A Cross-Layer Fault Atlas for Cloud-Native 5G Core Networks

Mapping fault classes to observability signals across infrastructure, orchestration, and application layers

Boyan Bonev

Supervisor(s): Dr. Nitinder Mohan, Sehan Samarakoon

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 17, 2026

Name of the student: Boyan Bonev (5950856)
Final project course: CSE3000 Research Project
Thesis committee: Dr. Nitinder Mohan, Sehan Samarakoon, Dr. Jérémie Decouchant

Abstract

5G core networks are moving from monolithic applications to containerised microservices on Kubernetes. This brings flexibility and scalability, but it also makes faults harder to detect, since a single fault can surface in the physical infrastructure, the Kubernetes orchestration, and the network functions at the same time. There is little empirical evidence on which telemetry signals reveal which kind of fault. This paper presents a fault atlas for a cloud-native 5G core: an empirical mapping from 22 injected faults in eight classes (resource stress, pod crashes, network degradation, attacks on the Packet Forwarding Control Protocol (PFCP), and dependency failures) to the 40 observability signals, collected from metrics (including user-plane round-trip time), logs, traces, and Kubernetes events, that detect them. On our setup, all but one of the 22 faults were detectable in more than one architecture layer, and within each fault class the layers reacted in a stable order. The orchestration layer, typically the most closely watched one, detected only 10 of the 22 faults and missed all CPU-stress, network-delay, and PFCP-attack faults, while 21 of the 22 faults were visible in more than one of the four telemetry modalities. The atlas is robust to the statistical methodology (at least 95.9% of signals unchanged under threshold and detector variations), and a second independent run agreed on 93.1% of its cells, with the differences confined to near-threshold signals. The atlas and the pipeline that generated it are released as a reusable ground truth for fault-detection and cross-layer diagnosis research.

1 Introduction

Mobile networks have become critical infrastructure: payment systems, emergency services, and everyday communication all assume that the network is available and responsive. At the centre of a 5G network sits the 5G core, which authenticates users, manages their sessions, and forwards their traffic. The 3GPP specifies the core as a Service-Based Architecture (SBA) in which the control plane is decomposed into network functions (NFs) that communicate over HTTP/2 APIs [1]. In practice, operators deploy these NFs as containerised microservices on Kubernetes. This improves scalability and deployment flexibility, but it also brings the operational complexity of large microservice systems into the mobile core [2].

That complexity changes how faults behave. In a monolithic core a fault stays inside one process, while in a cloud-native core it propagates. CPU starvation on one node slows down the NFs scheduled there, their peers see timeouts on the Service-Based Interface, retries amplify the load, and eventually user equipment (UE) registrations start failing. The same fault is now visible, in different forms, at three layers at once: the infrastructure (resource metrics), the Kubernetes orchestration (pod events and restarts), and the application (NF logs, traces, and 5G-specific counters). Studies of general microservice systems show that no single one of these views is enough on its own. Kubernetes health probes miss gradual degradations entirely [3], a multi-source fault detector loses half of its precision when traces are removed [4], and tracing helps to diagnose interaction faults but is ineffective for environment faults [5, 6].

Operators respond by collecting telemetry at every layer of the stack: infrastructure metrics, orchestration events, application logs, and traces of inter-NF calls. The difficult part is interpretation. When UE registrations begin failing, an engineer must decide which of the dozens of available signals distinguishes, say, a network partition from node-level memory pressure. Existing cloud-native 5G studies do not answer this question. They measure the user-level impact of resource stress [7], train classifiers on logs and Kubernetes events [8], or build diagnosis frameworks evaluated on a handful of fault scenarios [9], but none of them maps fault classes to the signals that reveal them. There is no empirical ground truth for which observability signal indicates which kind of fault in a cloud-native 5G core.

This paper provides such a map. We deployed an Open5GS core on Kubernetes with full-stack observability and a simulated radio access network, and used chaos engineering [10] to inject 22 faults from eight classes across the major network functions and their supporting infrastructure. For every fault we recorded 40 signals, drawn from metrics (including user-plane round-trip time), logs, eBPF-derived traces, and Kubernetes events, through baseline, fault, and recovery phases. From the resulting dataset we built a fault atlas: a mapping from each fault class to its impact on the different architecture layers (vertical) and across network functions (horizontal).

The atlas shows that all but one of the 22 faults manifest in more than one layer and that each fault class has a characteristic layer ordering. The orchestration layer turned out to be the weakest observation point. It detected only 10 of the 22 faults, staying silent for all CPU-stress and network-delay faults among others. These findings are robust to how we detect anomalies: they hold under variations of the detector's thresholds and statistical estimator, and a second independent run agreed on 93.1 % of the atlas, with the differences falling on near-threshold signals. They come from one Open5GS version on a single-node cluster, so we treat the structural results as specific to this testbed until they are replicated elsewhere.

The main research question is: **RQ5: How do different fault classes manifest across multiple layers in a cloud-native 5G core network?**, which is split into three sub-questions:

RQ5.1 How do faults of different classes manifest across the application, orchestration, and infrastructure layers in cloud-native systems?

RQ5.2 Which observability signals are most indicative of each fault class at different layers of the system?

RQ5.3 How can the resulting fault-signal pairs be organised into a useful categorisation?

The rest of the paper is organised as follows. Section 2 compares our work with the microservice fault analysis and 5G core literature. Section 3 describes the experimental setup, faults, and injection protocol. Section 4 shows the resulting fault atlas, the per-layer signal analysis, and a discussion of the findings. Section 5 explains the ethical and reproducibility aspects of the study, and Section 6 concludes and gives directions for future work.

2 Background and Related Work

5G core networks are transitioning to cloud-based systems. They consist of various network functions, including the Access and Mobility Management Function (AMF) and the User Plane Function (UPF), which are deployed as separate microservices in Kubernetes and communicate over the network. This is done to improve scalability, but it complicates the process of identifying and debugging faults, as shown by [2]. For example, resource depletion in one network function may result in increased latency for other NFs trying to call it. A faulty Network Repository Function (NRF) can disrupt service discovery in the entire cluster. These issues are visible in different ways at the three architectural levels: infrastructure (CPU and memory metrics), orchestration (pod events and restart counts), and application (NF logs, trace spans, native metrics). So, the detection and characterisation of these faults require cross-layer telemetry.

The microservice fault-analysis literature is very extensive and provides a good baseline for our study. Zhou et al. [5] surveyed 22 real-world fault cases across microservice systems with varying sizes and proposed a classification into Internal faults (bugs within a single service), Interaction faults (coordination failures in multiple services), and Environment faults (infrastructure and configuration errors). They show that trace-based debugging cuts fault-localisation time by 49% for Interaction faults but is ineffective for Environment faults, which do not benefit from trace analysis. This shows that no single observability modality is sufficient for capturing every fault class. Silva et al. [6] extend this view with a review of 117 faults across six non-functional requirement categories, which do not appear in monolithic fault taxonomies.

There are also empirical observability studies like ours that confirm the need for cross-layer monitoring. Flora et al. [3] show that Kubernetes liveness probes detect none of the gradual degradation faults they inject: they show a signal only for pod crashes. Fu et al. [4] measure the need for multiple modalities by ablating their multi-source detector: with metrics, distributed traces, and logs combined it reaches 97 % precision, but precision falls to 51 % without traces and to 85 % without metrics. They also show that trace span durations are the most informative feature for network-delay faults, while resource faults correlate with CPU and memory metrics.

The closest works in the 5G field use a similar experimental setup, but answer different questions. Moreira et al. [7] apply Chaos Mesh CPU and memory stress to each 5G core network function and measure the effect on UE registration latency, finding that the AMF has the biggest user impact. They collect only a single UE-level metric and do not characterise which observability signals at which layer are indicative of each fault. Hatami et al. [8] fine-tune a large language model on 118 fault-injection experiments covering five fault types, detecting faults from pod logs and Kubernetes state with 93% accuracy. PRISM [9] combines logs, metrics, and configuration state from the service, orchestration, and infrastructure layers of an Open5GS core into a temporal knowledge graph and extracts subgraphs that localise the components involved in a fault. A smaller body of work builds cloud-native 5G observability tooling rather than characterising faults: eBPF protocol-aware monitoring with Z-score anomaly detection [11] and white-box instrumentation for load-driven performance testing [12], both injecting at most a single fault.

These papers build various tooling: a diagnosis framework, a classifier, or a monitoring stack, validated on only a handful of fault scenarios. They do not characterise how fault classes differ in their cross-layer footprint, and PRISM in particular relies on logs, metrics, and configuration state without tracing, and evaluates only four faults. Our contribution is complementary to theirs - an empirical map, built from 22 experiments across eight fault classes and including eBPF-derived traces and end-to-end RTT, of which signals at which layer reveal each fault class. Such a map would be useful for evaluating a cross-layer diagnosis framework like PRISM.

3 Methodology and Experimental Setup

Our methodology is experiment-based:

1. Deploy a 5G core implementation in Kubernetes.
2. Simulate user traffic.
3. Inject one fault at a time.
4. Record telemetry from every observability layer.
5. Rebuild the cluster after each fault.

The resulting per-fault telemetry is transformed by a statistical detector into a binary fault atlas and time-based views of layer propagation. This section describes the experimental platform (Section 3.1), the fault model (Section 3.2), the collected observability signals (Section 3.3), the injection protocol (Section 3.4), and the anomaly detector and its thresholds (Section 3.5).

