Pull-based Scraping vs. eBPF Auto-instrumentation: Overhead, Coverage, and Trade-offs

Victor Ilchev

Supervisor(s): Nitinder Mohan, Sehan Samarakoon Mudiyanselage

EEMCS, Delft University of Technology, The Netherlands

A Thesis Submitted to EEMCS Faculty Delft University of Technology,
In Partial Fulfilment of the Requirements
For the Bachelor of Computer Science and Engineering
June 20, 2026

Name of the student: Victor Ilchev
Final project course: CSE3000 Research Project
Thesis committee: Nitinder Mohan, Sehan Samarakoon Mudiyanselage, Jérémie Decouchant

An electronic version of this thesis is available at https://repository.tudelft.nl/.

Abstract

Cloud-native 5G core networks transform network functions into containerised microservices, which simplifies their management but fragments their observability across multiple telemetry layers. Monitoring these systems requires balancing between visibility and the overhead created by the control plane being observed. This paper evaluates two fundamentally different collection paradigms: *pull-based scraping* via Prometheus and *eBPF auto-instrumentation* via Grafana Beyla, on a live Open5GS 5G core deployed on a three-node Kubernetes cluster. Each stack is deployed in isolation and in combination: resource overhead is quantified across multiple granularity settings, scalability is measured by changing the number of Network Functions (NFs) being monitored at a time, and fault-detection coverage is assessed over 22 injected scenarios across five fault classes, using Chaos Mesh for controlled injection. Prometheus incurs substantially higher monitoring stack overhead; Beyla's sampling rate has negligible effect on the cost, because kernel uprobes fire on every HTTP/2 library call regardless of the sampling decision. For the fault observability, across all 3 runs, Beyla flags all 22 fault types in at least one of those runs, while Prometheus misses only one (NRF cascade failure). However throughout all three runs together, only 10 / 22 faults are detected reliably by both methods. Per-run reliability favours Prometheus (87.9% vs. 81.8%). We conclude that Beyla offers broader fault-type coverage at lower overhead, but Prometheus provides more consistent detection per individual injection.

1 Introduction

The transition of 5G core networks toward cloud-native architecture has replaced monolithic hardware with a distributed Service-Based Architecture (SBA), transforming network functions (NFs) into containerised microservices orchestrated by a platform such as Kubernetes [3; 8]. While this shift improves scalability and automation, it introduces a new layer of complexity: system state and behaviour must now be inferred from fragmented telemetry spanning multiple NFs, containers, the orchestration platform, and host infrastructure. As 5G continues to support increasingly demanding applications, operators face a critical trade-off between deep system visibility and the strict performance requirements of the 3GPP control plane [1].

Two architecturally distinct collection mechanisms define the primary solutions of this trade-off. The first is *pull-based scraping*: Monitoring software (like Prometheus in our case) periodically polls each NF's `/metrics` HTTP endpoint at a fixed scrape interval, collecting application counters and container resource metrics from the Container Advisor (cAdvisor). This approach is well-established in production, but captures only what application developers choose to expose. The second is *eBPF auto-instrumentation*: Auto-instrumentation software (like Grafana Beyla in this case) attaches eBPF uprobes to the HTTP/2 and TLS library entry/exit points of each NF process, reconstructing complete inter-NF request (latency, status code, throughput) transparently without any application changes [7].

These two main solutions are not exhaustive: cloud-native observability also employs log aggregation (e.g., Loki, Elasticsearch), service-mesh sidecar injection (Istio/Envoy), and synthetic monitoring via active RTT probing. This paper focuses on pull-based scraping and eBPF auto-instrumentation as the two most architecturally distinct and widely deployed collection mechanisms for metrics. Log-based (Loki) and RTT-based signals are included as supplementary reference channels in the fault-detection evaluation (Section 4.5), but are not the primary subject of the overhead comparison. Importantly, each method is evaluated using its own native signal type: Prometheus via infrastructure and application metrics; Beyla via span-derived HTTP observables. The comparison, therefore, reflects each architecture's inherent capability, not a common signal type shared between them.

Additional motivation comes from the Kubernetes orchestration layer. Standard pod status events, like CrashLoopBackOff or OOM kills, do not fire during CPU congestion, network delays, or protocol-level anomalies. Such faults leave pods in a `Running` state with no status change [6]. While container-level resource metrics from cAdvisor and Prometheus partially bridge this gap, they remain blind to protocol-layer anomalies that do not cause observable resource spikes. This limitation highlights the need to evaluate what each telemetry architecture can and cannot observe across a diverse fault space.

Despite growing interest in cloud-native observability for mobile networks [5], no prior work has directly benchmarked pull-based scraping against eBPF auto-instrumentation under controlled fault injection in a live 5G core with explicit overhead measurement. This paper addresses that gap. The central research question is: *How do pull-based and eBPF auto-instrumentation telemetry methods compare in overhead, fault-detection coverage, and scalability in a live 5G core?*

Four sub-questions guide the investigation:

- **RQ1a** - How do the two methods compare in CPU and memory overhead at equivalent collection rates?
- **RQ1b** - What are the trade-offs between telemetry granularity and monitoring cost, and where do returns diminish?
- **RQ1c** - How does overhead scale with the number of pods monitored, and traffic intensity?
- **RQ1d** - What is the fault-detection coverage of each method across five fault classes, and what is the observability gap between them?

To answer these questions, an Open5GS 5G core is deployed on a three-node Kubernetes cluster and is then subjected to 22 fault scenarios spanning five fault classes: resource exhaustion, pod crashes, network delay, network partition, and protocol/dependency failures using Chaos Mesh

for controlled injection. Pull-based scraping is evaluated at scrape intervals of 1s, 5s, and 15s, while eBPF auto-instrumentation is performed at sampling rates of 10%, 50%, and 100%. Scalability experiments cover 1, 2, ... 8 NF's monitored at a time. Fault detection is assessed using a rolling z-score detector ($Z > 3.0$, applied to 30-second windows). The results show substantially higher Prometheus overhead, similar fault detection per class (compared to Beyla) (Figure 6), and higher per-run detection reliability.

