effects, making it difficult to localize a delay increase for a given user or slice. This paper presents an on-demand, slice-aware latency measurement and decomposition framework for 5G systems. A passive mapper reconstructs UE, slice, IP address, and tunnel context from captured control-plane exchanges. A Kubernetes controller then deploys ephemeral eBPF probes at selected anchors, including the gNB and UPF, without modifying the monitored 5G functions. The probes observe existing user traffic and export per-UE, per-slice, and per-anchor latency measurements, together with statistics and operational health metrics. We validate the framework against ICMP probes and pcap-based RTT measurements. We then evaluate controlled impairments – radio degradation, UPF CPU stress, and traffic-server delay – and show that the joint analysis of latency, radio, transport, and infrastructure measurements enables accurate localization of degradation root causes across the RAN, core, and traffic server. Our monitoring remains external to the 5G stack and incurs low overhead in both container management and packet processing.**

***Index Terms*—5G; network slicing; eBPF; latency; observability**

I. Introduction

Fifth-generation (5G) systems are increasingly deployed as *cloud-native* infrastructures, where Radio Access Network (RAN) and core functions run as software components across distributed platforms. They also support network slicing, which lets operators tailor services to different application requirements. This improves flexibility, programmability, and deployment speed. That same flexibility, however, makes performance diagnosis more difficult. In particular, user-plane *latency* in 5G systems no longer reflects a single network segment. It arises from interactions across the RAN, the core, and the underlying cloud infrastructure. When delay increases, identifying where it originates and understanding how it affects individual slices becomes challenging. This is made more difficult by the limited visibility of existing monitoring solutions and the wide range of factors that can cause latency degradation, including congestion, software bugs, radio resource contention, and wireless transmission issues. Generic cloud observability tools can expose resource-level measurements from containers, pods, and hosts, but they remain blind to telecom-specific entities. At the same time, 5G-specific monitoring frameworks have made significant progress in exposing measurements from selected network functions, yet they still tend to remain confined to either the core or the RAN, rather than offering a coherent cross-domain view. More importantly, they provide limited support for *latency attribution*: they may reveal that performance has degraded, but not whether the cause lies in the radio path, the user-plane path, or the interaction between them. This missing cross-domain, slice-aware view is especially problematic in cloud-native deployments, where monitoring must keep pace with dynamic function placement, elastic workloads, and heterogeneous implementations. Existing work has improved observability in cloud-native and open-source 5G systems in several complementary ways. Some approaches focus on infrastructure-level visibility, exposing resource and service metrics from containers, pods, and hosts [8]. Others rely on tracing between network functions or on telecom-specific measurements from selected parts of the RAN or the core [12]–[14]. Our previous work addressed part of this broader challenge by extending observability to the RAN and integrating fine-grained gNB telemetry into a cloud-native monitoring pipeline [4], [5]. Yet the user plane remains difficult to observe in a way that is directly useful for latency analysis. Current observability signals do not clearly tell the operator where latency first increases along the path, how that increase should be attributed to a given UE or slice, or which subsystem should be investigated first. Full packet tracing can provide the required detail, but it is expensive as a permanent monitoring method. Traces grow quickly, often need to be captured at multiple points, and require offline correlation. Raw user-plane packets also expose tunnel identifiers rather than the UE and slice information needed for interpretation, so recovering this context requires additional control-plane processing. Moreover, implementation-specific tracing hooks make experiments difficult to reproduce across different open-source gNB and core stacks. Our goal is to provide a lighter and more general alternative for open-source cloud-native 5G deployments. Rather than modifying the internals of a specific 5G implementation, the framework proposed in this paper observes standard user-plane traffic and combines it with externally reconstructed UE context. This keeps the measurement principle portable across implementations that expose the relevant interfaces, while allowing instrumentation to be activated only for the selected UE, slice, logical direction, and network function. The framework relies on two building blocks:

979-8-3195-0167-7/26/$31.00 ©2026 European Union

a *UE context mapping mechanism* that reconstructs slice-relevant user information from packet-captured control-plane exchanges, and a Kubernetes-native controller that deploys *ephemeral eBPF instrumentation* only when a measurement is requested. In this way, it decomposes latency across the path without requiring permanent instrumentation of every network function. Using controlled scenarios on a realistic open-source 5G testbed, we show that it reveals distinct *latency signatures* for radio impairments, UPF overload, and traffic-server delay while maintaining low overhead. A companion demonstration paper presents the framework in live operation, including UE attachment, on-demand probe deployment, and synchronized visualization of latency and radio measurements [22].

The main contributions of this paper are as follows.

- We propose a *slice-aware UE context mapper* that reconstructs per-user and per-slice state from packet-captured control-plane signaling.
- We design a *controller-driven, on-demand instrumentation framework* that injects ephemeral eBPF probes at selected anchors without modifying the 5G stack.
- We introduce a *multi-anchor latency decomposition method* that separates RAN-, core-, and server-side contributions.
- We validate the framework on a realistic cloud-native 5G testbed and demonstrate its low runtime overhead. All results are reproducible using the publicly available deployment, experiment, and analysis scripts [6].

The remainder of this paper is organized as follows. Section II reviews the relevant literature. Section III presents the system design, including the eBPF latency filter, measurement definitions, and latency decomposition methodology. Section IV describes the implementation. Section V reports the validation and experimental results. Section VI discusses the monitoring overhead, and Section VII concludes the paper.