3.1 Experimental platform

The tech stack is listed in Table 1. We use Open5GS 2.7.5 [13], a standards-compliant open-source 5G core whose control-plane procedures and field behaviour have been independently validated against 3GPP specifications and commercial radio access networks [14–16]. The core has the following network functions: the Access and Mobility Management Function (AMF), Session Management Function (SMF), User Plane Function (UPF), Network Repository Function (NRF), Service Communication Proxy (SCP), Authentication Server Function (AUSF), Unified Data Management (UDM), Unified Data Repository (UDR), Policy Control Function (PCF), Network Slice Selection Function (NSSF), Security Edge Protection Proxy (SEPP), and Binding Support Function (BSF) and MongoDB as the subscriber

Table 1: Experimental platform.

Layer	Component
5G core	Open5GS 2.7.5 (12 NFs + MongoDB)
RAN simulation	UERANSIM (1 gNB, 10 UEs)
Orchestration	Kubernetes via kind
Fault injection	Chaos Mesh
Metrics	Prometheus [24] (cAdvisor, node-exporter, kube-state-metrics, Open5GS exporters)
Logs	Loki [25] + Promtail
Traces	Jaeger, fed by Grafana Beyla (eBPF)
Analysis	Python (pandas, NumPy, matplotlib)

database. Network functions communicate over the HTTP/2 Service-Based Interface (SBI) using indirect communication (3GPP model D) through the SCP [1]. The radio access network is simulated with UERANSIM [17] (**1 gNodeB (gNB) and 10 user equipments (UEs)**), which generates 3GPP-compliant NAS/NGAP signalling and has been validated against Open5GS at scale [18]. The cluster runs on kind [19] (Kubernetes [20] in Docker [21]). The Kubernetes deployment overhead for Open5GS is bounded at roughly 7 % relative to bare metal and does not affect within-experiment comparisons [22]. All experiments ran on a single workstation, an Intel Core i7-12700H (14 cores, 20 threads) with 32 GB of RAM, running Ubuntu 24.04 (Linux 6.14), Docker 29.3, and kind 0.27, so the entire core and observability stack share one physical node. Faults are injected with Chaos Mesh [23], a Kubernetes-native chaos-engineering controller [10].

Implementing tracing was the biggest technical difficulty: Open5GS uses HTTP/2 without the usual protocol negotiation, so sidecar proxies cannot recognise its traffic and fail to instrument it. We therefore capture SBI traffic with Grafana Beyla [26], an eBPF agent that reads HTTP/2 frames at the kernel socket layer and, since Open5GS does not expose trace context, records each call as an individual span exported to Jaeger [27], which we generalise per network function into error-rate and latency signals.

3.2 Fault model

We inject 22 faults from eight classes (Table 2). The classes were chosen to cover the resource, crash, network, and dependency failure modes from the existing microservice fault-taxonomy literature [5, 6], and to target the most impactful network functions according to prior studies (AMF, SMF, UPF, NRF, and SCP) [7, 28]. Each fault focuses on a specific network function so that we can separate the directly impacted function from propagating damage in the rest of the network. We also apply some faults on different network functions (such as CPU stress on AMF, SCP, AUSF, NRF) to inspect whether their impact depends on the target's role in the 5G core. We annotate each fault with its Zhou [5] class (Internal / Interaction / Environment) and the closest Silva [6] fault code, so our catalogue is in line with the existing taxonomies. Table 2 groups the faults by class; the per-fault injection parameters (target, exact Chaos Mesh mechanism and magnitude, duration, and effect) are listed in Appendix A (Table 5).

3.3 Observability signals

We collect 40 signals and assign each to one of the three layers of the architecture from Section 2. The infrastructure layer contributes 8: seven container- and node-level CPU, memory, and network-rate metrics, plus end-to-end UE round-trip time (RTT). The orchestration layer contributes 4: pod restart, ready, and running transitions, and Kubernetes Warning events. The remaining 28 are application-layer: five Beyla SBI latency, error, and request-rate metrics, twelve Open5GS state gauges, four Open5GS failure counters, four Loki log streams, Jaeger trace error rate and p95 latency, and an NRF registration snapshot. The complete dictionary, with the meaning and source of every individual signal, is given in Appendix B (Table 6).

3.4 Injection protocol

Each fault runs through the same baseline/fault/recovery phases with durations of 600s/300s/300s. To prevent cross-fault contamination we rebuild the entire cluster for every fault and gate the bring-up on strict readiness checks (Packet Forwarding Control Protocol (PFCP) peers of the UPF established, gNB NG-Setup complete and 10/10 UE PDU sessions active) before the baseline phase begins - a fault is injected if and only if the core is verifiably healthy. During each fault we simulate a fixed synthetic workload. All ten UEs continuously send user-plane traffic, by pinging the UPF data-network gateway at five packets per second. At the same time, three of the ten UEs generate control plane traffic: on a 70-second cycle the three deregister together, pause ten seconds, then re-register, driving the full NGAP, AMF, AUSF, UDM, NRF, and SCP signalling chain. The purpose of this workload is only to keep the core working so that faults produce an observable signal, not to reproduce a realistic production load, so no heavier traffic is required. We acknowledge that this control-plane load is light, which understimulates registration-path faults (AMF, AUSF, UDM). This is described in more detail in Section 4.

3.5 Anomaly detection

To go from raw telemetry to firing signals, we need an anomaly detector. We created a baseline-relative detector of the kind surveyed by Chandola et al. [29] and adapted to streaming operational telemetry [30, 31] - each signal is compared only to the pre-fault baseline from the same run. Our sliding-window approach follows prior 5G-core observability work: 5GC-Observer also flags degradation with a sliding-window Z-score over reported metrics [11], and PRISM scores anomalies with median/MAD statistics over a pre-incident reference window [9]. We instead combine a mean/σ test with the explicit floors introduced below, which directly target the idle signals in a 5G core.

For a continuous signal we aggregate samples into 5 s bins, matching the 5 s interval at which Prometheus exports the metrics, and slide a 30 s window. We intentionally lowered the interval below the default so that we can get higher resolution data for all of the fault phases. Let μ and σ be the mean and standard deviation of the pre-fault baseline and $\bar{x}_w$ the window mean. The window is anomalous when

Table 2: The 22 injected faults, grouped by class, with target network function, Chaos Mesh mechanism, and mapping to the Zhou [5] and Silva [6] microservice fault taxonomies.

Class	IDs	Target	Chaos Mesh mechanism	Zhou	Silva
CPU stress (4)	01, 08, 16, 18	AMF, SCP, AUSF, NRF	`StressChaos/cpu`	Environment	PF31 CPU Hog
Memory pressure (2)	02, 22	UPF, AMF	`StressChaos/mem`	Environment	PF26 Mem. Alloc
Pod crash (3)	03, 07, 19	AMF, SMF, UDM	`PodChaos/pod-kill`	Environment	PF03 Crash
Network delay (3)	04, 09, 17	AMF, NRF, SCP	`NetworkChaos/delay`	Interaction	PF15 Net. Delay
Network partition (2)	05, 21	AMF–SCP, AMF–gNB	`NetworkChaos/partition`	Interaction	PF22 Packet Loss
Packet loss (2)	06, 14	UPF	`NetworkChaos/loss`	Interaction	PF22 Packet Loss
PFCP attack (4)	10, 11, 12, 13	UPF, gNB	`NetworkChaos` (4 variants)	Interaction	PF22 Packet Loss
Dependency failure (2)	15, 20	NRF, MongoDB	`PodChaos/pod-kill`	Interaction	RF12 Svc. Unavail.

$$|\bar{x}_w - \mu| \;>\; \tau, \qquad \tau = \max\big(Z\,\sigma,\; \rho\,|\mu|,\; f\big), \tag{1}$$

with $Z = 3$ (the standard three-sigma test), a relative floor $\rho = 0.30$, and a per-signal absolute floor f. Signals that define a ratio guard $r > 1$ must additionally satisfy a multiplicative test, so that the window mean is a genuine multiple of the baseline rather than a large relative swing around a near-zero value:

$$|\bar{x}_w| \;\geq\; r\,|\mu| \qquad \text{or} \qquad |\bar{x}_w| \;\leq\; |\mu|/r. \tag{2}$$

A signal is reported as manifested only if both conditions hold for two consecutive windows, which suppresses single-bin spikes and prevents false alarms [30].

The two extra terms in Equation 1 are needed because a single global threshold is inappropriate and each signal needs its own. A pure three-sigma test would fail on the many 5G signals that are near-idle at baseline: when $\sigma \approx 0$, ordinary sampling jitter would be many standard deviations away and the detector fires constantly. The relative floor $\rho|\mu|$ and the absolute floor f form a guard against these flat baselines - they make it so a physically meaningful change is required instead of just a statistically large one. The ratio guard r of Equation 2 adds the same protection multiplicatively: a latency signal with $r = 1.5$, for example, must rise to at least $1.5\times$ its baseline before it counts, which removes near-zero bursty false positives. Because the 40 signals differ by orders of magnitude in unit, scale, and noise (byte rates, time latencies, session counts, log-line tallies), each carries its own f and, where appropriate, its own r. Non-continuous signals use purpose-built detectors of the same persistence form: failure counters require two consecutive non-zero samples above the pre-fault maximum, pod restarts require a step increase, ready/running gauges must drop to zero and Kubernetes events are matched on Warning reasons within the fault window.