The remainder of this paper is organised as follows. (Section 2) provides background and reviews related work. (Section 3) describes the experimental platform and methodology. (Section 4) presents and discusses the results for each sub-question. (Section 5) reflects on responsible research considerations. (Section 6) concludes and outlines limitations and future work.

2 Background and Relevant Work

2.1 5G Core Network Functions

A 5G core deployment consists of several cooperating NFs. The Access and Mobility Management Function (AMF) is the entry point for User Equipment (UE) registration and mobility procedures. The Session Management Function (SMF) establishes and tears down PDU sessions, instructing the User Plane Function (UPF) on how to forward user-plane traffic. The Network Repository Function (NRF) acts as a service registry, enabling NFs to discover one another at runtime. The Service Communication Proxy (SCP) routes SBA messages between NFs. Supporting functions include the Policy Control Function (PCF), the Unified Data Management (UDM), and the Authentication Server Function (AUSF). In a cloud-native deployment, each of these runs as one or more Kubernetes pods, communicating over HTTP/2 using the SBA reference-point model [1]. A single UE registration involves a chain of synchronous NF-to-NF calls - AMF to NRF, AMF to AUSF, AMF to UDM, AMF to SMF - meaning a fault anywhere in the chain can silently damage end-to-end service without triggering any pod restart.

2.2 eBPF as a Kernel Instrumentation Primitive

Extended Berkeley Packet Filter (eBPF) allows user-supplied programmes to run inside the Linux kernel in a sandboxed, JIT (just-in-time) compiled virtual machine. Programmes are attached to kernel hooks: kprobes, uprobes, tracepoints, and socket filters, and communicate with user space via lock-free ring buffers [10]. Grafana Beyla uses uprobes on the TLS and HTTP/2 library entry and exit points of each NF process to reconstruct complete request-response pairs (latency, status code, and payload size), without any modification to the NF binary. Similar kernel-level eBPF instrumentation has been proposed for cloud-native microservices [7; 8]. Because the hook fires on every library function invocation, overhead scales with request rate: at low traffic volumes, the cost is negligible, but under high UE counts generating sustained NF-to-NF signalling, the per-request kernel-to-user-space copy and BPF map lookup accumulate into a measurable CPU and memory penalty [8].

2.3 Related Work

Soldani et al. [8] survey eBPF across networking, security, and observability for 5G and 6G, arguing that kernel-level instrumentation is the natural fit for cloud-native NFs that cannot be recompiled. The survey does not include overhead measurements relative to pull-based scraping, and fault detection is not evaluated.

Sharma and Nadig [7] build a complete eBPF observability stack for cloud-native microservices and demonstrate lower overhead than conventional agent-based collectors (Node Exporter and cAdvisor). The comparison baseline, however, is other observability agents rather than a Prometheus scraper, and the target workload is a generic microservice benchmark rather than a 5G control plane.

Menin et al. [6] use observability data from open-source 5G cores to produce explainable performance diagnoses, moving beyond black-box performance testing to expose internal NF interactions and bottlenecks. Their work motivates fine-grained telemetry for 5G cores but does not compare collection methods, quantify instrumentation overhead, or evaluate fault detection under controlled injection.

Matysiak et al. [5] address observability for telco edge deployments, focusing on aggregating and acting on telemetry under edge resource constraints. They quantify the overhead of their Telco-Observer platform but do not compare distinct collection architectures (e.g., pull-based scraping vs. eBPF) or evaluate fault detection under controlled injection.

Across these contributions, eBPF is consistently positioned as the higher-fidelity option, but no prior study has directly measured its overhead against pull-based scraping under the same 5G workload, varied collection granularity, or characterised which fault classes each method can and cannot detect.

3 Methodology and Experimental Setup

3.1 Platform

All experiments were conducted on a three-node *kind* (Kubernetes-in-Docker) cluster (one control-plane node and two worker nodes) hosted on a single workstation. Open5GS (Gradiant Helm chart v2.3.4) was deployed with the full set of 5G SA core NFs: AMF, SMF, UPF, NRF, SCP, AUSF, UDM, PCF, and UDR. UERANSIM (Gradiant gNB chart v0.2.6 / UE chart v0.1.2) simulates the radio access network, providing a software gNB and configurable numbers of UE processes. The cluster was completely reset between every experimental run to purge residual GTP tunnels, Prometheus TSDB write-ahead log backlogs, and eBPF trace buffers.

3.2 Telemetry Stacks

Prometheus. The `kube-prometheus-stack` Helm chart deploys the full Prometheus monitoring stack, comprising the Prometheus server, Alertmanager, Grafana, node-exporter (DaemonSet, one pod per node), and kube-state-metrics. Container-level CPU and memory metrics are provided by cAdvisor, embedded in the kubelet. Open5GS exposes application-level counters (registration requests, PDU-session counts, PFCP statistics, GTP errors) via per-NF `/metrics` endpoints. The scrape interval is varied across 1s,

#	Fault description	Class
01	CPU stress - AMF	Resource
02	Memory pressure - UPF	Resource
03	Pod crash - AMF	Crash
04	Network delay - gNB↔AMF	Delay
05	Network partition - AMF↔SCP	Partition
06	Packet loss - UPF	Partition
07	Pod crash - SMF	Crash
08	CPU stress - SCP	Resource
09	Network delay - NRF	Delay
10	PFCP session flood - UPF	Protocol
11	PFCP session deletion - UPF	Protocol
12	PFCP modification drop - UPF	Protocol
13	PFCP modification duplicate - UPF	Protocol
14	Infrastructure packet loss - UPF	Partition
15	NRF cascade failure	Protocol
16	CPU stress - AUSF	Resource
17	Network delay - SCP	Delay
18	CPU stress - NRF	Resource
19	Pod crash - UDM	Crash
20	MongoDB pod kill	Protocol
21	N2 partition - AMF↔gNB	Partition
22	Memory pressure - AMF	Resource

Table 1: 22 fault scenarios used in the fault-detection experiment.

