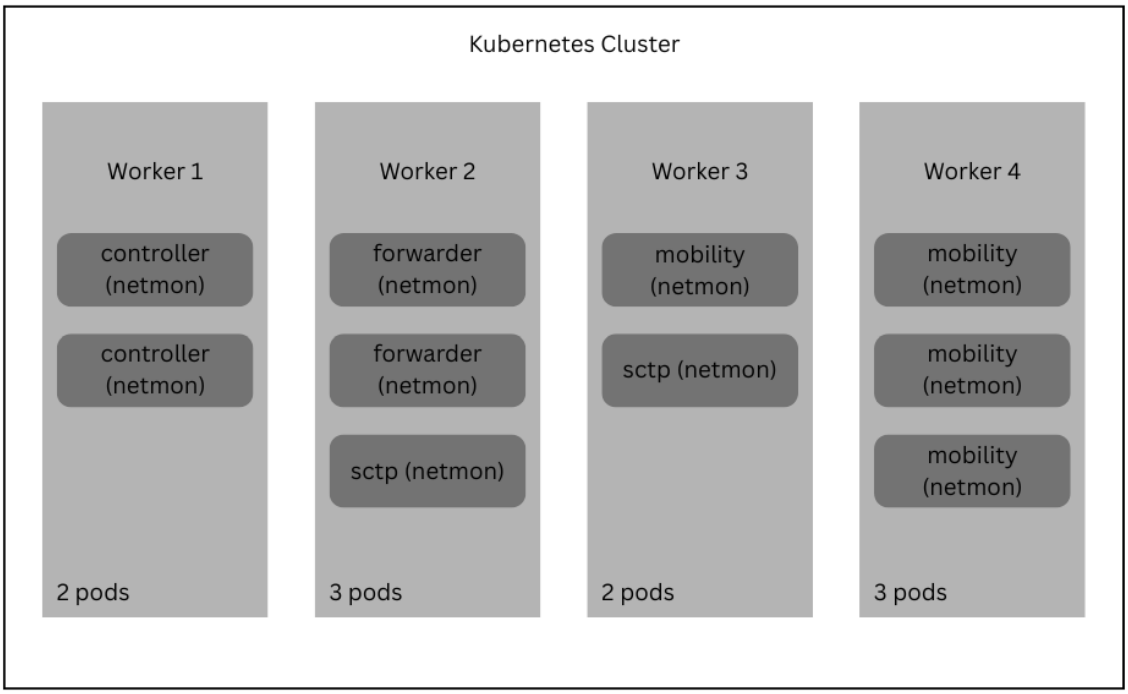

Hybrid Monitoring for Early Fault Detection in Cloud-Native 5G Systems

NetMon: A network monitoring tool designed to detect signs of faults in Ericsson's AMF clusters.

Master's thesis in Computer science and engineering

Anton Andersson, Sai Akshara Naineni

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2026

Master's thesis 2026

Hybrid Monitoring for Early Fault Detection in Cloud-Native 5G Systems

NetMon: A network monitoring tool designed to detect signs of faults in Ericsson's AMF clusters.

Anton Andersson, Sai Akshara Naineni

UNIVERSITY OF
GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2026

Hybrid Monitoring for Early Fault Detection in Cloud-Native 5G Systems
NetMon: A network monitoring tool designed to detect signs of faults in Ericsson's AMF clusters.
Anton Andersson, Sai Akshara Naineni

Supervisor: Yixing Zhang, Computer and Network Systems
Advisor: Mats Jansborg, Ericsson
Examiner: Romaric Duvignau, Computer and Network Systems

Master's Thesis 2026
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Cover: Cluster topology with 10 pods across 4 worker nodes.

Typeset in LaTeX
Gothenburg, Sweden 2026

Hybrid Monitoring for Early Fault Detection in Cloud-Native 5G Systems
NetMon: A network monitoring tool designed to detect signs of faults in Ericsson's AMF clusters.
Anton Andersson
Sai Akshara Naineni
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

This thesis presents the design, implementation, and evaluation of a hybrid network monitoring system for Kubernetes-based 5G packet core deployments. The system combines eBPF-based passive kernel-level traffic observation with active TCP probing and centralized correlation to detect and localize network degradation within seconds. The evaluation results demonstrate that the system detects faults as subtle as 10ms of added latency or 5% packet loss, correctly attributes them to the affected infrastructure component, and maintains this capability under application loads up to 50% of the cluster's capacity. The total resource overhead of 3.4 millicores CPU and 4.5 MiB memory per pod confirms that the approach is viable for production deployment without impacting the monitored workload. The hybrid approach addresses a gap in existing monitoring tools: standard health checks cannot detect partial degradation, scrape-based systems introduce detection delays measured in tens of seconds, and purely passive tools cannot verify idle network paths. By combining these complementary techniques and centralizing the analysis, the system provides the early detection and fault localization capabilities required for maintaining service quality in cloud-native 5G infrastructure.