II. Background and Related Work

5G introduces explicit service differentiation mechanisms through PDU sessions, slices, QoS flows, and standardized QoS characteristics such as packet delay budgets [1]. As a result, latency becomes a first-class performance concern: an operator or experimenter should not only know that a UE is connected, but also whether the packets associated with that UE or slice experience delay degradation and where that delay appears along the path. This matters not only for diagnosis and root-cause analysis, but also for checking whether low-latency service objectives and SLA requirements are being met. In cloud-native deployments, however, this visibility is difficult to obtain because the entities exposed by Kubernetes observability do not naturally align with those of the 5G system. General-purpose monitoring stacks such as Prometheus and Grafana [8], [9] are useful in cloud-native 5G experiments, but they remain infrastructure-centric: they can show pod-level counters and resource usage, yet they do not directly explain which UE or slice is responsible for a latency increase. Distributed tracing and service-mesh telemetry address a related but different observability problem. OpenTelemetry and Istio follow requests across microservices using spans, propagation headers, and sidecar-generated telemetry [10], [11]. Several 5G studies follow this direction as well. 5GC-Tracer improves 5G core observability with non-intrusive tracing [12], 5G-MAP studies cloud-based 5G core behavior using detailed measurements and OpenTelemetry-based tracing [13], 5GC-Observer uses eBPF to expose cloud-native 5G core telemetry and detect QoS degradation from traffic observed between core components [14], and more recent thesis work extends this tracing-based perspective to end-to-end 5GS analysis across the core and the RAN [15]. These approaches improve observability inside the 5G core, but they do not directly address latency decomposition along the user-plane path. In our case, the goal is not only to observe cloud-native 5G functions, but also to attribute latency to specific UEs and slices and to localize where it is introduced along the path. On this path, traffic is carried as GTP-U packets rather than as application requests with trace context. As a result, packets expose tunnel identifiers rather than full UE and slice semantics, and the forwarding path may traverse privileged functions or kernel hooks where a service-mesh proxy would either miss the relevant traffic or alter the path being measured. User-plane latency decomposition therefore requires packet-level observation tied to externally reconstructed UE and slice context. 5G-specific observability closes part of this gap by exposing telecom-aware metrics from the RAN or the core. Our previous work extends Monarch [3] with real-time RAN observability for open-source 5G deployments based on OAI [16] and srsRAN [19], and complements it with an Ansible-based framework for reproducible experiments [4], [5]. This makes the radio side visible and supports controlled experimentation, but it still does not show where latency is introduced along the path. Because existing approaches expose different monitoring outputs rather than the same latency decomposition, a like-for-like experimental comparison is unavailable; we instead validate against independent pcap-based and application-visible references in Section V.

Applying packet-level instrumentation directly in 5G is non-trivial. User traffic on the N3 interface is encapsulated in the GPRS Tunnelling Protocol User Plane (GTP-U) [2], so a latency monitor must recover both the outer tunnel context and the inner transport packet. At the same time, the semantic labels required for analysis –such as the IMSI, UE IP address, slice, and session context–come from control-plane state rather than from each user-plane packet itself. Permanent packet tracing is also undesirable in large cloud-native 5G deployments: it increases operational complexity, produces large traces, captures traffic unrelated to the question being investigated, and is often difficult to reuse across different open-source 5G stacks without implementation-specific hooks.

This paper brings these missing capabilities together. It uses packet capture to reconstruct control-plane context and connects this live UE context map with controller-driven eBPF instrumentation [7]. The UE mapper tracks active UEs and their slice and TEID state (GTP-U Tunnel Endpoint Identifiers) by decoding control-plane packets; the controller injects eBPF

probes only for the selected UE, slice, direction, and monitored network function; and the exporter exposes both fine-grained per-UE metrics and aggregate per-slice distributions. The result is an *on-demand latency-decomposition capability* that can be added to open-source cloud-native 5G deployments, while keeping the instrumentation external to the 5G stack.

III. System Design

The proposed framework decomposes latency by combining a continuously maintained control-plane view with on-demand data-plane instrumentation. The control-plane component is built from packet capture: it passively observes and decodes control-plane exchanges to maintain the relationship between UE identity, slice information, UE IP address, and GTP-U Tunnel Endpoint Identifiers (TEIDs). A Kubernetes-native controller then uses this mapping to decide where and how to inject packet-level eBPF probes and to label the resulting latency samples when a measurement is requested. These probes are passive with respect to user traffic. Once attached, they observe packets already in flight and export timing information without generating probe traffic, altering the measured packets, or modifying network-function implementations.

A. Control-Plane UE Context Mapping

The UE context mapper reconstructs a slice-aware view of active UEs by capturing and decoding standard 5G control-plane packets. It runs alongside the deployment and updates its state as UEs attach, establish or release PDU sessions, and disconnect. For each active UE, it maintains the identifiers needed by the monitoring stack: the IMSI[1] or stable UE identifier, the UE IP address, the RAN UE identifier when available, the selected slice, and the uplink and downlink TEIDs associated with the PDU session. The result is a live TEID-to-UE map consumed by the controller, latency probes, and exporters. This mapping lets measurements collected at different points be correlated without modifying the 5G stack. For example, a latency sample emitted by an eBPF program contains a TEID together with the probe metadata needed for interpretation. Before exposing it to Prometheus, the exporter enriches the sample with the corresponding UE and slice labels. The dashboard can then group latency by UE, slice, direction, and probe role, that is, whether the sample was observed at the gNB-side or UPF-side probe, rather than showing anonymous packet events. When a UE disconnects or its session state changes, the stale mapping is removed or replaced so that new samples are not attributed to an obsolete UE context. Mapping accuracy depends on observing control-plane updates; missed signaling, rapid mobility, or route changes outside the monitored interfaces may temporarily yield stale or incomplete context. Evaluating these cases remains future work.

[1]IMSI: International Mobile Subscriber Identity.

B. On-Demand Deployment

The instrumentation is deployed as ephemeral monitoring pods, following the automation principles used in our previous *5g_ansible* framework for reproducible experiments [4], [5]. A custom Kubernetes controller drives this on-demand behavior. It watches the measurement intent and the current UE mapping, identifies the target network functions, selects the appropriate network interface, injects the TEID mapping and mode configuration, and launches the eBPF latency probe. The measurement intent can be scoped to one UE, one slice, or a set of UEs within the same slice. In each case, the controller translates the requested scope into the corresponding uplink and downlink TEIDs, so the injected probe follows the tunnels that actually carry the selected UE or slice traffic. In this user-plane scenario, the same mechanism can target a gNB, a UPF data-plane interface, or both, depending on the experiment. This keeps the monitoring logic external to the 5G implementation. As a result, the gNB, UPF, and core control-plane functions do not need to be patched to support a new experiment. Instead, the controller injects the probe only on demand – for instance, when a specific UE, slice, direction, or network-function anchor requires observation. It attaches the eBPF program dynamically through Linux traffic control (TC) and can remove the probe when the experiment or troubleshooting session ends. This is particularly useful in large and dynamic deployments, where instrumentation should be precise, reproducible, and limited to the traffic and network functions relevant to the current diagnosis.