5s, and 15s in separate experimental runs. All other stack components are kept constant.

Beyla. Grafana Beyla (v3.9.5 or later; earlier releases cannot instrument Open5GS due to the kernel BPF verifier's instruction limit) is deployed as a DaemonSet, one pod per worker node. Each pod attaches eBPF uprobes to the HTTP/2 and TLS library entry/exit points of every NF process on that node, transparently reconstructing the inter-NF requests. Those are exported via an OpenTelemetry collector to a Jaeger all-in-one backend (v4.7.0, in-memory storage). The trace sampling rate is varied across 100%, 50%, and 10% in separate runs. All other stack components here are also kept constant.

3.3 Fault Injection

Chaos Mesh (v2.7.2) is used for programmatic fault injection following chaos-engineering principles [2]. The 22 fault scenarios are grouped into five classes: (1) *resource exhaustion*: CPU stress or memory pressure on a single NF; (2) *pod crash*: abrupt container termination triggering a Kubernetes restart; (3) *network delay*: added latency on a pod's virtual network interface; (4) *network partition*: complete traffic blackout between two NFs; and (5) *protocol/dependency failure*: PFCP session-level anomalies (floods, deletions, and message manipulation on the N4 interface), cascading NRF de-registration, and database pod termination. (Table 1) lists all 22 scenarios.

3.4 Data Collection

Each fault experiment follows a three-phase timeline: a 10-minute *pre-fault* baseline window, a 5-minute *fault injection* window, and a 5-minute *post-fault* recovery window. During each phase, Prometheus scrapes all endpoints at 5s intervals;

Method	Signals
Prometheus	Container and pod metrics; Open5GS application counters
Beyla	Per-NF HTTP duration, error rate, and request rate

Table 2: Detection signals per method.

Beyla exports spans at 100% sampling to Jaeger; Promtail forwards NF container logs to Loki; and a heartbeat script measures data-plane round-trip time (RTT) every second via ICMP pings through the UE tunnel interface. All runs use 50 concurrent UEs maintaining PDU sessions throughout.

For the overhead and granularity sweep experiments, the scrape interval (Prometheus) and sampling rate (Beyla) are varied independently over 5-minute steady-state runs with 50 concurrent UEs and six replications per configuration. For the scalability experiment, the number of actively scraped NF endpoints is varied from 1 to 8 at fixed UE load (15 UEs); limitations of this sweep are discussed in (Section 4.3).

3.5 Detection Algorithm

The two collection methods produce fundamentally different signal types - Prometheus yields infrastructure and application metrics, while Beyla yields HTTP observables. (Table 2) lists the specific signals fed into the detector for each method. The signals are not comparable in kind; the detector applies the same statistical procedure to each method's native outputs to ensure a fair comparison.

A unified rolling z-score detector is applied identically to both primary methods to eliminate detector-design bias. Each fault is injected three times each in its own clean cluster environment. (Table 5) shows how many of those runs each method flags (out of 3), and (Figure 6) groups faults detected reliably (3/3). For each time series, the pre-fault window establishes a baseline with mean μ and standard deviation σ. A fault is declared detected if any signal from that method exceeds $Z = (x - \mu)/\sigma_{\text{eff}} > 3.0$ during the fault window, where x is the mean of a 30-second rolling window and $\sigma_{\text{eff}} = \max(\sigma, 0.02|\mu|, \epsilon)$. Before a metric is tested against this threshold, three checks suppress false positives: *(i)* the robust denominator σ_{eff} prevents near-constant counters (e.g. pod-ready gauges) from crossing the boundary on sampling noise alone: a shift must exceed approximately 6% of the baseline magnitude to reach $Z = 3.0$; *(ii)* insufficient baseline data causes the z-score to be discarded rather than evaluated (Prometheus requires at least two pre-fault samples per series; Beyla requires at least 30 per NF service); and *(iii)* x is the mean of a 30-second rolling window, not a single sample, so a transient spike cannot trigger detection without a sustained deviation from μ. A z-score threshold is chosen because it requires no training data beyond the pre-fault window and produces interpretable thresholds. More sophisticated ML-based predictors that combine trace logs and monitoring metrics [4], or deep-learning models for 5G NF-interaction anomalies [9], may change the absolute detection rates but would affect telemetry quality with detector sophistication; the simple z-score isolates the former. The threshold $Z = 3.0$ (three standard deviations from the pre-fault mean)

Configuration	CPU (m)	Mem (MiB)
No telemetry	-	-
Prometheus (5s)	69.4	1 004
Beyla (100%)	1.5	110
Both (5s + 100%)	72.6	1 128

Table 3: Monitoring stack overhead by configuration (50 UEs, steady state). CPU in millicores; memory in MiB. Values are medians across six runs.

is a standard statistical criterion for anomaly flagging.

The per-run signal strength reported in (Section 4.5) is the maximum z-score per method, normalised to $[0, 1]$ using a log-scale mapping: $\text{norm}(Z) = \min\bigl(\log_{10}(1 + Z)/2,\ 1\bigr)$, so that $Z = 3 \to 0.30$, $Z = 10 \to 0.52$, and $Z \geq 100 \to 1.0$. Log scaling compresses extreme outliers (like a pod crash producing $Z > 200$ in restart counters) while preserving visual differentiation near the threshold. For Beyla, z-scores are computed at the per-NF-service level, and the maximum across all services is taken. Services with fewer than 30 pre-phase data points are excluded to avoid unreliable estimates for low-traffic NFs.

