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

Victor Ilchev • Supervisors: Nitinder Mohan, Sehan Samarakoon Mudiyanselage

CSE3000 Research Project • TU Delft, EEMCS • v.ilchev@student.tudelft.nl

How do pull-based scraping and eBPF auto-instrumentation compare in overhead, fault-detection coverage, and scalability in a live 5G core?

Background & Motivation

Cloud-native 5G cores run as microservice meshes on Kubernetes. Operators need **observability** to detect faults — but monitoring itself consumes CPU and memory on the control plane.

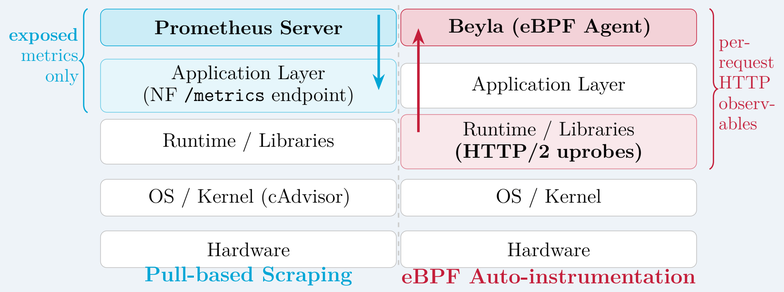
Two architecturally distinct **data-collection paradigms** define the trade-off:

Pull-based scraping (Prometheus)
The monitoring server *polls* each NF's `/metrics` HTTP endpoint at a fixed scrape interval. Collects application counters and cAdvisor container metrics — only what developers choose to expose.

eBPF auto-instrumentation (Grafana Beyla)
Attaches eBPF uprobes to HTTP/2 and TLS library entry/exit points, reconstructing inter-NF request latency, status, and throughput **transparently** — without application changes.

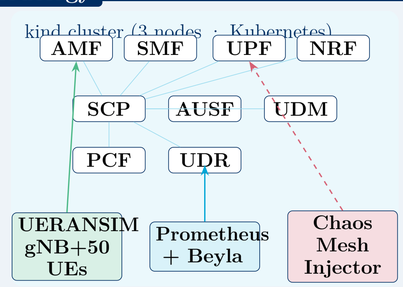
Each method is evaluated on its **native signal type**: Prometheus via infrastructure and application metrics; Beyla via span-derived HTTP observables. Loki (logs) and data-plane RTT pings are supplementary reference channels in fault detection, not primary overhead subjects.

How Data is Collected



Kubernetes pod events (CrashLoopBackOff, OOM) do **not** fire during CPU congestion, network delays, or protocol anomalies — pods remain `Running` with no status change. Application metrics and kernel-level traces are needed to cover this blindspot.

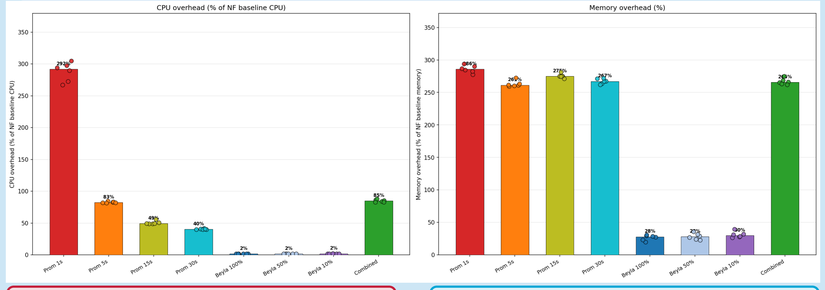
Testbed & Methodology



Component	Details
5G Core	Open5GS (Helm v2.3.4), 9 NFs
RAN	UERANSIM (1 gNB, 50 concurrent UEs)
Cluster	kind v0.27, 3-node Kubernetes
Fault Injection	Chaos Mesh 2.7.2, 22 scenarios / 5 classes
Pull-based	kube-prometheus-stack + cAdvisor
eBPF	Grafana Beyla ≥3.9.5 → Jaeger
Logs / Traces	Loki + Promtail; RTT via `uesimtun0` ping
Detection	Rolling z-score (Z>3.0, 30 s windows)

Experiment phases: 10 min baseline → 5 min fault → 5 min recovery. Each fault injected **3 times** in a clean cluster. Overhead sweeps: scrape 1/5/15 s; Beyla sampling 10/50/100%. Scalability: 1–8 NF scrape targets at 15 UEs.