C. On-Demand Control Loop

Algorithm 1 summarizes the control loop. The mapper first exposes the current UE state, after which an experiment or operator specifies the measurement scope. In the notation below, $\mathcal{U}$ is the set of selected UEs, $\mathcal{S}$ is the set of selected slices, D is the logical direction, $\mathcal{A}$ is the set of requested observation anchors, and M is the measurement mode. The controller resolves this request against the live state, creates or updates the monitoring pods, and keeps their labels synchronized while the request remains active. The labels produced by this loop support two levels of analysis. At the fine-grained level, a user can inspect an individual UE, TEID, direction, mode, and probe placement. At the aggregate level, the same samples can be grouped by slice, network function, or direction. In this paper, *slice-aware* means that slice identifiers recovered from control-plane state are used to label, filter, and aggregate latency samples per slice. This is already useful even though current open-source cloud-native 5G deployments do not yet provide complete slice-specific data-plane QoS enforcement, and it remains directly applicable when stronger open-source slice-isolation and QoS mechanisms become available.

D. eBPF Latency Filter

Extended Berkeley Packet Filter (eBPF) is a Linux kernel mechanism that allows small verified programs to be attached dynamically to selected execution points, including the packet-processing path, without recompiling the kernel [7]. In our

Require: Measurement request $r = (\mathcal{U}, \mathcal{S}, D, \mathcal{A}, M)$
Read live UE/slice/TEID context from the mapper
Resolve $\mathcal{U}$ or $\mathcal{S}$ to active UE IPs and UL/DL TEIDs
for each requested anchor $a \in \mathcal{A}$ **do**
    Locate the target pod, data plane interface, and probe role
    Create or update the ephemeral monitoring pod
    Load selected TEIDs, mode M, and role into eBPF maps
    Attach the TC program to the selected ingress/egress hooks
end for
Export UE-, slice-, mode-, and anchor-labeled metrics
while request r is active **do**
    Refresh labels if UE context changes
    Track sample counts, lost events, map pressure, and match health
end while
Detach probes and remove stale measurement state when r ends

Fig. 1. On-demand slice-aware latency measurement control loop.

case, it is well suited to the problem because the monitored traffic crosses standard Linux network interfaces and must be observed at packet granularity, close to where packets are actually forwarded. We attach the program through Linux traffic control (TC), which exposes ingress and egress hooks on the selected interface and allows packets to be inspected before they leave or after they arrive at the monitored anchor. Depending on the experiment, the filter can be deployed on one or more host-network interfaces, in our case the gNB-side and UPF-side data-plane interfaces.

The latency filter itself is an eBPF program attached to both ingress and egress TC hooks of the monitored interface. It first retains only IPv4/UDP packets using the GTP-U port 2152. It then parses the GTP-U header, extracts the TEID, handles optional GTP extension headers, and locates the encapsulated inner IPv4 packet. From that inner packet, it extracts the transport protocol, addresses, ports, TCP sequence and acknowledgement numbers, ICMP identifiers if applicable, and packet direction. In its current form, the filter supports TCP and ICMP latency measurements. TCP is the primary focus because it carries most application traffic in practice, while ICMP provides an additional validation path through controlled ping measurements.

The filter produces three classes of events. TEID events report the GTP-U tunnels observed at ingress and egress. Flow events describe the inner packet flow and protocol, which helps verify that the correct traffic is being measured. Latency events report round-trip samples in nanoseconds for the downstream part of the path, together with the TEID, measurement mode, and packet type. These events are streamed to user space through perf buffers and converted into JSON records consumed by a local Prometheus exporter.

For TCP traffic, the filter uses sequence-aware matching between a data-bearing segment and the corresponding reverse ACK. When a data-bearing TCP segment is observed, the program stores a timestamp keyed by the five-tuple and the expected acknowledgement number. When the corresponding ACK is later observed in the reverse direction, it emits the elapsed time as a latency sample and removes the pending entry. This design produces a local RTT sample without requiring clock synchronization between different nodes. One-way delay would be attractive in principle, but it would require synchronized time sources that are not generally available by default in open-source cloud-native 5G deployments.

TABLE I
MAPPING BETWEEN PROBE PLACEMENT, TC HOOK ON THE MONITORED N3 INTERFACE, AND LOGICAL DIRECTION.

Probe role	Logical uplink	Logical downlink
gNB	N3 egress	N3 ingress
UPF	N3 ingress	N3 egress

For ICMP echo request/reply traffic, the same principle is applied to request and reply packets, using the ICMP identifier and endpoint addresses as the correlation key.

E. Measurement Definitions and Modes

We now define the labels used by the measurement pipeline. The exporter implements the labels `mode="CORE"` and `mode="RAN"`. These labels are tied to the logical traffic direction used for matching, but more importantly they indicate whether the resulting RTT is evaluated toward the core side or toward the RAN side of the path. Let $A \in \{\mathrm{gNB}, \mathrm{UPF}\}$ denote the monitored network function where a probe is attached, and let $d \in \{\mathrm{UL}, \mathrm{DL}\}$ denote the logical direction of a user-plane packet. We use *probe role* to denote the side from which the packet is observed, namely a gNB-side probe or a UPF-side probe. Logical uplink means UE-to-traffic-server traffic, while logical downlink means traffic-server-to-UE traffic. Because the same N3 interface is observed from different endpoints, the mapping between TC hook and logical direction depends on the probe role, as shown in Table I. For a packet-response pair p, the direct probe emits an anchor-local round-trip sample:

$$L_A^m(p) = t_A^{\mathrm{rev}}(p) - t_A^{\mathrm{fwd}}(p), \tag{1}$$

where $t_A^{\mathrm{fwd}}(p)$ is the timestamp of the forward packet observed at anchor A, $t_A^{\mathrm{rev}}(p)$ is the timestamp of the matching reverse packet at the same anchor, and m is the measurement mode.