These thresholds are hand-tuned per signal, which is the part of the pipeline most open to question. The complete per-signal floor f and ratio guard r, together with the fixed rules of the non-continuous detectors, are listed in Appendix F (Table 9). We test exactly the tuning that invites the question: Section 4.5 sweeps every constant and Section 4.6 replaces the whole mean/σ estimator with PRISM's median/MAD, and the atlas holds under both.

4 Results and Discussion

We present the results in three ways: the fault to signal matrix (Section 4.1, RQ5.1/RQ5.2), cross-layer timing (Section 4.2, RQ5.1), and network-function blast radius (Section 4.3, RQ5.1). We then categorise them (Section 4.4, RQ5.3), test their robustness to the detector and across runs (Sections 4.5–4.7), and list the threats to validity (Section 4.8). The findings are tied to one Open5GS version on a single-node cluster under a light workload, so we present the structural claims as testbed-specific until they are replicated elsewhere.

4.1 The fault atlas (RQ5.1, RQ5.2)

Figure 1 shows the full atlas: for each of the 22 faults (rows, grouped by class) it marks which of the 40 signals (columns, grouped by layer) fired. The exact per-fault onset, layer-propagation order, and blast radius underlying Figures 1–3 are described in Appendix C.

21 of the 22 faults manifest in more than one layer, which confirms that fault characterisation in a cloud-native 5G core is cross-layer (RQ5.1). Resource faults (CPU stress, memory pressure) raise infrastructure metrics first and then propagate into application-layer SBI latency and traces. Interaction faults (network delay, partition, packet loss, PFCP attacks) are mostly invisible to infrastructure counters, but show in traces, RTT, and logs. Crash and dependency faults are the only classes that trigger a signal in the orchestration layer.

As a concrete example, CPU stress on the SCP runs the full resource-fault path. CPU throttling on the SCP container fires in the first 5 s bin, raised SBI client latency and trace p95 then follow on the network functions that call through it, and the fault eventually shows on 12 of the 14 functions (Section 4.3), while the orchestration layer stays silent throughout. By contrast, network delay on the NRF runs the interaction-fault path. Trace p95 latency rises at t_0 on the NRF and the functions that depend on it for discovery, but no infrastructure counter and no Kubernetes signal moves at all, and only three signals manifest in total. It is the only fault in our catalogue that stays within a single layer.

The orchestration layer is blind to many of the faults. Kubernetes signals (restarts, ready/running drops, Warning events) fired for only 10 of the 22 faults: every pod crash, both OOM-driven memory-pressure faults, both partitions, and both dependency failures. They stayed completely silent for all four CPU stress faults, all three network-delay faults,

Table 3: Number of faults detectable by exactly k independent observability modalities (of four: metrics, logs, traces, Kubernetes events). Only one fault is single-modality, and the median is two.

Modalities k	1	2	3	4	total
# faults	1	12	5	4	22

and all four PFCP attacks. This directly confirms, in a 5G setting, Flora et al.'s finding that Kubernetes health probes detect none of the gradual-degradation faults they inject and react only to terminal crashes [3].

These two results, that almost all faults are cross-layer (21 of the 22 here) and that the orchestrator misses more than half of them, are the ones we lean on most, and neither is sensitive to how we detect anomalies: they hold under a sweep of the detector's constants (Section 4.5), under PRISM's median/MAD estimator instead of our mean/σ (Section 4.6), and in a second independent run (Section 4.7).

Table 3 counts, per fault, how many of the four modalities (metrics, logs, traces, Kubernetes events) independently detected it. Only one fault, memory pressure on the AMF, was detectable from a single modality (metrics), while the remaining 21 are visible on at least two. This shows, for 5G specifically, the modality complementarity that Fu et al. report for general microservices [4]: different fault types are best detected by different modalities, and their combination outperforms any single one.

4.2 Cross-layer timing (RQ5.1)

Figure 2 reports, per fault, the time after injection at which each layer first produced a signal. The first-manifesting layer is different by class: infrastructure is fastest for 11 faults (all resource faults and most UPF-side network faults, where RTT or CPU throttling react within the first 5s bin), orchestration is fastest for 6 (the crashes and dependency failures, where a Kubernetes Warning is emitted essentially at t_0), and the application layer is fastest for the remaining 5 (especially on the NRF and SCP network-delay faults, where trace p95 latency moves before any metric). The order in which the layers react is therefore a discriminator on its own: an infrastructure-then-application sequence signals a resource fault, an orchestration-first sequence a crash, and an application-only sequence an interaction fault that never reaches the infrastructure counters.

4.3 Network-function blast radius (RQ5.1)

Figure 3 shows, for every fault, which network functions emitted a signal and how fast, with the chaos target in red. Blast radius ranges from a single function (AMF memory pressure stays local to the AMF) to almost the whole network, for example stressing the SCP or injecting delay at the SCP affects 12–13 of the 14 functions, because in 3GPP model D every inter-NF call traverses the SCP. The NRF behaves similarly, as its failure cascades to all discovery-dependent functions, matching the shared-dependency importance noted in the taxonomy and 5G literature [5, 28]. Across the table the AMF is the most frequently affected function (it shows a signal under all 22 faults, usually as collateral rather than as the target), which makes it a sensitive but non-specific place to watch. This empirically confirms PRISM's design choice of the AMF as the anchor for cross-layer validation [9].

Table 4: Per-class cross-layer signature: typical first-detected layer and the most distinguishing signals.

Class	**First layer**	**Distinguishing signals**
CPU stress	Infra	CPU usage/throttle, SBI latency, trace p95
Memory pressure	Infra/App	pod restart, working set, N4 estab.
Pod crash	Orch	K8s Warning, ready-drop, AMF gauges
Network delay	App	SBI client latency, trace p95/error
Network partition	App	UE-failure logs, AMF gauges, restart
Packet loss	Infra/Orch	RTT, PFCP peers, PDU success
PFCP attack	Infra	RTT, mem. working set, trace p95/error
Dependency failure	Orch	K8s Warning, NRF/SCP logs, AMF reg-fail

4.4 Categorisation (RQ5.3)

The first two views answer where a fault shows up and when. RQ5.3 asks how to organise the fault–signal pairs usefully. We do so in two ways: the first is the per-class cross-layer signature summarised in Table 4: each fault class maps to a typical first-detected layer and a small set of distinguishing signals, those that fire for that class but for few other classes. The second way is signal specificity, measured as $1 - H/H_{\max}$, where H is the entropy of the fault-class distribution among the faults a signal fires for (the full ranking is in Appendix D, Table 8). This cleanly separates broad alarms from localisers: trace p95 latency and trace error rate fire for 18–19 of 22 faults (specificity 0.07–0.11, excellent detectors that something is wrong but useless for attribution), while signals such as the AMF registration-failure counter, the AMF authentication-failure counter, and the pod `Running`-drop fire for a single class (specificity 1.0, rare but unambiguous). A practical detection stack therefore needs both: a high-recall trace/RTT layer to notice the fault and a high-specificity counter/event layer to identify it.

Further: within the 300 s recovery phase only 5 of 22 faults returned every signal to baseline. Crash faults recovered their orchestration and trace signals quickly but session-state gauges often did not, and the MongoDB kill left eight signals still triggered, so fast orchestration recovery does not imply application-level recovery.

4.5 Robustness to the detector's constants

Because the detector's constants are hand-tuned (Section 3.5), we ran a one-factor-at-a-time sweep around the baseline ($Z = 3$, persistence $= 2$, window $= 30$ s, floor, ratio, and relative guards $\times 1$), changing each in turn ($Z \in \{2.5, 3.5\}$, persistence $\in \{1, 3\}$, window $\in \{15, 45\}$ s, and the floor, ratio, and relative ($30\,\%$) guards $\times 0.5$ and $\times 2$) and recomputing the en-

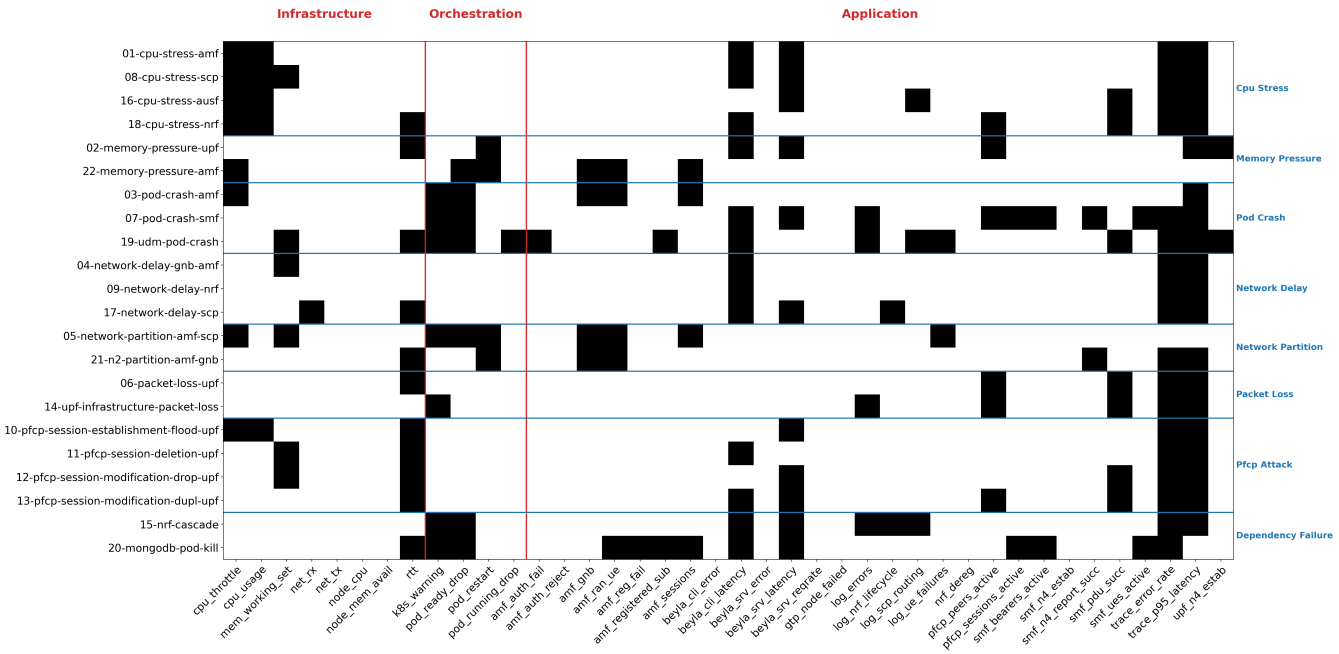

Figure 1: The fault atlas: which of the 40 observability signals (columns, grouped by layer) each fault (rows, grouped by class) manifested (black). Vertical red lines separate the three layers, and horizontal blue lines separate fault classes.