3.6 Reproducibility

All experiment scripts, Chaos Mesh manifests, Helm values files, and Kubernetes configuration files are available at https://github.com/V1K1NGbg/open5gs-observability-benchmark. The cluster can be reproduced with `kind` and the provided `cluster-start.sh` script.

4 Results

4.1 RQ1a: Monitoring Overhead

(Table 3) summarises the monitoring overhead for the baseline (no telemetry), Prometheus-only, Beyla-only, and combined configurations. Overhead is the aggregate resource consumption of the monitoring stack (all monitoring-namespace pods summed at each 5-second timestamp and averaged over the run), excluding Open5GS NF pods.

The Prometheus monitoring stack is substantially heavier than the Beyla push path in absolute terms: 69.4 m CPU and 1 004 MiB across the monitoring namespace, compared to 1.5 m CPU and 110 MiB for Beyla. This $\sim$46$\times$ CPU difference reflects the broader scope of the Prometheus platform (TSDB, dashboards, alerting, log shipping) vs Beyla's single-purpose tracing role. The combined configuration adds only $\sim$3.2 m CPU and $\sim$24 MiB beyond Prometheus alone. (Figure 1) shows the overhead decomposition.

(Figure 2) complements the median values in (Table 3) by showing the full per-run distribution across all seven sweep conditions. Prometheus CPU boxes span a wide range at aggressive scrape intervals (especially 1s), while Beyla conditions cluster tightly at 1-3 m CPU across six runs. A Mann-Whitney U test (a non-parametric rank test that compares two independent samples without assuming a normal distribution) confirms that the Prom 5s and Beyla 100% CPU distributions differ significantly ($p = 0.0022$) (the p-value is the probability of observing a difference at least this large if both methods had the same underlying CPU distribution), formalising the pull-versus-push gap as statistically robust rather than a side effect of run-to-run variance.

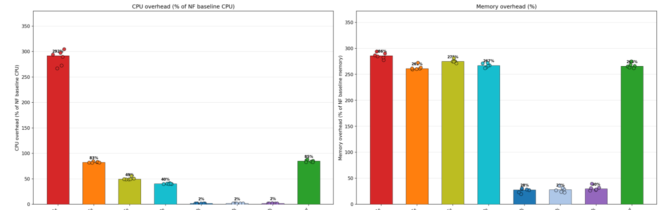

Figure 1: Monitoring overhead as a percentage of NF workload CPU (left) and memory (right). Bars are medians across six runs; individual run points are overlaid. Dashed vertical line separates pull (Prometheus) from push (Beyla) configurations. Combined is Prom 5s + Beyla 100% total overhead (single bar).

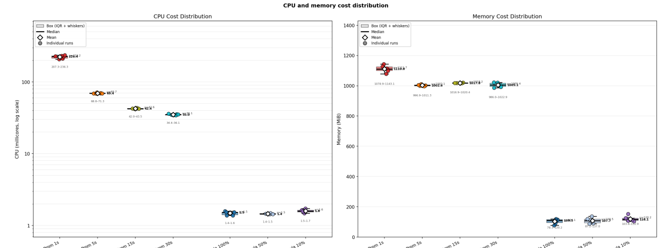

Figure 2: Per-run CPU (left, log scale) and memory (right) distributions for all seven monitoring conditions. Boxes show IQR with whiskers; thick lines are medians; diamonds are means; dots are individual runs. Dashed separator divides Prometheus (pull) from Beyla (push). Mann-Whitney U test: Prom 5s vs. Beyla 100% CPU, $p = 0.0022$ (significant).

4.2 RQ1b: Granularity Trade-offs

Prometheus scrape interval. (Table 4) and (Figure 3) show the effect of scrape interval on the Prometheus server component (the primary interval-sensitive component; node-exporter and kube-state-metrics overhead is approximately constant across intervals). Reducing the interval from 5s to 1s increases the Prometheus server CPU by 3.5$\times$ (24.4 m$\to$86.4 m). The memory footprint of the TSDB is largely insensitive to the interval because Prometheus compresses data in chunks; the 15s and 5s configurations differ by less than 3 MiB in server-side memory.

The 5s interval represents an efficient knee point: dropping to 15s saves an additional 10.6 m CPU (a 43% reduction relative to 5s versus the 72% reduction from 1s to 5s).

eBPF sampling rate. An interesting result is that the sampling rate for Beyla has essentially no effect on overhead: CPU varies by less than 0.2 m and memory by less than 8 MiB across 10%-100% sampling rates (Table 4). Also, interestingly, the 10% configuration's CPU is slightly higher than at 100% (1.6m vs. 1.5m). This could potentially be happening because the eBPF uprobe fires on every HTTP/2 library entry, regardless of the sampling decision; the overhead budget is dominated by the kernel-to-user-space context switch and BPF map lookup, which are invariant to sampling. The

Configuration	CPU (m)	Mem (MiB)
Prometheus 1s	86.4	501
Prometheus 5s	24.4	419
Prometheus 15s	13.8	422
Beyla 100%	1.5	110
Beyla 50%	1.5	108
Beyla 10%	1.6	116

Table 4: Effect of granularity on Prometheus server and Beyla component overhead. CPU is the Prometheus server process (interval-sensitive) or Beyla pods (sampling-sensitive).

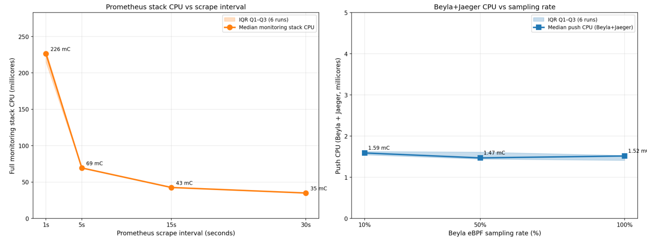

Figure 3: Full Prometheus monitoring stack CPU vs. scrape interval (left) and Beyla push CPU vs. sampling rate (right). Lines are medians across six runs; shaded bands show the IQR (Interquartile range) (Q1-Q3). Prometheus exhibits a steep convex cost curve (226 m at 1s to 35 m at 30s); Beyla push overhead is essentially flat across all sampling rates (range: 1.47-1.59 m CPU, 8% variation).

additional cost at low sampling rates reflects the overhead of the per-request Bernoulli sampling test in user space (each request is independently kept or dropped for export with a fixed probability (e.g., 10%)). This finding has a practical implication: *reducing eBPF sampling provides no meaningful resource saving*, making the 10% and 50% configurations strictly worse than 100% (same cost, fewer spans). (Figure 3) visualises both effects.