In *CORE* mode, the forward packet is logical uplink and the reverse packet is the corresponding logical downlink ACK or response. Accordingly, L_A^{CORE} is the RTT visible from anchor A toward the traffic-server side. At the gNB anchor, it includes the path from the gNB N3 interface through the UPF and traffic-server and back to the gNB N3 interface. At the UPF anchor, it reflects the UPF/traffic-server side visible from the UPF N3 interface.

In *RAN* mode, the forward packet is the logical downlink and the reverse packet is the corresponding logical uplink ACK or response. Accordingly, L_A^{RAN} is the RTT visible from anchor A toward the UE/RAN side. At the gNB anchor, this primarily reflects the gNB/UE/RAN-facing response path; at the UPF anchor, it additionally includes the N3 path between UPF and gNB. In both modes, both matching logics are enabled and the emitted samples are distinguished by the mode label. For TCP, a data-bearing segment with sequence number s and payload length l creates an expected acknowledgement point $a = s + l$, with the standard SYN/FIN sequence advancement when relevant. A reverse ACK that acknowledges this point

closes the sample. For ICMP, an Echo Request and Echo Reply with the same identifier and endpoints form the pair. This symmetric structure avoids hard-coding anchor-specific logic and allows the same eBPF object and user-space loader to operate at different observation points. The exporter retains several statistical views because they answer different questions. It records emitted samples as counters, histograms, latest-value gauges, and one-second medians. In the figures presented in this paper, the one-second median latency metric denotes the median of all latency samples received by the exporter within a one-second interval for a given label set. This metric differs from both a last-seen gauge and a Prometheus histogram quantile, which is estimated over a longer query window. The resulting samples are labeled with the probe role, target pod, node, TEID, UE identity, and slice, making it possible to compare measurements from multiple points without changing the measurement code. The current passive RTT estimators support TCP and ICMP, but not QUIC or arbitrary UDP traffic, for which response-correlation information may be encrypted or unavailable. The general monitoring and instrumentation mechanisms still apply, but latency extraction would require a suitable protocol-specific signal or cooperation from an instrumented client or server.

F. Latency Decomposition Methodology

A single probe provides the latency visible from one anchor, but decomposition requires comparing multiple anchors. When probes are deployed at both the gNB and the UPF, the framework can optionally enable a same-packet pairer. This service receives TCP observation records from different probes and matches the same packet using a fingerprint composed of the mode, logical direction, protocol, TEID, inner five-tuple, TCP sequence information, payload length, and flags. For a matched packet p, the pairer exposes the local RTT observed at the UPF, $L^{m}_{\mathrm{UPF}}(p)$, the local RTT observed at the gNB, $L^{m}_{\mathrm{gNB}}(p)$, and their direction-aware gap:

$$G_{\mathrm{DL}}(p) = L^{\mathrm{RAN}}_{\mathrm{UPF}}(p) - L^{\mathrm{RAN}}_{\mathrm{gNB}}(p), \tag{2}$$

$$G_{\mathrm{UL}}(p) = L^{\mathrm{CORE}}_{\mathrm{gNB}}(p) - L^{\mathrm{CORE}}_{\mathrm{UPF}}(p). \tag{3}$$

Taken together, these quantities provide a practical path-decomposition view. The four anchor-local RTT quantities are $L^{\mathrm{CORE}}_{\mathrm{gNB}}(p)$ and $L^{\mathrm{CORE}}_{\mathrm{UPF}}(p)$ for the core-facing view at the two anchors, and $L^{\mathrm{RAN}}_{\mathrm{gNB}}(p)$ and $L^{\mathrm{RAN}}_{\mathrm{UPF}}(p)$ for the RAN-facing view at the same anchors. Among them, the direct UPF-anchor core RTT, $L^{\mathrm{CORE}}_{\mathrm{UPF}}(p)$, reflects the UPF-to-traffic-server side, while the direct gNB-anchor RAN RTT, $L^{\mathrm{RAN}}_{\mathrm{gNB}}(p)$, reflects the gNB-to-UE/RAN side. The uplink gap $G_{\mathrm{UL}}(p)$ then captures the additional RTT contribution of the gNB–UPF segment on the core-facing path. Fig. 2 illustrates this interpretation. These gap quantities must be interpreted together with the anchor-local RTT components introduced above. They are packet-level differences between anchor-local RTTs for the same packet, not direct one-way delays. In practice, the uplink gap $G_{\mathrm{UL}}(p)$ provides a core-side comparison between the two anchors, while the downlink gap $G_{\mathrm{DL}}(p)$ provides the complementary

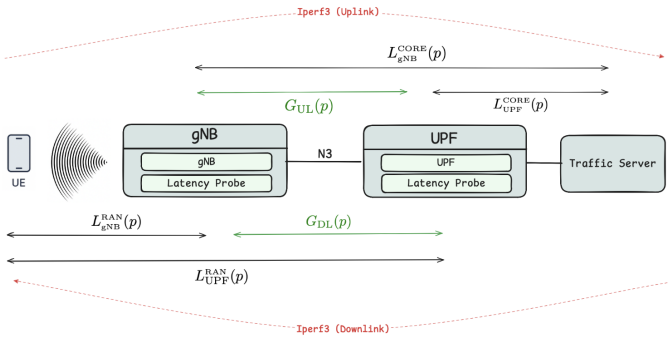


Fig. 2. Path-decomposition view.

cross-anchor signal on the downlink path. Together with the direct gNB-anchor quantity $L^{\mathrm{RAN}}_{\mathrm{gNB}}(p)$, which reflects the RTT seen from the gNB toward the UE/RAN side, these quantities support localization of latency changes across the RAN-facing and core-facing parts of the path. Negative gaps and unmatched samples are rejected and counted separately, which helps detect matching, timing, or loss anomalies during experiments. Thanks to this proposed latency decomposition, the pairer can therefore determine the principal source of latency variations for a given UE or slice by assessing whether the observed changes are predominantly associated with the RAN, the core network, or both components.