tire atlas. Across all fourteen configurations the atlas agreed with the baseline on at least **97.0 %** of its 22×40 cells, and the first-manifesting layer was unchanged for at least 20 of 22 faults (for most configurations, all 22). Disabling each guard in turn isolates what it does: removing the absolute floor inflates the atlas from 162 to 237 manifested cells ($+75$, almost all on noisy infrastructure and application metrics), so the floor is the main false-positive control. Removing the ratio guard changes a single cell, so it serves only as a backstop. Two-thirds of the continuous-signal detections clear the 3σ threshold, so the black cells show genuine signals. What remains constant is that almost every fault stays cross-layer, the orchestrator misses more than half of them, no single modality suffices, and the per-class layer firing order is unchanged. The exact constants set the detector's operating point without determining the atlas's structure. A stronger test is to change the statistical approach itself, which we do next.

4.6 Comparison with PRISM's detector

PRISM scores anomalies with a median/MAD estimator rather than our mean/σ [9]. To test whether the atlas is an artefact of that choice, and to relate our characterisation to PRISM's detection machinery on identical data, we recomputed the entire atlas with median and $1.4826 \times$MAD in place of mean and σ, holding every other constant fixed. The two estimators agree on **95.9 %** of the 22×40 cells (844 of 880).

The disagreement is small and concentrated. Every one of the 36 differing cells is one that median/MAD gains: it never drops a manifestation, only adds borderline ones. This is expected: a robust scale estimator is not inflated by baseline jitter, so the threshold sits lower and weak signals cross. It makes our mean/σ detector the more conservative of the two. The gained cells split as 24 in the application layer and 12 in infrastructure, almost all on drift-prone or bursty signals (the memory working set and SBI/trace latencies), exactly those whose σ is most sensitive to outliers. Not one of the 36 lies in the orchestration layer, so the blind-spot result that the orchestrator detects only 10 of 22 faults is identical under both detectors. The first-manifesting layer is unchanged for 18 of 22 faults, and the four shifts all move to an earlier infrastructure signal. So, the atlas's structure reflects the system rather than the statistical approach.

4.7 Reproducibility across runs

The sweep in Section 4.5 changed the detector's constants and Section 4.6 changed its estimator, both on the same recorded data. A stronger test is to regenerate the data itself. We ran the full 22-fault batch a second time, with a fresh cluster rebuilt before each fault, and detected anomalies again with the identical detector. This second run reproduces the published atlas on **93.1 %** of the 22×40 cells (819 of 880), with a Cohen's κ of 0.768, which is substantial agreement once chance is removed. The 61 cells that differ are balanced between the two runs (32 fire only in the original run, 29 only in the second), so neither run detects systematically more, and they sit on weak signals close to the detection threshold (Appendix E, Figure 4). Among the cells that fire in either run, agreement is lower, with a Jaccard overlap of $68.1\,\%$, which is the same near-threshold volatility that the single-run caveat below describes. The block structure of the atlas, the orchestration blind spot, and the per-class layer ordering are all unchanged, so the atlas reproduces as a structure even where a few borderline cells move. The cross-layer property itself shifts by only one fault: 21 of the 22 faults

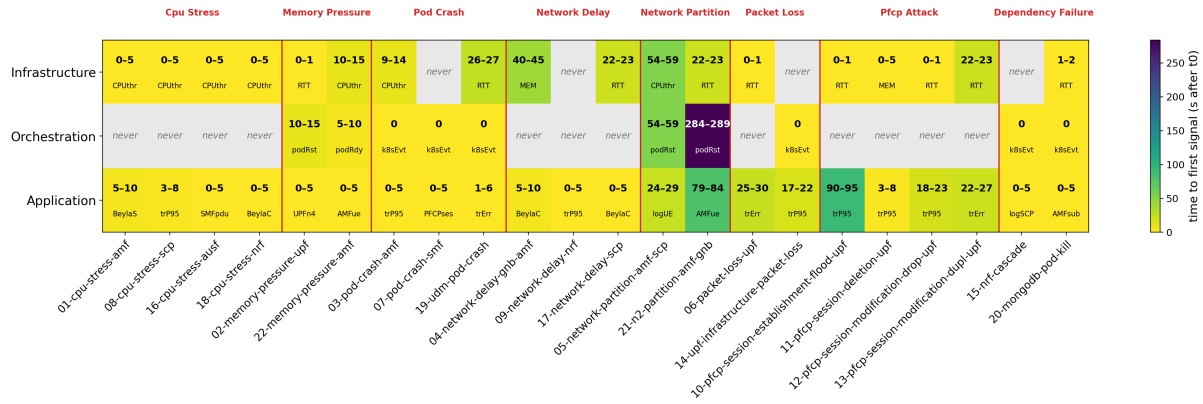

Figure 2: Time to first signal per layer (seconds after t_0) for each fault. Each cell gives the onset bucket and first signal, and "never" marks a layer with no signal. Resolution is 5 s for Prometheus/Loki/Jaeger/pod gauges, 1 s for RTT, and sub-second for Kubernetes events.

are cross-layer in the canonical run and 20 in the rerun, the single difference being a borderline PFCP modification fault whose lone infrastructure signal (RTT) drops below threshold the second time. The per-layer timing is more stable than the individual cells: the first-manifesting layer agrees for 18 of the 22 faults across the two runs, and the cell-by-cell onset differences are given as a timing companion to the cross-run atlas in Appendix E (Figure 5). Because this rests on a single rerun, we read it as evidence that the structure of the atlas is stable, not as a confidence on any individual cell. The cells that move are the near-threshold application signals, which is where repeated runs would matter most.

4.8 Threats to validity

Several limitations bound our claims. First, the atlas's headline is computed from one clean run per fault, and a signal that fires or not near a threshold could flip between runs. A second independent run reproduces 93.1 % of the atlas (Section 4.7), but attaching per-cell statistical confidence still needs more repetitions. The pipeline is deterministic and re-runnable for exactly this purpose (Section 5). Another potential issue is that only three of ten UEs cycle registration, which understimulates the AMF/AUSF/UDM registration path and likely understates the control-plane faults. A heavier signalling workload is a good next experiment.

Our tracing is also span-level rather than fully distributed: because Open5GS does not propagate trace context, Beyla records each SBI call as an individual span instead of stitching a request into an end-to-end trace, so the trace signals capture per-service latency and error rates but not causal request paths across network functions.

Results are for Open5GS 2.7.5 on kind with eBPF tracing. Absolute onset times and a handful of unexported counters are implementation-specific, though the cross-layer structure should transfer, and the testbed's external validity is supported by independent benchmarks against 3GPP procedures and commercial RAN [14, 15]. These limitations affect individual cells and absolute onset times rather than the structure of the atlas: the cross-layer manifestation, the orchestration blind spot, and the per-class layer ordering hold across every robustness check in Sections 4.5–4.7.

5 Responsible Research

Reproducibility and artefacts. The study is designed to be reproduced. The entire experiment is scripted: one command (`run_all.sh`, which invokes `cluster-start.sh` for each fault) recreates the Kubernetes cluster, deploys Open5GS, UERANSIM, and the observability stack, enforces the readiness gates of Section 3.4, and runs the 22-fault batch to produce per-phase CSV telemetry. All software used is open source and pinned to specific versions (Open5GS 2.7.5 via Gradiant chart 2.3.4, UERANSIM, Chaos Mesh, Prometheus, Loki, Jaeger, Grafana Beyla), and the fault definitions, collectors, detector, and analysis code are released in a public repository, developed in collaboration with the research project group [32]. The raw per-phase telemetry for every run is archived on 4TU.ResearchData [33]. The detector is deterministic given the raw telemetry, so the atlas can be recomputed exactly. We additionally report a sensitivity sweep (Section 4.5) so that readers can see how the results would change under different detector settings rather than having to trust a single configuration.

Integrity and honest reporting. We report the limitations that bear on the conclusions rather than only the favourable results. In particular we state plainly that the published atlas rests mainly on one clean run per fault, which a second independent run reproduces on 93.1 % of cells ($\kappa = 0.768$) as a check on the structure rather than a per-cell confidence, that the control-plane workload is light and under-stimulates registration-path faults, and that 10 of the 40 signals, including expected ones such as the GTP packet counters and the AMF registration-time histogram, never fired, either because Open5GS 2.7.5 does not export them or because the injected faults did not cause a change in them. These disclosures are made so that the atlas is understood for what it is: a blank cell means "not observed," not "proven absent".