4.3 RQ1c: Scalability

(Figure 4) shows how collection CPU scales with the number of actively monitored NF endpoints at a fixed load of 15 UEs. For Prometheus, process CPU grows from 27.4 m (1 NF) to 35.4 m (8 NFs), an average increase of roughly 1.1 m per additional scrape target; growth is steeper at low counts ($\sim$2.3 m per NF from 1 to 3) and flattens above six endpoints, consistent with a fixed per-target scrape cost (one HTTP round-trip plus TSDB write per cycle) amortised across a largely constant baseline. For Beyla, container CPU grows from 3.1 m (1 NF) to 5.0 m (8 NFs), averaging $\sim$0.3 m per additional instrumented executable; the curve plateaus after four NFs ($\sim$4.9 m) even as scope expands. The IQR across different choices of *which* NFs to monitor is narrow at each point for both methods, confirming that overhead scales with endpoint count rather than NF identity.

4.4 Data-Plane RTT Impact

Infrastructure overhead alone does not capture whether telemetry affects the latency experienced by end users. As shown in (Figure 5) data-plane round-trip time measured

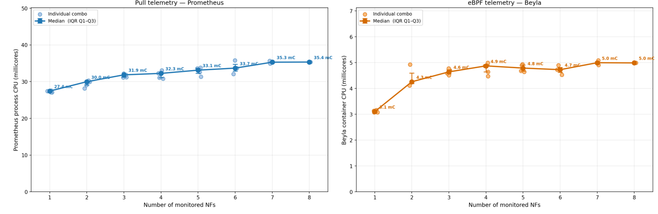

Figure 4: Collection CPU vs. number of monitored NF endpoints (15 UEs, 5s scrape interval for Prometheus). Left: Prometheus process CPU; right: Beyla container CPU. Each dot is one NF combination (horizontally jittered); the line connects medians; error bars show IQR (Q1–Q3). Prometheus grows from 27.4 m (1 NF) to 35.4 m (8 NFs); Beyla from 3.1 m to 5.0 m.

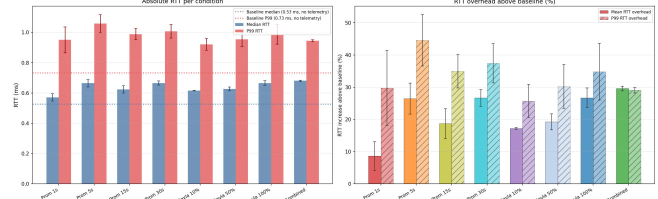

Figure 5: Data-plane ping RTT per telemetry condition. Left: absolute median and P99 RTT with IQR/2 error bars; dotted lines mark the no-telemetry baseline. Right: RTT overhead as a percentage above baseline (solid = median; hatched = P99). RTT is measured on the UE data path (`uesimtun0` $\rightarrow$ UPF), not registration procedure latency.

as continuous ICMP pings from the UE tunnel interface (`uesimtun0`) through the UPF to the external gateway. The no-telemetry baseline is 0.53 ms (median) and 0.73 ms (P99). All eight configurations (every Prometheus interval, every Beyla sampling rate, and the combined stack) show only a mild increase above baseline: median RTT remains below 0.68 ms and P99 below 1.06 ms across conditions. Relative overhead ranges from roughly 10-30% above baseline for the median and up to $\sim$45% for P99, but absolute values stay well below 1 ms. No single configuration stands out as systematically worse; values are not ordered by scrape interval or sampling rate. Because Beyla instruments HTTP/2 control-plane library calls via kernel uprobes and does not intercept the IP/GTP user-data path, eBPF tracing adds negligible data-plane latency (a result consistent with the similar RTT across all Beyla conditions).

Taken together with the results from (Figure 2), these RTT measurements show that the dominant cost of telemetry is borne by the monitoring infrastructure rather than by the user data path. The $\sim$46$\times$ CPU gap between Prometheus and Beyla (Table 3) is therefore control-plane and platform heavy, not so much latency heavy: P99 data-plane RTT increases by at most 0.33 ms ($\sim$45% relative to the 0.73 ms baseline) even under the most demanding Prometheus scrape interval (1s).

4.5 Fault Detection and Observability Gap

Detection coverage. (Figure 6) summarises how reliable is the detection for each method: a fault counts only if flagged in all three repeated injection runs (3/3). Only 10 of 22 faults

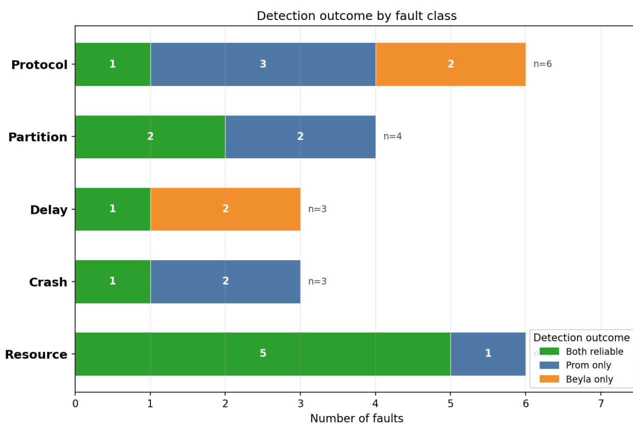

Figure 6: Reliable detection by fault class (detected in all 3 runs). Stacked bars: both methods (green), Prometheus only (blue), Beyla only (orange). Counts per class: Resource: 5/1/0; Crash: 1/2/0; Delay: 1/0/2; Partition: 2/2/0; Protocol: 1/3/2 (both / Prom-only / Beyla-only).

are detected reliably by both methods; 8 by Prometheus only and 4 by Beyla only. (Table 5) lists the per-fault detection fraction (runs detected out of 3). Per-run reliability averages 87.9% for Prometheus versus 81.8% for Beyla.