IV. Implementation

The framework consists of four components. The kernel component is the eBPF TC program that parses user plane traffic and emits latency events. The user-space loader configures the eBPF maps, attaches the program to ingress and egress hooks, loads the selected mode and probe role, and forwards structured events over a local socket. The Prometheus exporter then transforms these events into gauges, counters, histograms, and one-second median latency metrics. Histograms support distribution and quantile queries over configurable windows, latest-value gauges are useful for live dashboards, and the one-second median metric is used when the analysis requires a stable per-second latency signal. Finally, an optional same-packet pairer mode correlates observations from different probes to expose paired RTT and gap metrics. Each latency event exported by the framework is labeled with the TEID, slice, mode, packet kind, probe role, probe pod, probe node, probe target, and UE metadata. This label set is deliberately redundant. It supports both low-level debugging, where TEID and target pod matter, and experiment-level analysis, where IMSI, UE IP, slice, and mode are more useful. Histograms and one-second medians provide per-UE and per-slice latency distributions, while Prometheus aggregation provides slice-wide summaries over all UEs belonging to the selected slice. Counters track emitted samples, lost perf-buffer events, pending-map pressure, unmatched ACKs, same-packet match rates, and pair rejection reasons. These counters support operational diagnostics by verifying that matching is healthy, detecting

buffer pressure or dropped events, and explaining sparse or inconsistent samples during debugging.

The same-packet pairer is an optional cross-anchor mode. When enabled, it matches the same TCP packet across gNB and UPF observations and exports the paired anchor-local RTTs together with the gap quantities defined in Eqs. (2) and (3). These packet-level gaps are therefore defined only for same-packet matches, enabling deeper validation and cross-anchor inspection when the pairer is enabled. However, when the pairer is not enabled, exact packet-level gaps are not computed; instead, the framework compares anchor-local direct metrics statistically over common windows, using one-second medians, distributions, and phase-level summaries.

V. Validation and Experimental Evaluation

In this section, we first validate the framework by assessing measurement accuracy and overhead, and then demonstrate its ability to decompose end-to-end latency and identify the part of the 5G path responsible for an observed delay increase.

A. Experimental Setup

We run our experiments on the SophiaNode testbed, a site of the SLICES research infrastructure (SLICES-RI) [21] that includes the R2lab open wireless experimentation platform [20]. The experimental platform comprises a Kubernetes-based open-source 5G deployment with OAI gNB functions, an Open5GS core [17], a real 5G Radio Unit (RU), and three commercial off-the-shelf (COTS) UEs. A host orchestrates the deployment through Ansible but lies outside the measured packet path. In the setup reported in Fig. 3, the Open5GS core runs on sopnode-f1, the monitoring pipeline on sopnode-f2, the gNB on sopnode-f3, and a dedicated traffic server on sopnode-w3. UE1, UE2, and UE3 denote the three 5G UEs, which are based on Quectel RM500Q-GL devices. UE1 and UE3 are located closer to the AW2S Jaguar RU, whereas UE2 is positioned physically farther away and is therefore referred to as the far UE in the radio-degradation experiment. Further details about the infrastructure and its nodes are available in the SLICES-RI and R2lab documentation [20], [21]. Unless stated otherwise, all four decomposition scenarios summarized in Table II use the same traffic template: the three attached UEs exchange TCP traffic with a dedicated Iperf3 endpoint on the traffic server sopnode-w3 beyond the UPF. Each scenario runs in both uplink and downlink, with 180 s windows per direction and Prometheus scraping at 1 s resolution. Scenario 1 is the baseline reference. Scenarios 2, 3, and 4 then introduce one perturbation at a time: Scenario 2 uses UE2 as the physically distant UE and increases the offered load with two parallel Iperf3 TCP streams per UE, Scenario 3 activates eight CPU stress workers on the UPF pods, and Scenario 4 injects 100 ms of artificial delay at the traffic server.

B. Registration-level Validation of UE Context Tracking

We start with the validation of the control-plane context tracking path under synchronized UE registration load. This scenario was used only for registration-level validation and, unlike the TCP latency experiments, did not require the machine running Iperf3, since no user-plane traffic was generated. Instead, we used UERANSIM [18], a software 5G RAN simulator, with software-simulated UE and gNB components deployed on the same RAN node in the testbed. The UERANSIM UEs were activated in synchronized waves of 10, 50, 100, and 200 users while the monitoring pipeline reconstructed UE state from packet-captured control-plane exchanges. Across 10 validation runs, totaling 3,600 attempted UE registrations, AMF registration succeeded for 3,592 UEs (99.78%). End-to-end monitoring succeeded for 3,589 UEs (99.69%), and, conditioned on successful registration, the monitoring pipeline succeeded for 3,589 out of 3,592 UEs (99.92%). These results indicate that, for the current test scope, the external UE-to-context mapping mechanism behaves correctly at the registration level and does not show systematic loss on the AMF side under bursty attach conditions.

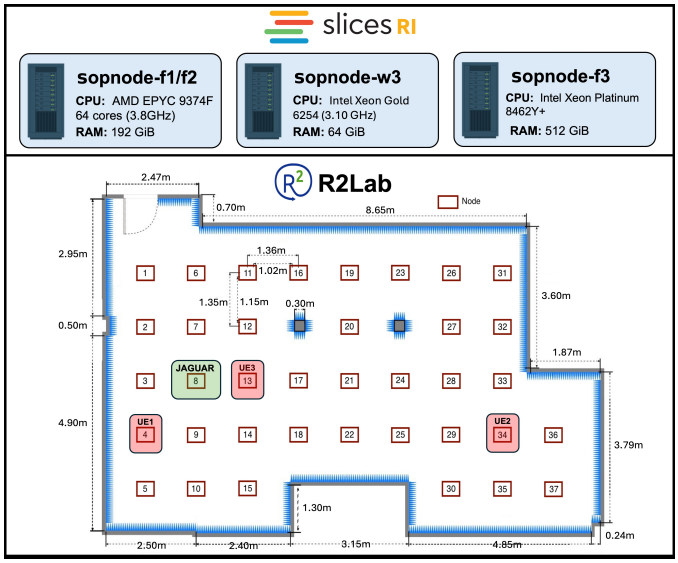


Fig. 3. Experimental setup.