Ethical considerations. The research raises no human-subjects or privacy concerns. All traffic is synthetic: simulated UEs generated by UERANSIM register and exchange

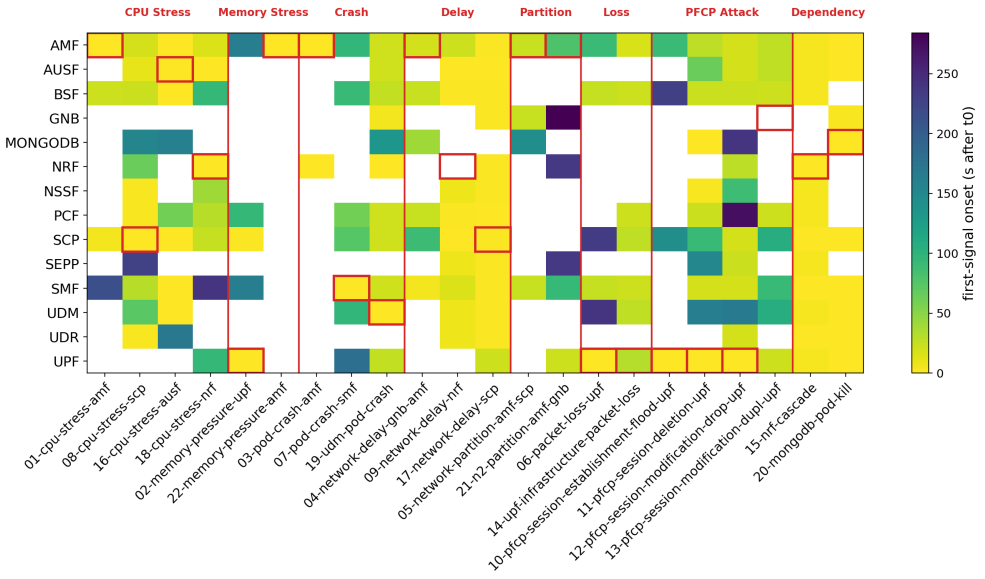

Figure 3: Network-function blast radius: first-signal onset (seconds after t_0, colour) for every function under every fault. White marks an unaffected function and a red box marks the chaos target.

data with a self-hosted core running in an isolated local cluster. No real subscriber data, personal data, or production network is involved, and no external system is targeted. The fault-injection techniques are standard chaos-engineering practice [10] applied only to our own testbed. While some faults are framed as "attacks" (the PFCP cases), they are used only to characterise observability and carry no exploit content beyond what is already public in Chaos Mesh.

Use of AI tools. Large-language-model assistance (Anthropic Claude) was used as a coding and writing aid - for drafting analysis scripts, helping with testbed setup, and editing prose. It was not used to generate or alter experimental data. All experimental results, every quantitative claim, and all cited literature were verified by the author against original sources. The author takes full responsibility for the content.

6 Conclusions and Future Work

We asked how different fault classes manifest across the application, orchestration, and infrastructure layers of a cloud-native 5G core, and which observability signals best reveal each. To answer it, we deployed an Open5GS core on Kubernetes with full-stack observability, injected 22 controlled faults across eight classes, and reduced the resulting metrics, logs, traces, and orchestration events into a fault atlas with a single baseline-relative anomaly detector.

On RQ5.1, all but one of the 22 faults proved cross-layer, and the classes separate by which layer they reach first and how they propagate: resource faults appear first in infrastructure metrics and propagate into traces, crashes and dependency failures announce themselves at the orchestration layer, and interaction faults surface only in application traces, RTT, and logs. On RQ5.2, no single modality suffices: 21 of 22 faults required at least two of four modalities, and the orchestration layer, the one most operators watch, detected only 10 of 22, missing every CPU-stress, network-delay, and PFCP-attack fault, reproducing Flora et al.'s Kubernetes blind-spot result in the 5G setting. On RQ5.3, we organised the fault-signal pairs by per-class cross-layer signature and by signal specificity, distinguishing high-recall detectors (trace latency, RTT) from rare but unambiguous localisers (failure counters, pod-running drops). A sensitivity sweep showed these conclusions hold across every perturbation of the detector's constants ($\geq 97\,\%$ atlas agreement) and under PRISM's median/MAD estimator (95.9 %), so they characterise the system itself rather than our choice of detector.

Several gaps remain. A second independent run reproduces the atlas on 93.1 % of its cells ($\kappa = 0.768$). This shows the structure is stable but does not assign a confidence to any single cell, which one rerun cannot do. The clearest next step is to repeat the full batch ten or more times and report, for each cell, the fraction of runs in which it fires, turning the binary atlas into per-cell frequencies with confidence intervals, which the deterministic pipeline is built for. A heavier control-plane workload is also needed to properly stimulate registration-path faults that the current light signalling load under-drives. Beyond that, the atlas is a labelled, per-layer ground truth that future work can build on: it can seed and benchmark automated fault classifiers and cross-layer diagnosis frameworks such as PRISM [9], be extended to severity rather than binary manifestation, and be replicated on other cores (free5GC, OpenAirInterface) to test how far the cross-layer structure generalises. We deliberately stop at characterisation: automated mitigation, such as recovery actions triggered by the distinguishing signals of each fault class, is a natural next step that the atlas makes possible. We hope the atlas, and the reproducible pipeline behind it, give both operators and researchers a concrete map from 5G fault classes to the signals that reveal them.

References

[1] 3GPP, "System architecture for the 5G system (5GS); stage 2," Technical Specification (TS) 23.501, 3rd Generation Partnership Project (3GPP), 2023. Version 18.3.0.

[2] J. Soldani, D. A. Tamburri, and W.-J. V. D. Heuvel, "The pains and gains of microservices: A systematic grey literature review," *Journal of Systems and Software*, vol. 146, pp. 215–232, 2018.

[3] J. Flora, P. Gonçalves, M. Teixeira, and N. Antunes, "A study on the aging and fault tolerance of microservices in Kubernetes," *IEEE Access*, vol. 10, pp. 132786–132799, 2022.

[4] N. Fu, G. Cheng, G. Dai, H. Mei, X. Qiu, and Y. Teng, "A failure analysis framework to provide pure anomalous data using multi-source data of fault-sensitive microservices," *Journal of Systems and Software*, vol. 230, p. 112513, 2025.

[5] X. Zhou, X. Peng, T. Xie, J. Sun, C. Ji, W. Li, and D. Ding, "Fault analysis and debugging of microservice systems: Industrial survey, benchmark system, and empirical study," *IEEE Transactions on Software Engineering*, vol. 47, no. 2, pp. 243–260, 2021.

[6] F. G. S. Filho, V. Lelli, I. de Sousa Santos, and R. M. C. Andrade, "Towards a fault taxonomy for microservices-based applications," in *Proceedings of the XXXVI Brazilian Symposium on Software Engineering (SBES)*, pp. 247–256, 2022.

[7] R. Moreira, L. F. R. Moreira, and F. de Oliveira Silva, "Performance evaluation and threat mitigation in large-scale 5G core deployment," 2025.

[8] P. Hatami, A. Majlesara, A. Majlesi, and B. H. Khalaj, "Automated fault detection in 5G core networks using large language models," 2025.

[9] S. Samarakoon, N. Mohan, and F. Kuipers, "PRISM: Cross-layer observability framework for mobile core networks," in *Proceedings of the IFIP Networking Conference (IFIP Networking)*, 2026.

[10] A. Basiri, N. Behnam, R. de Rooij, L. Hochstein, L. Kosewski, J. Reynolds, and C. Rosenthal, "Chaos engineering," *IEEE Software*, vol. 33, no. 3, pp. 35–41, 2016.

[11] A. Khichane, I. Fajjari, N. Aitsaadi, and M. Gueroui, "5GC-Observer: A non-intrusive observability framework for cloud-native 5G system," in *Proceedings of the IEEE/IFIP Network Operations and Management Symposium (NOMS)*, 2023.

[12] C. E. Menin, I. M. Silva, V. B. Alves, G. Lando, C. B. Both, J. M. S. Nogueira, and J. A. Wickboldt, "Enabling cloud-native observability for white-box performance analysis of the 5G core," in *Proceedings of the IEEE/IFIP Network Operations and Management Symposium (NOMS)*, 2025.

[13] Open5GS, "Open5GS: Open source implementation of 5G core and EPC." https://open5gs.org/, 2024. Accessed: 2026-05-31.

[14] B. Mukute, C. Deysel, F. Malan, and H. Kriel, "Control plane performance benchmarking and feature analysis of popular open-source 5G core networks: OpenAirInterface, Open5GS, and free5GC," *IEEE Access*, vol. 12, 2024.

[15] G. Zu, J. O. Boateng, V. S. Advani, T. U. Islam, V. Lee, S. Babu, M. Nadim, D. Qiao, M. Y. Selim, and H. Zhang, "Real-world integration and evaluation of open-source 5G core with commercial RAN," in *Proceedings of the IEEE Military Communications Conference (MILCOM)*, 2025.

[16] M. Barbosa, M. Silva, E. Cavalcanti, and K. Dias, "Open-source 5G core platforms: A low-cost solution and performance evaluation," in *Proceedings of the International Conference on Information Networking (ICOIN)*, pp. 99–104, 2025. arXiv:2412.21162.