Why does Prometheus miss fault 15? The NRF cascade failure is the only fault with 0/3 Prometheus detections (prom$z = 2.58$, just below the $z = 3.0$ threshold). The cascade unfolds gradually: the NRF begins rejecting service-discovery requests, so NFs that rely on it cannot locate their dependencies. From Prometheus's perspective, the NRF pod remains in a `Running` state with no restart; CPU and memory are stable; and application-layer registration counters increase by a small amount that falls short of the threshold. Beyla detects the cascade via elevated HTTP/2 error rates on NF-to-NF calls that fail because NRF cannot respond (beyla $z = 3.93$, narrowly above threshold). Both signals are weak and near-threshold, suggesting that a z=3.0 threshold may be too conservative for subtle service-registry failures.

Beyla per-run reliability. Several faults that Beyla detects in at least one run are not reliably detected in every run (81.8% on average run). The analysis script requires a minimum of 30 pre-phase data points per NF service before computing a per-service z-score; under 50-UE load, low-traffic NFs such as AUSF and NRF generate few HTTP/2 spans per measurement window, and some runs fall below this threshold, causing those detections to be silently skipped. Prometheus counters, by contrast, are collected from every scrape regardless of traffic volume, which explains its higher per-run reliability (87.9%).

(Figure 7) shows per-fault signal strength and run fractions (e.g., 2/3). (Figure 6) shows reliable detection by fault class.

Cost-benefit analysis. To compare detection value per monitoring resource consumed, we define the following efficiency metric (our own formulation, not derived from prior

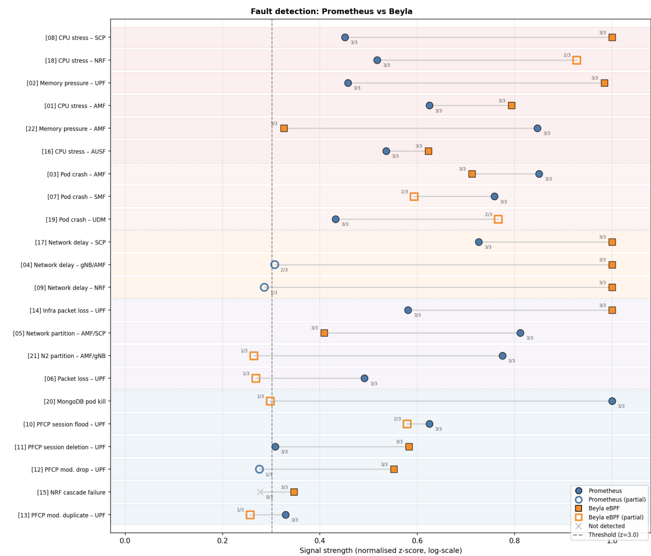

Figure 7: Detection signal strength per fault (normalised log-scale z-score, 0-1) for Prometheus (blue circles) and Beyla (orange squares). Labels next to each marker show the run fraction (e.g., 2/3 = detected in 2 of 3 runs). Filled markers = detected in all runs (reliable); hollow = borderline (some runs); $\times$ = not detected. The dashed vertical line marks the $z = 3.0$ detection threshold. Fault #15 (NRF cascade failure, highlighted) is the only fault with 0/3 Prometheus detections.

work):

$$E = \frac{\text{detection rate}}{\text{monitoring CPU (m)}} \quad (1)$$

where the detection rate is expressed as a fraction (1.00 for 100%), and CPU is the Prometheus server or Beyla component millicores (per-component cost, to compare the collection mechanisms rather than the full platforms). (Table 6) reports E for all configurations. Beyla at 100% sampling achieves the highest practical efficiency ($E = 0.66$), roughly $10\times$ above Prometheus at 15s ($E = 0.069$). All Beyla sampling rates dominate every Prometheus interval; because sampling does not reduce CPU cost, the 50% and 10% configurations offer no resource advantage and export fewer spans at essentially the same cost.

4.6 Discussion

The central finding is that the two methods are complementary rather than substitutable. Only 10 of 22 faults are detected reliably by both (Figure 6). Prometheus achieves more reliable detection for Protocol and Crash faults. Beyla does so for Delay faults. On per-run detections (Table 5), Beyla totals 54/66 detections and Prometheus 58/66. Fault #15 is never detections with Prometheus, but is detected in 3/3 with Beyla. Still, Per-run reliability favours Prometheus (87.9% vs. 81.8%).

Prometheus exposes application and infrastructure-level indicator, such as session counts and error rates, that remain available under low traffic and reflect state changes encoded by the NF implementation. Beyla, by contrast, reconstructs per-request HTTP/2 latency, status, and throughput at the library boundary, which provides stronger sensitivity to inter-