C. TCP-based Latency Validation

The TCP-based latency validation serves two purposes: it validates the direct eBPF ACK-based RTT signal and establishes the reference operating point used in the later scenarios. For this validation, TCP downlink traffic is generated with Iperf3 from the traffic server toward the UE at 30 Mb/s. The pcap reconstruction decapsulates GTP-U, extracts the inner TCP stream, and pairs each data segment with the reverse ACK acknowledging its payload. The pcap RTT is the timestamp difference between the data segment and the matching ACK and is used here for validation purposes. Note that the pcap reference and the eBPF probe are not expected to expose the exact same set of packet samples. For this reason, the validation compares distributions and one-second medians rather than relying on a raw last-seen latency gauge.

After aggregation, we found that the median of downlink TCP probe latency was 15.557 ms and the median of pcap-

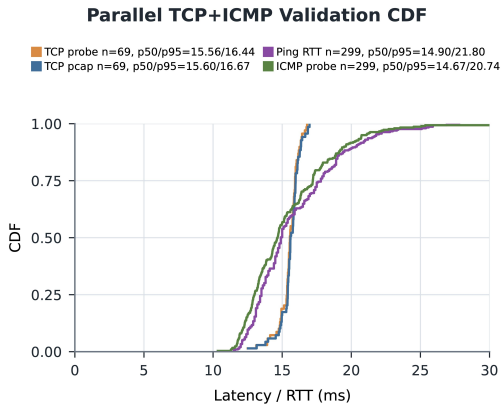


Fig. 4. Parallel downlink TCP and ICMP validation.

reconstructed latency was 15.601 ms. The median difference is then 0.044 ms, which is small relative to the measured RTT and supports using the direct ACK-based eBPF signal of our framework as the main latency metric.

D. ICMP-based Latency Validation

ICMP validation provides another sanity check of our framework, this time by comparing our measurements to application-visible measurements carried out with ICMP echo request/reply traffic generated with ping. During the validation run, TCP Iperf3 traffic and ICMP probing are executed in parallel, and the exported probe metrics are compared with independent references: pcap-reconstructed TCP RTT for TCP and the RTT printed by ping for ICMP. The ping payload is set to 1472 bytes, producing packets close to the Ethernet MTU after IP and ICMP headers. This avoids comparing TCP data packets with very small ICMP probes. Results are reported in Fig. 4. The figure shows that the delays measured by our probes closely match those computed from the pcap traces for both TCP and ICMP traffic. As expected, the distribution of ICMP delays differs from that of TCP delays, since the operating system kernel processes these protocols differently, applying distinct priorities and scheduling mechanisms. In this run, the ICMP probe median remains close to the application-visible ping RTT, with a difference of 0.234 ms. The TCP probe median differs from the pcap reconstruction median by only 0.044 ms, as seen beforehand. These values are consistent across the probe measurements, the packet-capture reference, and the application-visible ping RTT at the scale relevant to this paper's experiments.

E. Latency Decomposition Experiment Scenarios

Having validated the measurement signal, we turn to the main experimental question: can the framework identify where a latency increase originates along the end-to-end path? We use the four scenarios summarized in Table II, all driven by the Iperf3 TCP traffic template described above and exported at one-second Prometheus resolution. The main analysis relies on the direct one-second median latency metric, filtered to bins with non-zero event rate and interpreted together with radio and system-level metrics. The baseline run establishes the per-UE reference for UE1, UE2, and UE3 under matched TCP traffic. Because UE2 is physically farther from the radio unit, this baseline already reveals the latency differences associated with radio placement. The far-UE scenario then exercises that weaker radio condition under higher load, using two parallel Iperf3 TCP streams per UE, and compares UE2 with UE1 and UE3 as near references. The UPF-stress scenario perturbs the user-plane processing path with CPU worker processes running on the UPF pods, whereas the traffic-server scenario adds artificial delay at the traffic server. Taken together, these four cases show not only that latency changes are detected, but also that the framework helps indicate which part of the path is most consistent with the observed change. Each reported scenario signature was reproduced in at least ten campaign runs, and the diagnostic pattern remained stable across repetitions. For the following figures, direct latency samples are filtered to one-second bins.

TABLE II
LATENCY DECOMPOSITION SCENARIOS.

Scenario	Perturbation	Expected decomposition signature
1	Baseline all UEs	Reference per-UE latency under matched UL/DL TCP traffic.
2	Far UE / RAN degradation	Downlink RAN latency and retransmissions increase for the far UE while core-side medians stay low.
3	UPF CPU stress	Core-side RTT rises during stress and returns during recovery, while UPF container CPU increases sharply and gNB CPU remains stable.
4	Traffic-server delay	Both gNB and UPF core RTT shift by the injected 100 ms delay while gNB and UPF CPU stay flat.

Radio degradation. Fig. 5 illustrates Scenario 2. It contrasts the far UE with the near-reference UEs and the baseline behavior, which is what reveals the radio-side latency signature. In the baseline downlink window, UE1 and UE3 have gNB-side RAN medians of 24.1 ms and 25.0 ms, with p95 values of 28.7 ms and 33.5 ms. In the far-UE case shown on the right-hand side of the figure, UE2 reaches a gNB-side RAN median of 534.5 ms and p95 of 627.4 ms. The gNB-side core medians remain below about 0.15 ms in the same comparison, so the latency inflation is consistent with a radio/RAN-side degradation rather than a UPF-side or endpoint-side bottleneck. The supporting radio and transport signals point in the same direction: the far-UE downlink window has an average RSRP of about -89 dBm, whereas the near-reference UEs are around -68.5 dBm. UE2 also records 7,405 TCP retransmissions, compared with 1,866 and 1,704 for UE1 and UE3, which is consistent with a degraded radio link rather than a core-side collapse.