[17] A. Güngör, "UERANSIM: Open-source state-of-the-art 5G UE and RAN (gNodeB) implementation." https://github.com/aligungr/UERANSIM, 2023. Accessed: 2026-05-31.

[18] S. S. M. S, M. R. Kanagarathinam, and K. M. Sivalingam, "Performance evaluation of 5G core network control-plane using Open5GS and Kubernetes," in *2025 17th International Conference on COMmunication Systems and NETworks (COMSNETS)*, pp. 584–592, 2025.

[19] The kind Authors, "kind: Kubernetes IN Docker." https://kind.sigs.k8s.io, 2024. v0.27.0, accessed 2026-06-02.

[20] The Kubernetes Authors, "Kubernetes: Production-grade container orchestration." https://kubernetes.io, 2024. Accessed: 2026-06-02.

[21] Docker, Inc., "Docker: Accelerated container application development." https://www.docker.com, 2024. Accessed: 2026-06-02.

[22] C. Deysel, F. Malan, *et al.*, "A brief performance comparison of bare-metal and Kubernetes deployments for 5G core control plane network functions using Open5GS," tech. rep., CSIR, 2024.

[23] The Chaos Mesh Authors, "Chaos Mesh: A chaos engineering platform for Kubernetes." https://chaos-mesh.org, 2024. Accessed: 2026-06-02.

[24] The Prometheus Authors, "Prometheus: Monitoring system and time series database." https://prometheus.io, 2024. Accessed: 2026-06-02.

[25] Grafana Labs, "Grafana Loki: Horizontally scalable log aggregation system." https://grafana.com/oss/loki/, 2024. Accessed: 2026-06-02.

[26] Grafana Labs, "Grafana Beyla: ebpf-based application auto-instrumentation." https://grafana.com/oss/beyla-ebpf/, 2024. Accessed: 2026-06-02.

[27] The Jaeger Authors, "Jaeger: Open source, distributed tracing platform." https://www.jaegertracing.io, 2024. Accessed: 2026-06-02.

[28] A. Paul, S. Keluskar, and M. Vutukuru, "A scalable and fault-tolerant 5G core on Kubernetes," in *Proceedings of the 17th International Conference on COMmunication Systems & NETworkS (COMSNETS)*, 2025.

[29] V. Chandola, A. Banerjee, and V. Kumar, "Anomaly detection: A survey," *ACM Computing Surveys*, vol. 41, no. 3, pp. 15:1–15:58, 2009.

[30] S. Ahmad, A. Lavin, S. Purdy, and Z. Agha, "Unsupervised real-time anomaly detection for streaming data," *Neurocomputing*, vol. 262, pp. 134–147, 2017.

[31] P. Notaro, J. Cardoso, and M. Gerndt, "A survey of AIOps methods for failure management," *ACM Transactions on Intelligent Systems and Technology*, vol. 12, no. 6, pp. 81:1–81:45, 2021.

[32] B. Bonev, S. Kutsarov, Y. Mihaylova, D. Ghergut, and V. Ilchev, "Cloud-native 5G core fault-injection experiments." https://github.com/bonevboyan/cse3000-5g-core-fault-experiments, 2026.

[33] B. Bonev, "Data supporting the research of full-stack observability under controlled fault injection in a cloud-native 5G core network." Version 1. 4TU.ResearchData. dataset. https://doi.org/10.4121/e62eef6e-4958-4713-9cef-ed5b317e40a3.v1, 2026.

A Fault dictionary

Table 5 expands the class-level summary of Table 2 into the full per-fault injection specification: the target network function, the exact Chaos Mesh mechanism and its magnitude, the injection duration, and a one-line description of the intended effect. All values are taken directly from the Chaos Mesh manifests used in the experiment. Faults whose name spans two endpoints (e.g. AMF↔SCP) act on traffic between those two functions; PFCP-attack faults combine more than one Chaos Mesh object, listed together in the mechanism column.

Table 5: The 22 injected faults: target, Chaos Mesh mechanism and magnitude, duration, and effect. Grouped by the eight classes of Table 2.

ID	Target	Mechanism & magnitude	Dur.	Effect
CPU stress				
01	AMF	`StressChaos/cpu`: 2 workers @ 80 %	5 m	Saturates AMF CPU, throttling registration and N2 signalling
08	SCP	`StressChaos/cpu`: 2 workers @ 80 %	5 m	Saturates SCP CPU, slowing SBI message routing between NFs
16	AUSF	`StressChaos/cpu`: 2 workers @ 80 %	5 m	Saturates AUSF CPU, slowing UE authentication
18	NRF	`StressChaos/cpu`: 2 workers @ 80 %	5 m	Saturates NRF CPU, slowing NF discovery and heartbeats
Memory pressure				
02	UPF	`StressChaos/mem`: 4 workers, 2048 MiB	5 m	Allocates ∼60× UPF RSS, forcing an OOM kill and pod restart
22	AMF	`StressChaos/mem`: 2 workers, 192 MiB	5 m	Applies sub-OOM memory pressure to the AMF
Pod crash				
03	AMF	`PodChaos/pod-kill`	1 m	Kills the AMF pod once at t_0; Kubernetes reschedules it
07	SMF	`PodChaos/pod-kill`	5 m	Kills the SMF pod once at t_0
19	UDM	`PodChaos/pod-kill`	5 m	Kills the UDM pod once at t_0
Network delay				
04	AMF→SCP	`NetworkChaos/delay`: 500 ms, 20 ms jitter, corr. 25	5 m	Delays AMF egress to the SCP by 500 ms
09	NRF	`NetworkChaos/delay`: 500 ms, 20 ms jitter, corr. 25	5 m	Delays all NRF egress by 500 ms
17	SCP	`NetworkChaos/delay`: 500 ms, 20 ms jitter, corr. 25	5 m	Delays all SCP egress by 500 ms
Network partition				
05	AMF↔SCP	`NetworkChaos/partition`: both directions	5 m	Fully severs AMF–SCP connectivity
21	AMF↔gNB	`NetworkChaos/partition`: both directions	5 m	Severs the N2 link between AMF and gNB
Packet loss				
06	UPF	`NetworkChaos/loss`: 30 %, corr. 25	5 m	Drops 30 % of UPF packets
14	UPF (N4, N6)	`NetworkChaos/loss`: 30 %, corr. 25, on N4↔SMF and N6↔DN	5 m	Drops 30 % on the UPF N4 and N6 data paths
PFCP attack				
10	UPF (N4)	`NetworkChaos/bandwidth`: 900 Mbps inbound from SMF + `StressChaos/cpu`: 4 workers @ 85 %	5 m	Session-establishment flood: floods the UPF N4 interface and saturates its CPU
11	UPF, gNB	`NetworkChaos/loss`: 100 % on N4 (UPF→SMF) and N3 (→UPF)	5 m	Session deletion: drops all PFCP on N4 and all GTP-U on N3
12	UPF, gNB	`NetworkChaos/corrupt`: 40 %, corr. 60 on inbound PFCP + `loss`: 100 % on N3 GTP-U	3 m	Session-modification drop: corrupts PFCP modification requests and severs the N3 downlink
13	gNB (N3)	`NetworkChaos/duplicate`: 100 %, corr. 25 on inbound GTP-U	5 m	Session-modification duplicate: forks every downlink GTP-U packet
Dependency failure				
15	NRF	`PodChaos/pod-kill`	3 m	Kills the NRF pod, the discovery dependency for all NFs
20	MongoDB	`PodChaos/pod-kill`	5 m	Kills MongoDB, the subscriber-data store behind UDR/UDM

B Signal dictionary

Table 6 describes each of the 40 collected signals summarised in Section 3.3: its layer, the component that produces it, and what it measures. These are the column names of the atlas in Figure 1. All metric signals are scraped and stored by Prometheus, so the source column names the exporter that produces the measurement, not the database it is queried from. The RTT signal does not pass through Prometheus and is read directly from the UE ping output.

Table 6: The 40 observability signals: layer, source, and meaning.