#	Fault	Class	Prometheus	Beyla
01	CPU stress - AMF	Resource	3/3	3/3
02	Memory pressure - UPF	Resource	3/3	3/3
03	Pod crash - AMF	Crash	3/3	3/3
04	Network delay - gNB/AMF	Delay	2/3	3/3
05	Network partition - AMF/SCP	Partition	3/3	3/3
06	Packet loss - UPF	Partition	3/3	1/3
07	Pod crash - SMF	Crash	3/3	2/3
08	CPU stress - SCP	Resource	3/3	3/3
09	Network delay - NRF	Delay	1/3	3/3
10	PFCP session flood - UPF	Protocol	3/3	2/3
11	PFCP session deletion - UPF	Protocol	3/3	3/3
12	PFCP modification drop - UPF	Protocol	1/3	3/3
13	PFCP modification duplicate - UPF	Protocol	3/3	1/3
14	Infrastructure packet loss - UPF	Partition	3/3	3/3
15	NRF cascade failure	Protocol	0/3	3/3
16	CPU stress - AUSF	Resource	3/3	3/3
17	Network delay - SCP	Delay	3/3	3/3
18	CPU stress - NRF	Resource	3/3	2/3
19	Pod crash - UDM	Crash	3/3	2/3
20	MongoDB pod kill	Protocol	3/3	1/3
21	N2 partition - AMF/gNB	Partition	3/3	1/3
22	Memory pressure - AMF	Resource	3/3	3/3
	Total		**58/66**	**54/66**

Table 5: Per-fault detection fraction: runs in which the detector fired, out of 3 repeated injections per fault. Cell colour: red = 0/3; orange = 1/3; yellow = 2/3; green = 3/3 (reliable). Values of 3/3 match the reliable-detection category in (Figure 6).

Configuration	CPU (m)	Det.	E
Beyla 100%	1.5	100%	**0.66**
Beyla 50%	1.5	100%	0.66
Beyla 10%	1.6	100%	0.63
Prometheus 15s	13.8	95.5%	0.069
Prometheus 5s	24.4	95.5%	0.039
Prometheus 1s	86.4	95.5%	0.011

Table 6: Cost-benefit efficiency across all configurations. E = detection fraction/monitoring CPU (m); higher is better. CPU is the Prometheus server or Beyla component only.

NF delay and to cascading failures that manifest as protocol errors without corresponding pod-level alerts.

The NRF cascade failure (fault #15) illustrates this division. A gradual service degradation propagates as elevated HTTP error rates on NF-to-NF discovery traffic, which Beyla observes directly. Prometheus counters for the same event change slowly and, in this experiment, remain below the $z = 3.0$ threshold.

Among the configurations tested, Beyla at 100% sampling attains the highest detection-per-CPU-millicore ratio ($E = 0.66$), approximately 10× that of Prometheus at a 15 s scrape interval ($E = 0.069$), even with lower per-run consistency. Deploying both stacks together covers all 22 fault types in at least one run. Because eBPF sampling rate has negligible effect on CPU cost, reducing Beyla overhead requires limiting the number of instrumented pods rather than lowering the sampling rate.

5 Responsible Research

AI tool usage. This research used Claude (Anthropic) as an AI assistant for exploratory analysis, script-development support, and writing assistance. All experimental data were collected exclusively by the author on a local cluster; no synthetic or AI-generated data appear in the results. The experimental methodology, analysis decisions, and conclusions are the author's own. AI assistance was limited to language refinement and grammatical correction of the author's original drafts. All research findings are entirely original. All AI-generated suggestions were reviewed and validated by the author before inclusion.

Reproducibility. All experiment scripts (Bash), Chaos Mesh YAML manifests, Helm values files, Kubernetes configuration files, and the Python analysis scripts are available in the public repository at https://github.com/V1K1NGbg/open5gs-observability-benchmark. This repository was built in collaboration with the other members of the Observability for Intelligent Fault Management in Cloud-native B5G Networks research group. The cluster can be reproduced from scratch using the provided `cluster-start.sh` and `experiments/run_all_phases.sh` scripts with a `kind` installation. The only non-reproducible element is hardware timing jitter; all experiments use sufficiently long measurement windows (10 minutes pre-fault) to average over transient load spikes. The UERANSIM segfault above 50 concurrent UEs is a known limitation of the tested Helm chart version and is documented alongside the raw data.

Integrity. The fault-detection algorithm is identical for both methods (the same rolling z-score, threshold, and win-

dow length), preventing any inadvertent tuning advantage for either. Overhead baselines were collected with telemetry fully disabled to ensure measurements reflect only the monitoring stack. The 22 fault scenarios were specified before analysis began and were not modified after observing results.

Ethical considerations. All experiments were conducted entirely within a simulated, isolated local environment; no real network infrastructure, user data, or production systems were involved. There are no privacy, safety, or third-party impact considerations beyond ordinary software-systems research practice.

6 Conclusion and Future Work

This paper compared pull-based scraping (Prometheus) and eBPF auto-instrumentation (Grafana Beyla) for observability in a cloud-native 5G core, characterised across three dimensions: resource overhead, granularity trade-offs, and fault-detection coverage. The key findings are as follows.

RQ1a (Overhead). The Prometheus monitoring stack consumes 69.4 m CPU and 1 004 MiB across the full monitoring namespace, compared to 1.5 m CPU and 110 MiB for Beyla+Jaeger. The gap reflects the broader scope of the Prometheus platform (TSDB, dashboards, alerting, log shipping) versus Beyla's single-purpose tracing role.

RQ1b (Granularity). Prometheus shows a clear diminishing-returns cost curve: moving from 1s to 5s scrape interval reduces the Prometheus server CPU by 72% (86.4 m→24.4 m). Moving to 15s reduces it by a further 43%. The 5s interval is an efficient operating point. For Beyla, sampling rate has a negligible effect on overhead (<0.2 m CPU variation across 10%–100%), because the kernel uprobe fires regardless of the sampling decision. Consequently, reducing Beyla's sampling rate provides no resource saving.

RQ1c (Scalability). At fixed UE load (15 UEs), Prometheus process CPU grows from 27.4 m (1 monitored NF) to 35.4 m (8 NFs) and Beyla container CPU from 3.1 m to 5.0 m, confirming endpoint-driven rather than traffic-driven collection cost for both methods. A UE-count sweep up to 200 UEs was attempted, but produced unreliable results above 50 UEs owing to UERANSIM gNB instability.