UPF CPU stress. Fig. 6 reports the UPF-stress perturbation for one representative UE. We focus on one UE to avoid masking per-UE behavior through pooled summaries. During the stress phase, the direct uplink core RTT median rises from 0.20 ms to 200.3 ms at the gNB anchor and from 0.09 ms to 170.1 ms at the UPF anchor, before returning to 0.23 ms and 0.11 ms in recovery. On their own, these latency measurements

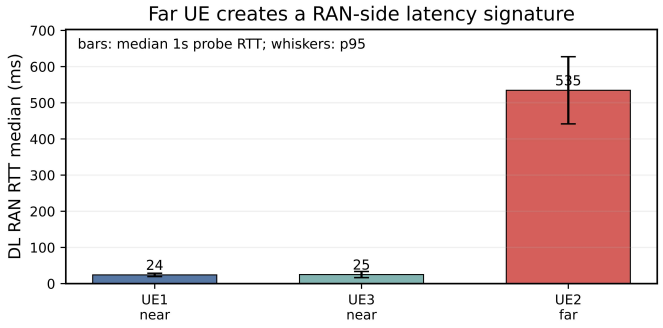


Fig. 5. RAN-side downlink latency in the near-reference and far-UE scenarios.

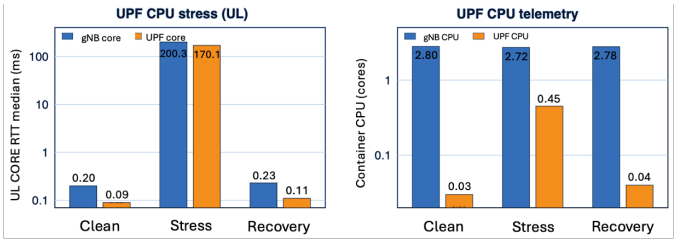


Fig. 6. UPF-stress localization.

localize the degradation to the core-side path rather than to the RAN. The co-collected CPU metrics then identify the stressed function more precisely: the gNB container remains approximately stable at 2.80, 2.72, and 2.78 CPU cores across clean, stress, and recovery, whereas the two stressed UPF containers increase from about 0.02 and 0.04 cores in the clean phase to about 0.43 and 0.48 cores during stress, before returning to about 0.02 and 0.05 cores in recovery. Taken together, the latency and CPU signals narrow the fault domain and point to the stressed UPF processing path.

Traffic server delay. Fig. 7 shows the traffic-server delay scenario. A controlled *tc netem* rule adds 100 ms of ingress delay at the dedicated traffic server running Iperf3 beyond the UPF. The median uplink core RTT increases from 0.20 ms to 100.2 ms at the gNB anchor and from 0.09 ms to 100.1 ms at the UPF anchor, then returns to 0.21 ms and 0.08 ms after cleanup. The accompanying CPU panel does not show a matching processing bottleneck: gNB CPU stays near 2.02 cores, and UPF CPU remains below 0.001 core throughout clean, delay, and recovery. This is also not the RAN signature seen in Scenario 2. Here, the delay appears almost equally at both core-side anchors while the monitored network-function CPU stays flat, so the remaining consistent location is the common endpoint-side path after the UPF. In this controlled experiment, the *tc* counters report approximately 471k redirected packets and no drops, confirming that the injected traffic-server-side rule was active.

VI. Overhead Discussion

We evaluate overhead at two levels. First, container CPU and memory metrics quantify the cost of the monitoring components as seen by Kubernetes. Second, low-level eBPF

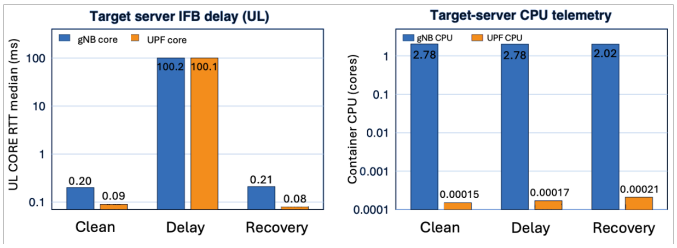


Fig. 7. Traffic-server delay localization.

TABLE III
Mean monitoring overhead in the reported setup.

Component	CPU (cores)	Memory (MiB)
gNB probe	0.116	26.4
UPF-side probes (mean across the 2 active UPF pods)	0.044	25.1
Kopf controller	0.070	112.3
Prometheus	0.076	760.7
Same-packet pairer (optional active pod)	0.046	71.2

runtime counters quantify the cost of the packet-path program itself. These two views answer different questions: container CPU reflects how much host compute the monitoring pipeline consumes, while eBPF runtime reflects how much processing time is spent per packet inside the attached TC program.

Table III summarizes the container-level overhead observed in the reported setup. For the eBPF rows, we report the active probe containers that carried the experiment traffic. At the UPF side, the reported setup uses two active UPF pods, corresponding to the two slice paths exercised in this campaign, and each has its own attached probe; the table therefore reports the mean across these two UPF-side probe containers. In these runs, the gNB latency probe consumes 0.116 CPU cores on average, while the mean UPF probe cost is 0.044 CPU cores. The controller and Prometheus remain outside the packet-forwarding path, but they are included because they represent the operational cost of the monitoring pipeline. We next examine the packet-path runtime cost of the attached TC/eBPF program itself. The latency probe is compared with a passive baseline TC program that performs minimal packet handling and forwards the packet. This baseline is not meant to reproduce the full monitoring logic; it provides a lower bound for the cost of attaching a simple TC/eBPF action to the same path. In the validation run, the passive TC baseline costs 11.5 ns per packet. The latency probe costs 2.29 μs on ingress and 1.25 μs on egress, corresponding to 199$\times$ and 109$\times$ the passive baseline, respectively. The multiplicative factor is large because the baseline does almost no work, but the absolute cost remains in the microsecond range. The eBPF runtime counter measures CPU service time spent executing the program, not a value that can be directly added to every packet's end-to-end RTT. It becomes visible as queueing only if packets arrive faster than the CPU can process them, or if the host is otherwise saturated. The TCP probe also keeps a bounded per-flow correlation window of size $B = 2048$, where B denotes the maximum number of pending packets retained for ACK matching, so saturation would eventually