RQ1a: Monitoring Stack Overhead (50 UEs, steady state)

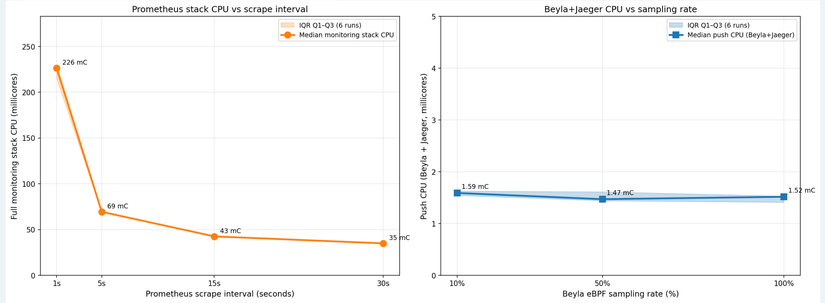


~46× more CPU — Prometheus stack vs. Beyla+Jaeger

1 004 MiB vs. 110 MiB — Prometheus vs. Beyla stack memory

Medians across 6 runs (monitoring-namespace pods only): Prometheus (5 s): 69.4 m CPU, 1 004 MiB; Beyla (100%): 1.5 m CPU, 110 MiB; Combined: 72.6 m CPU, 1 128 MiB. The gap reflects Prometheus's full platform (TSDB, Grafana, Alertmanager, node-exporter) vs. Beyla's single-purpose trace path. Mann-Whitney U: Prom 5s vs. Beyla 100% CPU, $p = 0.0022$.

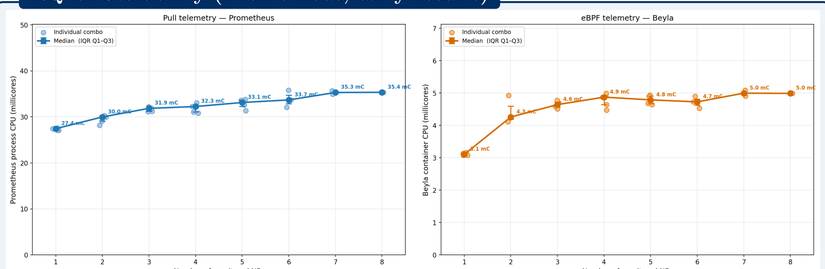
RQ1b: Granularity Trade-offs



Prometheus: 1 s→5 s scrape cuts server CPU by **72%** (86.4 m→24.4 m); 5 s→15 s saves a further **43%**. The 5 s interval is an efficient knee point; TSDB memory is largely insensitive to interval.

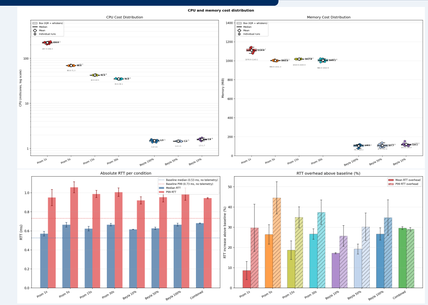
Beyla: Sampling rate has **negligible** overhead impact (<0.2 m CPU, <8 MiB across 10%–100%) because kernel uprobes fire on every HTTP/2 library call regardless of sampling. **10% and 50% are strictly worse than 100%** — same cost, fewer spans. Per-run detection rate is identical (81.8%) across all sampling rates.