RQ1d (Fault detection). Only 10 of 22 faults are detected reliably by both methods (Figure 6). 8 are detected only by Prometheus and 4, only by Beyla at that threshold. Per-run fractions in (Table 5) total 58/66 for Prometheus and 54/66 for Beyla; fault #15 is 0/3 vs. 3/3. Per-run reliability favours Prometheus (87.9% vs. 81.8%). Detection efficiency (E, Eq. (1)) favours Beyla at all sampling rates ($E \approx 0.63$–0.66), with 100% sampling the recommended operating point. This is roughly 10× above Prometheus at 15s ($E = 0.069$).

Answer to the research question. Neither method dominates: reliable detection is split by fault class (Figure 6), while per-run fractions in (Table 5) show Prometheus ahead on total detections (58/66 vs. 54/66) but behind because of fault#15 (0/3 vs. 3/3). Prometheus offers higher per-run consistency, while Beyla offers lower overhead and better detection efficiency per CPU millicore. A combined deployment covers all 22 fault types in at least one run.

Limitations. Five main limitations apply. *(i)* Endpoint scalability was measured at fixed UE load (15 UEs) across 1-8 monitored NFs; UE-count scaling above approximately 50 concurrent UEs remains uncharacterised because UERANSIM segfaults at higher loads. *(ii)* All experiments use a single hardware configuration (kind on one workstation); results may differ on multi-server clusters with different CPU and memory proportions. *(iii)* The z-score detector is a simple univariate threshold; more sophisticated multi-signal anomaly detectors may shift the relative performance of the two methods. The single missed fault (#15) is near-threshold for both methods ($z = 2.58$ vs. $z = 3.93$), and a slightly relaxed threshold would close the gap entirely. *(iv)* Overhead and efficiency comparisons rest on CPU figures that are not directly comparable: (Table 3) sums all pods in the `monitoring` namespace for Prometheus (server, Grafana, Alertmanager, Loki, Promtail, node-exporters, kube-state-metrics) against the Beyla DaemonSet and Jaeger backend only, so the ∼46× gap and the cost-benefit metric E (Eq. (1)) compare a full observability platform to a single-purpose trace pipeline. The E denominator narrows further to the Prometheus server alone, even though fault detection consumes metrics from the wider stack (cAdvisor, kube-state-metrics, node-exporter, and NF `/metrics` endpoints). *(v)* Both stacks run inside the same cluster as the monitored NFs; the Prometheus self-scraping contribution is included in the reported overhead figures, and the self-monitoring cost cannot be fully separated.

Future work. Re-running UE-count scalability experiments with a patched UERANSIM release or an alternative UE simulator would validate whether collection overhead scales with concurrent registration bursts and identify the crossover point where Beyla becomes the dominant monitoring cost under high signalling load. Extending the fault catalogue to slow-drift faults (gradual memory leaks, CPU degradation) would test whether Prometheus counters detect the onset before system saturation. Finally, combining Prometheus and Beyla signals in a multi-modal detector could close the residual detection gap while reducing total overhead through selective Beyla deployment on high-throughput NFs only.

References

[1] 3rd Generation Partnership Project. System Architecture for the 5G System (5GS); (Release 18). Technical Specification TS 123.501 V18.12.0, 3GPP, 2026.

[2] Ali Basiri, Niosha Behnam, Ruud de Rooij, Lorin Hochstein, Luke Kosewski, Justin Reynolds, and Casey Rosenthal. Chaos engineering. *IEEE Software*, 33(3):35–41, 2016.

[3] Brendan Burns, Brian Grant, David Oppenheimer, Eric Brewer, and John Wilkes. Borg, omega, and kubernetes. *Communications of the ACM*, 59(5):50–57, 2016.

[4] Neha Kaushik, Harish Kumar, Vinay Raj, and Puneet Garg. Proactive fault prediction in microservices applications using trace logs and monitoring metrics. In *2024 International Conference on Progressive Innovations in*

Intelligent Systems and Data Science (ICPIDS), pages 410–415, 2024.

[5] Piotr Matysiak, Ilhem Fajjari, Halina Tarasiuk, and Marcin Ziółkowski. Cloud native observability for an enhanced orchestration at the telco edge. In *ICC 2025 - IEEE International Conference on Communications*, pages 886–891, 2025.

[6] Carlos Eduardo Menin, Igor Martins Silva, Vinícius Boff Alves, Gabriel Lando, Cristiano Bonato Both, José Marcos S. Nogueira, and Juliano Araujo Wickboldt. Explainable performance analysis of open-source 5g core network implementations via observability. In *2025 IEEE 11th International Conference on Network Softwarization (NetSoft)*, pages 397–405, 2025.

[7] Bhavye Sharma and Deepak Nadig. ebpf-enhanced complete observability solution for cloud-native microservices. In *ICC 2024 - IEEE International Conference on Communications*, pages 1980–1985, 2024.

[8] David Soldani, Petrit Nahi, Hami Bour, Saber Jafarizadeh, Mohammed F. Soliman, Leonardo Di Giovanna, Francesco Monaco, Giuseppe Ognibene, and Fulvio Risso. ebpf: A new approach to cloud-native observability, networking and security for current (5g) and future mobile networks (6g and beyond). *IEEE Access*, 11:57174–57202, 2023.

[9] Yawen Tan, Jiajia Liu, Yuanhao Li, and Jiadai Wang. Deep learning-based proactive anomaly detection for 5g core control plane network function interactions. *IEEE Transactions on Cognitive Communications and Networking*, 11(6):4210–4222, 2025.

[10] Marcos A. M. Vieira, Matheus S. Castanho, Racyus D. G. Pacífico, Elerson R. S. Santos, Eduardo P. M. Câmara Júnior, and Luiz F. M. Vieira. Fast packet processing with ebpf and xdp: Concepts, code, challenges, and applications. *ACM Comput. Surv.*, 53(1), February 2020.