appear as systematic overwrites if the packet rate and in-flight latency jointly exceeded this budget. Under the validation workload, the observed capture rate is 7.2k packets/s at the gNB, whereas the corresponding p95-latency operating budget is approximately 124.6k packets/s before systematic overwrites would be expected. In other words, the measured workload is roughly $17\times$ below the point where the current matching window would become the first concern. This packet-rate bound corresponds roughly to 1.5 Gb/s for MTU-sized packets, while smaller packets would reach the same bound at lower throughput. A rough CPU-side extrapolation from the measured eBPF runtime gives a much higher per-core service ceiling, so in this setup the correlation window is the tighter implementation bound. The current value $B = 2048$ is a conservative design choice and could be increased further if a larger memory budget were acceptable. Together with the container CPU measurements, these results indicate that the monitoring overhead remains small in the reported experiments. At the same time, the design preserves the practical advantage of on-demand deployment and removal for a narrowly defined observation scope. The control-plane context tracking used in the registration-level validation under synchronized UE registration load also remained lightweight. Across the same 10 runs, the AMF-side sniffer consumed on average 0.032 CPU cores and 88.1 MiB of memory, with average p95 values of 0.207 CPU cores and 91.2 MiB, respectively. The two SMF-side sniffers consumed about 0.002 CPU cores and 78.9 MiB of memory on average. Interface drop counters did not increase and the packet socket receive backlog remained at zero during these runs, indicating that the monitoring pipeline did not overload the capture path. The current implementation targets research testbeds. Control-plane context tracking is validated with up to 200 synchronized UEs, whereas the user-plane latency experiments involve three active commercial UEs; production-scale scalability is therefore not established. At larger scales, active TEIDs and flow entries, packet rate, and metric-export load may limit performance, although on-demand scoping helps contain these costs. Production-scale evaluation remains future work.

VII. Conclusion

This paper presents an on-demand, slice-aware latency decomposition framework for cloud-native 5G systems. To the best of our knowledge, this is the first open-source cloud-native 5G framework to combine external UE-context reconstruction with controller-driven on-demand eBPF instrumentation at both gNB and UPF anchors for slice-aware end-to-end latency decomposition and localization. By combining packet-capture-based UE context mapping with eBPF GTP-U instrumentation, the framework associates packet-level latency samples with users, slices, tunnels, and observation points, and distinguishes RAN-, core-, and server-side latency signatures. Because the instrumentation remains external to the 5G stack, the approach is practical for experimental diagnosis and controlled troubleshooting while remaining agnostic to the 5G implementation used. This framework opens the way to closed-loop monitoring, where instrumentation is activated on demand to detect problems, find their cause, and correct them automatically. All results reported in this paper are fully reproducible using the deployment, experiment, and analysis scripts available in a public repository [6].

Acknowledgements

This work was supported by the French National Research Agency (ANR) under the France 2030 program through the NF-NAI project (grant ANR-22-PEFT-0003).

References

[1] 3GPP, "System architecture for the 5G System (5GS)," 3GPP TS 23.501, Release 18, 2024.
[2] 3GPP, "General Packet Radio System (GPRS) Tunnelling Protocol User Plane (GTPv1-U)," 3GPP TS 29.281, Release 18, 2024.
[3] N. Saha, N. Shahriar, et al., "Monarch: Monitoring architecture for 5G and beyond network slices," *IEEE TNSM*, 2024.
[4] Y. Amami, Z. Mabrouk, C. Barakat, and T. Turletti, "Toward real-time RAN observability in open-source 5G systems," *ICIN 2026*.
[5] Y. Amami, Z. Mabrouk, C. Barakat, and T. Turletti, "Demonstrating Real-Time RAN Observability in Open-Source 5G: Controlled Interference Scenario," *ICIN 2026*.
[6] Y. Amami, "5g_ansible latency paper artifact repository," GitHub repository, 2026. [Online]. Available: https://github.com/yassir63/5g_ansible/tree/develop-latency-paper. Accessed: 2026-05-30.
[7] eBPF Docs, "eBPF documentation," [Online]. Available: https://docs.ebpf.io/. Accessed: 2026-05-30.
[8] Prometheus Authors, "Prometheus: Monitoring system and time series database," [Online]. Available: https://prometheus.io/. Accessed: 2026-05-30.
[9] Grafana Labs, "Grafana OSS and Enterprise," [Online]. Available: https://grafana.com/docs/grafana/latest/. Accessed: 2026-06-11.
[10] OpenTelemetry Authors, "OpenTelemetry traces," [Online]. Available: https://opentelemetry.io/docs/concepts/signals/traces/. Accessed: 2026-05-30.
[11] Istio Authors, "Istio observability," [Online]. Available: https://istio.io/latest/docs/concepts/observability/. Accessed: 2026-05-30.
[12] A. Zhao, F. Desnoes, I. Fajjari, and N. Aitsaadi, "5GC-Tracer: A novel non-intrusive distributed tracing for enhanced 5G core network observability," in *NOMS*, 2025.
[13] T. O. Atalay, D. Stojadinovic, A. Famili, A. Stavrou, and H. Wang, "5G-MAP: Demystifying the performance implications of cloud-based 5G core deployments," in *ACM MobiCom*, 2025.
[14] A. Khichane, I. Fajjari, N. Aitsaadi, and M. Gueroui, "5GC-Observer: A non-intrusive observability framework for cloud native 5G system," in *NOMS*, 2023.
[15] A. Zhao, "Non-intrusive observability leveraging distributed tracing for AI-driven root cause analysis in 5G System," Ph.D. dissertation, Université Paris-Saclay, 2026.
[16] OpenAirInterface Software Alliance, "OpenAirInterface 5G Radio Access Network," [Online]. Available: https://openairinterface.org/. Accessed: 2026-05-30.
[17] Open5GS Project, "Open5GS," [Online]. Available: https://open5gs.org/. Accessed: 2026-05-30.
[18] UERANSIM Authors, "UERANSIM," [Online]. Available: https://github.com/aligungr/UERANSIM. Accessed: 2026-06-13.
[19] Software Radio Systems, "srsRAN Project," [Online]. Available: https://www.srsran.com/. Accessed: 2026-05-30.
[20] R2Lab. "An anechoic wireless testbed for reproducible research." https://r2lab.inria.fr/.
[21] SLICES-RI. "Scientific LargeScale Infrastructure for Computing/Communication Experimental Studies." https://www.slices-ri.eu/.
[22] Y. Amami, C. Barakat, and T. Turletti, "Demonstrating slice-aware latency decomposition in cloud-native 5G systems," in *Proc. MSWiM*, Paris, France, 2026, to appear.