RQ1c: Scalability (Prometheus, early results)



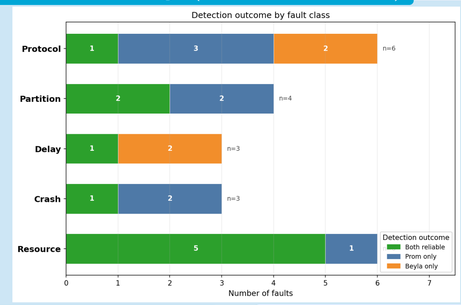
Prometheus process CPU grows from 27.4 m (1 scraped NF) to 32.0 m (3 NFs) at fixed 15-UE load — roughly 2.3 m per additional scrape target. Beyla endpoint scalability was not yet measured; UERANSIM instability above 50 UEs limits UE-count sweeps.

Statistical Comparison & Data-Plane RTT



Telemetry cost is borne by the **monitoring infrastructure**, not the user data path. Baseline median RTT: 0.53 ms; P99: 0.73 ms. All configurations keep median RTT <0.68 ms and P99 <1.06 ms (10–45% relative overhead). Beyla instruments control-plane HTTP/2 calls, not the IP/GTP user-data path — negligible data-plane penalty.

RQ1d: Fault Detection Coverage (22 Faults × 3 Runs)



10 / 22 reliable (3/3) — detected by **both** methods

87.9% per-run reliability — Prometheus (58/66 runs)

Reliable detection (3/3 runs): 10 both, 8 Prometheus-only, 4 Beyla-only (by fault class: Resource 5/1/0; Crash 1/2/0; Delay 1/0/2; Partition 2/2/0; Protocol 1/3/2).

Per-run totals: Prometheus 58/66 (87.9%); Beyla 54/66 (81.8%). Beyla flags **all 22 fault types** in at least one run; Prometheus misses only fault #15 (NRF cascade, 0/3 — $z = 2.58$, just below threshold). Beyla detects it via elevated HTTP/2 error rates ($z = 3.93$). Combined deployment covers all 22 fault types in at least one run.

Cost-benefit efficiency E = detection rate/CPU(m): Beyla 100%: $E = 0.66$; Prometheus 15 s: $E = 0.069$ — roughly **10×** higher efficiency for Beyla per CPU millicore.

Key Findings & Conclusions

1. **Neither method dominates.** The two paradigms are *complementary*, not substitutable: only 10/22 faults are reliably detected by both; 8 are Prometheus-only and 4 Beyla-only at the 3/3 threshold.
2. **Prometheus is far heavier in absolute overhead** (~46× CPU, ~9× memory vs. Beyla+Jaeger), reflecting the full observability platform scope.
3. **Prometheus offers higher per-run consistency** (87.9% vs. 81.8%), especially for Protocol and Crash faults; Beyla excels on Delay faults and the NRF cascade (#15).
4. **5 s scrape is the efficient Prometheus operating point;** detection rate (95.5%) is identical across 1/5/15 s intervals.
5. **eBPF sampling rate does not reduce cost** — use Beyla at 100% sampling; limit instrumented pods rather than sampling rate to reduce overhead.
6. **Data-plane RTT impact is minimal** (<1 ms P99 across all configurations); monitoring cost is control-plane heavy.
7. **Detection efficiency per CPU millicore favours Beyla** ($E \approx 0.66$ vs. 0.069 at Prometheus 15 s), even with lower per-run consistency.

Recommendation: Deploy **both** stacks together for full fault-type coverage. Use Prometheus at a 5 s scrape interval as the application/infrastructure baseline. Add Beyla at **100%** sampling for inter-NF HTTP observability — marginal combined cost (~3.2 m CPU beyond Prometheus alone) while closing gaps on delay faults and subtle service-registry failures invisible to metrics.