Signal	Source	What it measures
Infrastructure		
`cpu_usage`	cAdvisor	Per-container CPU usage rate, per NF
`cpu_throttle`	cAdvisor	Rate at which a container is CPU-throttled by the kernel scheduler
`mem_working_set`	cAdvisor	Per-container memory working set (bytes)
`net_rx`	cAdvisor	Per-pod network receive byte rate
`net_tx`	cAdvisor	Per-pod network transmit byte rate
`node_cpu`	node-exporter	Whole-node CPU usage
`node_mem_avail`	node-exporter	Node available memory
`rtt`	UE ping probe	End-to-end user-plane round-trip time: each UE pings the data-network gateway through the GTP tunnel via the UPF, sampled at 1 Hz
Orchestration		
`pod_restart`	kube-state-metrics	Cumulative container restart count per pod
`pod_ready_drop`	kube-state-metrics	Pod `Ready` condition gauge (fires when it drops to 0)
`pod_running_drop`	kube-state-metrics	Pod `Running` phase gauge (fires when it drops to 0)
`k8s_warning`	Kubernetes API	Events of type Warning emitted for core pods (e.g. `BackOff`, `Unhealthy`, `Killing`)
Application: Beyla SBI metrics (eBPF)		
`beyla_srv_latency`	Beyla	Server-side duration of SBI HTTP/2 requests handled by each NF
`beyla_cli_latency`	Beyla	Client-side duration of outgoing SBI requests per NF
`beyla_srv_error`	Beyla	Rate of SBI server responses with an error status code
`beyla_cli_error`	Beyla	Rate of outgoing SBI requests that end in an error
`beyla_srv_reqrate`	Beyla	Rate of incoming SBI requests per NF
Application: Open5GS state gauges		
`amf_registered_sub`	AMF exporter	Subscribers currently registered at the AMF
`amf_ran_ue`	AMF exporter	UE contexts the AMF holds over NGAP
`amf_gnb`	AMF exporter	gNBs with an active NG connection to the AMF
`amf_sessions`	AMF exporter	Active sessions tracked by the AMF
`pfcp_sessions_active`	SMF exporter	Active PFCP (N4) sessions between SMF and UPF
`pfcp_peers_active`	SMF exporter	Established PFCP peer associations
`smf_ues_active`	SMF exporter	UEs with an active context at the SMF
`smf_bearers_active`	SMF exporter	Active bearers at the SMF
`smf_pdu_succ`	SMF exporter	Rate of successful PDU-session establishments
`smf_n4_estab`	SMF exporter	Rate of N4 session establishments at the SMF
`smf_n4_report_succ`	SMF exporter	Rate of successful N4 session reports
`upf_n4_estab`	UPF exporter	Rate of N4 session establishments at the UPF
Application: Open5GS failure counters		
`amf_reg_fail`	AMF exporter	Failed initial UE registrations
`amf_auth_fail`	AMF exporter	Failed UE authentications
`amf_auth_reject`	AMF exporter	Authentication rejects sent by the AMF
`gtp_node_failed`	SMF exporter	GTP node failures (never fired: not exported by Open5GS 2.7.5)
Application: Loki log streams		
`log_errors`	Loki	Log lines across all NFs matching generic error patterns (error, exception, refused, failed, fatal, OOM)
`log_ue_failures`	Loki	Lines indicating UE procedure failures (registration reject, payload not forwarded, SBI receive errors)
`log_scp_routing`	Loki	SCP routing failures (connection refused, connection timer expired, failed connects to peer NFs)
`log_nrf_lifecycle`	Loki	NRF lifecycle lines (NF registered/de-registered, heartbeats, registration retries)
Application: Jaeger traces and NRF		
`trace_error_rate`	Jaeger	Fraction of spans with an error status, per service
`trace_p95_latency`	Jaeger	95th-percentile span duration, per service
`nrf_dereg`	NRF	Change in the set of NF registrations at the NRF between the pre-fault and fault phases (snapshot diff)

C Per-fault atlas details

Table 7 gives the full per-fault breakdown underlying Figures 1–3: the chaos target, the first-detected layer and signal, the time to first detection, the layer propagation order, the blast radius (number of network functions showing a signal), whether the orchestration layer detected the fault, and the total number of signals that manifested.

Table 7: Complete per-fault atlas. First layer / signal is the earliest detection, and t_0 is the injection time. Blast radius is the number of affected network functions (of 14).

Fault	Target	First layer / signal	t (s)	Layer order	Blast	K8s	#sig
01 cpu-stress-amf	amf	infra / cpu-throttle	0	infra→app	4	no	6
08 cpu-stress-scp	scp	infra / cpu-throttle	0	infra→app	12	no	7
16 cpu-stress-ausf	ausf	infra / cpu-throttle	0	infra→app	9	no	7
18 cpu-stress-nrf	nrf	infra / cpu-throttle	0	infra→app	9	no	8
02 memory-pressure-upf	upf	infra / rtt	0	infra→app→orch	5	yes	7
22 memory-pressure-amf	amf	app / amf-ran-ue	0	app→orch→infra	1	yes	6
03 pod-crash-amf	amf	orch / k8s-warning	0	orch→app→infra	2	yes	7
07 pod-crash-smf	smf	orch / k8s-warning	0	orch→app	7	yes	12
19 udm-pod-crash	udm	orch / k8s-warning	0	orch→app→infra	11	yes	15
04 network-delay-gnb-amf	amf	app / beyla-cli-latency	5	app→infra	6	no	4
09 network-delay-nrf	nrf	app / trace-p95	0	app	10	no	3
17 network-delay-scp	scp	app / beyla-cli-latency	0	app→infra	13	no	7
05 partition-amf-scp	amf	app / log-ue-failures	24	app→infra→orch	4	yes	9
21 n2-partition-amf-gnb	amf	infra / rtt	22	infra→app→orch	6	yes	7
06 packet-loss-upf	upf	infra / rtt	0	infra→app	6	no	5
14 upf-packet-loss	upf	orch / k8s-warning	0	orch→app	7	yes	6
10 pfcp-estab-flood-upf	upf	infra / rtt	0	infra→app	4	no	6
11 pfcp-deletion-upf	upf	infra / mem-working-set	0	infra→app	11	no	5
12 pfcp-mod-drop-upf	upf	infra / rtt	0	infra→app	13	no	6
13 pfcp-mod-dupl-upf	gnb	infra / rtt	22	infra→app	8	no	7
15 nrf-cascade	nrf	orch / k8s-warning	0	orch→app	12	yes	9
20 mongodb-pod-kill	mongodb	orch / k8s-warning	0	orch→app→infra	9	yes	13

Signals that never manifested. Ten of the 40 candidate signals fired for no fault and are retained only for completeness. `net_tx`, `node_cpu`, and `node_mem_avail` are infrastructure metrics too coarse at single-node scale, and the Beyla error and request-rate signals (`beyla_srv_error`, `beyla_cli_error`, `beyla_srv_reqrate`) likewise never crossed their detection thresholds. `smf_n4_estab`, `amf_auth_reject`, and `gtp_node_failed` are counters not exercised by the light control-plane workload or not exported by Open5GS 2.7.5, and the `nrf_dereg` snapshot resolution was too coarse to catch de-registration within a fault window. These absences are discussed in Sections 3.3 and 4.8.

D Signal specificity

Table 8 ranks every signal that fired by specificity, the quantity introduced in Section 4.4: specificity $= 1 - H/H_{\max}$, where H is the entropy of the fault-class distribution among the faults a signal fires for. A specificity of 1.0 means the signal fires for a single class (an unambiguous localiser), while a value near 0 means it fires across nearly all classes (a high-recall but non-attributing alarm). Prevalence is the number of faults (of 22) for which the signal manifested, and "classes" the number of distinct fault classes (of 8) it spanned. The ten signals that never fired (Appendix C) are omitted.

Table 8: Per-signal specificity, all 30 signals that fired, ranked from most to least specific. Specificity 1.0 = fires for a single fault class (localiser), and low values = broad "something is wrong" alarms.

Signal	Prevalence	Classes	Specificity
amf_reg_fail	1	1	1.00
amf_auth_fail	1	1	1.00
pod_running_drop	1	1	1.00
net_rx	1	1	1.00
cpu_usage	5	2	0.76
smf_ues_active	2	2	0.67
log_nrf_lifecycle	2	2	0.67
log_ue_failures	2	2	0.67
upf_n4_estab	2	2	0.67
smf_n4_report_succ	2	2	0.67
smf_bearers_active	2	2	0.67
pfcp_sessions_active	2	2	0.67
pod_restart	4	2	0.67
amf_registered_sub	2	2	0.67
amf_gnb	4	3	0.50
log_errors	4	3	0.50
log_scp_routing	3	3	0.47
k8s_warning	7	4	0.39
pod_ready_drop	7	4	0.39
amf_ran_ue	5	4	0.36
smf_pdu_succ	7	4	0.35
cpu_throttle	8	5	0.33
amf_sessions	4	4	0.33
pfcp_peers_active	6	5	0.25
mem_working_set	6	5	0.25
beyla_srv_latency	11	6	0.20
beyla_cli_latency	13	6	0.16
trace_error_rate	18	7	0.11
rtt	11	8	0.09
trace_p95_latency	19	8	0.07

E Cross-run reproducibility atlas

Figure 4 overlays the published canonical atlas with the independent second run discussed in Section 4.7. Each cell shows, for one (fault, signal) pair, in how many of the two runs it manifested: black means both runs detected it, grey means only one did, and white means neither. Red vertical lines separate the infrastructure, orchestration, and application layers, and blue horizontal lines separate fault classes.

The black and white cells, 819 of 880 (93.1 %), are the reproducible core of the atlas, and the layer and class block structure is visibly the same as in Figure 1. The 61 grey cells are where the two runs disagree. They are balanced between the runs (32 fire only in the canonical run, 29 only in the second run) and concentrate on near-threshold application signals such as the trace and Beyla latencies and RTT, and on the least-stable faults, mainly `19-udm-pod-crash`, `07-pod-crash-smf`, and `13-pfcp-session-modification-dupl-upf`. The grey columns are sparse in the orchestration band, so the orchestration blind-spot result reproduces almost exactly across the two runs.

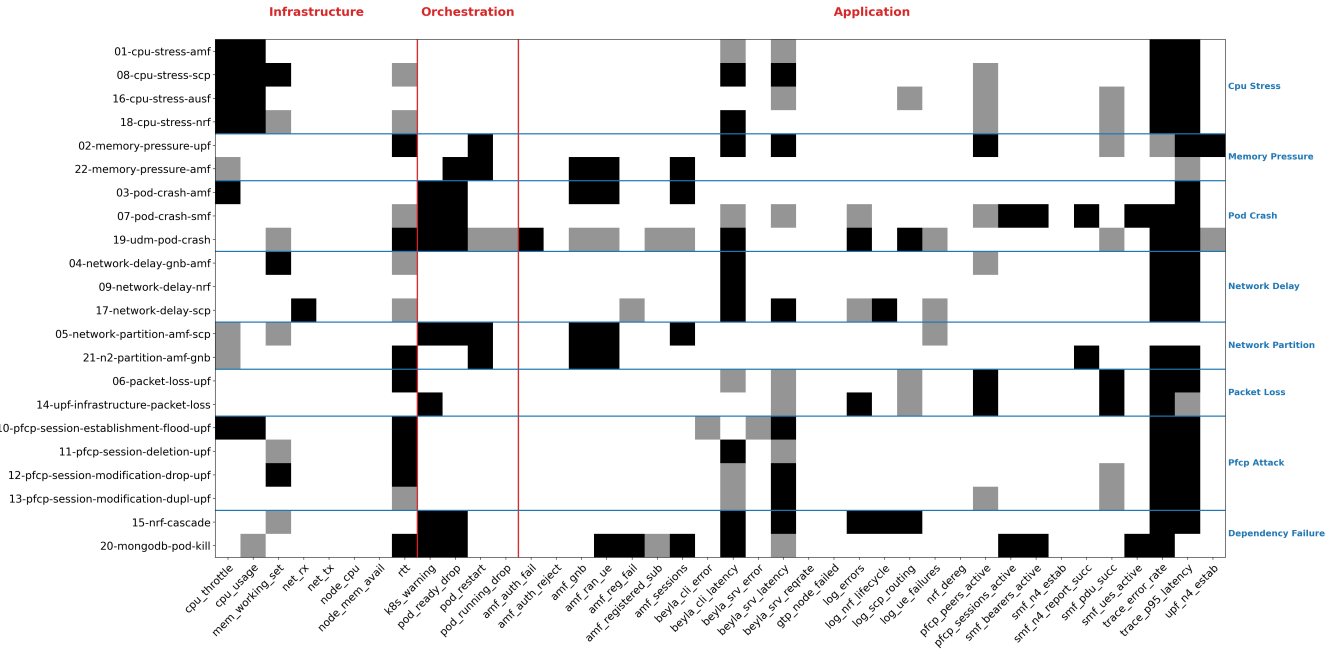

Figure 4: Cross-run reproducibility atlas (canonical run versus the independent second run). Cell shade is the fraction of the two runs in which each (fault, signal) cell fired: black = both runs, grey = one run only, white = neither. Red vertical lines separate the layers and blue horizontal lines separate fault classes, as in Figure 1. The two runs agree on 93.1 % of cells (Cohen's $\kappa = 0.768$), and the grey cells are the balanced, near-threshold disagreements.

The atlas in Figure 4 tests *whether* a cell fires; it says nothing about *when*. Because the per-layer timing in Figure 2 is one of our three main results, we check that the onsets reproduce too. Figure 5 is the timing companion to the cross-run atlas: for every fault and every layer it shows the signed difference in time-to-first-signal between the second run and the canonical run, on the same grid as Figure 2.

Of the 47 (fault, layer) cells that fire in both runs, 24 land on an identical onset bucket and 31 agree to within one 5 s bin, so two-thirds of the layer onsets are reproduced exactly at the resolution at which they are reported. Five cells flip between firing and silent (three fire only in the canonical run, two only in the second), and the single largest gap is 78 s. More importantly, the first-manifesting layer, which is the discriminator behind the timing result, agrees for 18 of the 22 faults. The four faults whose first layer moves are `06-packet-loss-upf`, `13-pfcp-session-modification-dupl-upf`, `21-n2-partition-amf-gnb`, and `20-mongodb-pod-kill`. The first three are the same pattern: the user-plane RTT and an application-layer trace both react inside the first 5 s bin, so which one is recorded first is a tie-break rather than a real change in propagation. The fourth is a crash, where the Kubernetes Warning and the first infrastructure metric race at t_0. The per-class layer ordering, the infrastructure-first/orchestration-first/ application-first split that the figure reports, is unchanged. The timing result therefore reproduces with the same robustness as the presence-based atlas.

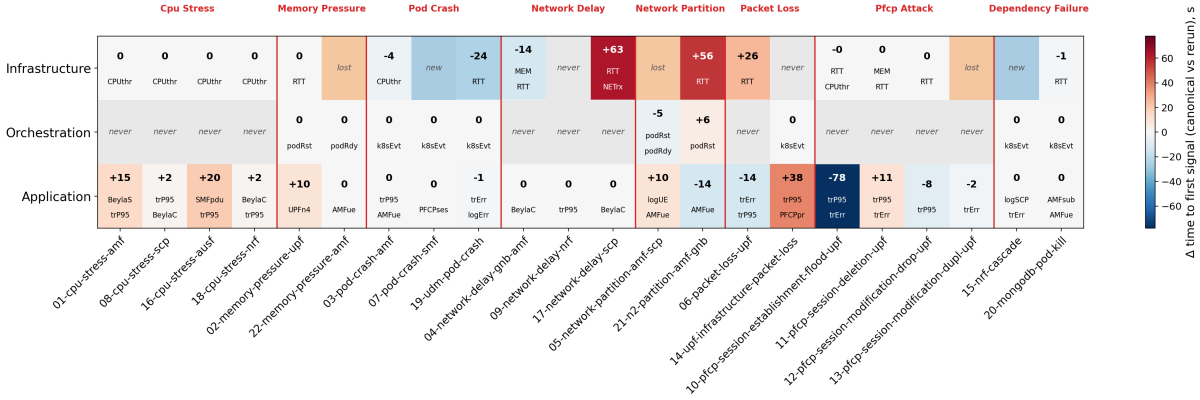

Figure 5: Cross-run timing reproducibility: per fault (column, grouped by class) and layer (row), the signed difference in time-to-first-signal between the second run and the canonical run, $\Delta = t_{\text{rerun}} - t_{\text{canonical}}$. White is an identical onset, blue means the second run was faster and red slower; `lost` marks a layer that fired only in the canonical run and `new` one that fired only in the second run, while `never` is silent in both. Each cell gives Δ in seconds and the first signal (both runs' signals are stacked when they differ). The layout matches Figure 2. Most cells are near zero and the disagreements concentrate on the same near-threshold RTT and trace signals as the cross-run atlas.

F Per-signal detector thresholds

Table 9 lists the per-signal constants of the detector defined in Section 3.5. For the continuous z-score signals these are the absolute floor f of Equation 1 and, where used, the multiplicative ratio guard r (the window mean must also reach $r\times$ or $1/r\times$ the baseline). The remaining signals use purpose-built detectors whose fixed rule is given in the last column. All signals share the global constants $Z = 3$, relative floor $\rho = 0.30$, 5 s bins with a 30 s window, and a two-window persistence requirement. Only f and r are tuned per signal. Byte floors are shown in kB or MB and the RTT floor in milliseconds.

Table 9: Per-signal detector thresholds. f is the absolute floor and r the ratio guard from Section 3.5, and "–" means the term is unused. The last column gives the fixed rule for the non-z-score detectors. The three caveat metrics excluded from scoring are omitted.

Signal	Detector	f	r	Fixed rule
Infrastructure				
cpu_usage	z-score	0.05	2.0	–
cpu_throttle	z-score	0.05	–	–
mem_working_set	z-score	20 MB	1.3	–
net_rx	z-score	10 kB/s	3.0	–
net_tx	z-score	10 kB/s	3.0	–
node_cpu	z-score	0.1	1.5	–
node_mem_avail	z-score	50 MB	1.15	–
rtt	z-score	5 ms	–	–
Orchestration				
pod_restart	step	–	–	restart count step up
pod_ready_drop	drop	–	–	gauge drops to 0
pod_running_drop	drop	–	–	gauge drops to 0
k8s_warning	event	–	–	Warning reason in window
Application: Beyla SBI				
beyla_srv_latency	z-score	0.01	1.5	–
beyla_cli_latency	z-score	0.01	1.5	–
beyla_srv_error	z-score	0.01	–	–
beyla_cli_error	z-score	0.01	–	–
beyla_srv_reqrate	z-score	0.5	2.0	–
Application: Open5GS state gauges				
amf_registered_sub	z-score	1	–	–
amf_ran_ue	z-score	1	–	–
amf_gnb	z-score	0.9	–	–
amf_sessions	z-score	1	–	–
pfcp_sessions_active	z-score	1	–	–
pfcp_peers_active	z-score	0.9	–	–
smf_ues_active	z-score	1	–	–
smf_bearers_active	z-score	1	–	–
smf_pdu_succ	z-score	0.02	2.0	–
smf_n4_estab	z-score	0.02	2.0	–
smf_n4_report_succ	z-score	0.02	2.0	–
upf_n4_estab	z-score	0.02	2.0	–
Application: failure counters				
amf_reg_fail	counter	–	–	$\geq$2 consec. $>$ pre-max
amf_auth_fail	counter	–	–	$\geq$2 consec. $>$ pre-max
amf_auth_reject	counter	–	–	$\geq$2 consec. $>$ pre-max
gtp_node_failed	counter	–	–	$\geq$2 consec. $>$ pre-max
Application: Loki logs				
log_errors	log rate	–	–	$\geq$3 extra lines/bin
log_ue_failures	log rate	–	–	$\geq$3 extra lines/bin
log_scp_routing	log rate	–	–	$\geq$3 extra lines/bin
log_nrf_lifecycle	log rate	–	–	$\geq$3 extra lines/bin
Application: Jaeger traces / NRF				
trace_error_rate	trace	–	–	error rate $\Delta \geq 0.05$
trace_p95_latency	trace	–	1.5	p95 $\geq 1.5\times$ baseline
nrf_dereg	NRF snap	–	–	pre vs during reg. delta