Keywords: Kubernetes, eBPF, Observability, Monitoring, Computer Networks, Cluster Networks, 5G.

Acknowledgements

We would like to thank our supervisor and examiner at Chalmers University of Technology, Yixing Zhang and Romaric Duvignau, for their guidance and feedback throughout this project. We also thank our advisor at Ericsson, Mats Jansborg, for his technical expertise and support, and our manager Patrik Lindberg for making this collaboration possible.

Anton Andersson, Gothenburg, 2026-07-02
Sai Akshara Naineni, Gothenburg 2026-07-02

Contents

List of Figures **xiii**

List of Tables **xv**

1 Introduction **1**
1.1 Motivation . . . 1
1.2 Problem Statement . . . 3
1.3 Goals . . . 4
1.4 Contributions . . . 4
1.5 Limitations . . . 5
1.6 Thesis Structure . . . 5

2 Background **7**
2.1 Cloud-Native 5G Core Networks . . . 7
2.1.1 Service-based Architecture . . . 7
2.1.2 Kubernetes as the Deployment Platform . . . 8
2.1.3 Failure Modes in Cloud-Native Systems . . . 9
2.1.3.1 Pod-Level Failures . . . 10
2.1.3.2 Network Partitions . . . 10
2.1.3.3 Resource Exhaustion . . . 11
2.1.3.4 Silent Degradation . . . 11
2.2 Extended Berkeley Packet Filter (eBPF) . . . 12
2.2.1 Architecture . . . 12
2.2.2 Use Cases . . . 13
2.2.3 Advantages for Network Monitoring . . . 13
2.3 Network Monitoring Approaches . . . 14
2.3.1 Passive Monitoring . . . 14
2.3.2 Active Monitoring . . . 15
2.3.3 The Case for Hybrid Monitoring . . . 15
2.4 Anomaly Detection in Distributed Systems . . . 16
2.4.1 Statistical Methods . . . 16
2.4.2 Machine Learning Approaches . . . 17
2.4.3 Rule-Based Detection . . . 17
2.4.4 Correlation and Attribution . . . 18
2.5 Related Work . . . 18

2.5.1 eBPF-Based Monitoring for 5G . . . 18
2.5.2 Cluster Monitoring and Distributed Telemetry . . . 19
2.5.3 Gray Failure Detection in Practice . . . 19
2.5.4 Summary and Research Gap . . . 21

3 Methods **23**
3.1 System Architecture . . . 23
3.1.1 Architecture overview . . . 23
3.1.2 Design Goals . . . 25
3.1.3 Active Probing Mechanism . . . 26
3.1.4 eBPF Passive Monitoring . . . 26
3.1.4.1 XDP Program: Port and Traffic Monitoring . . . 26
3.1.4.2 TC Program: Kernel-Level RTT Measurement . . . 27
3.1.5 Local Anomaly Detection . . . 28
3.1.5.1 Statistical Methods . . . 28
3.1.5.2 Warmup Suppression . . . 29
3.1.5.3 Types of Anomalies . . . 29
3.1.6 Metrics Reporting . . . 30
3.2 Central Server Design and Implementation . . . 30
3.2.1 Pod State Management . . . 30
3.2.2 Correlation Engine . . . 31
3.2.3 Online Change-Point Detection (CUSUM) . . . 32
3.2.4 Topology Awareness . . . 33
3.2.5 Reporting . . . 34
3.3 System Implementation . . . 34
3.3.1 Deployment . . . 34
3.3.2 Design Trade-offs . . . 35
3.4 Evaluation Methodology . . . 36
3.4.1 Evaluation Environment . . . 36
3.4.2 Fault Injection Scenarios . . . 37
3.4.3 Evaluation Metrics . . . 38
3.4.3.1 Detection Latency . . . 38
3.4.3.2 Correlation Latency . . . 39
3.4.3.3 Resource Overhead . . . 39

4 Results **41**
4.1 User Experience Results . . . 41
4.2 CPU/Memory Usage and Traffic Overhead . . . 43
4.3 Baselines during normal operation . . . 45
4.4 Experiment Results . . . 46
4.4.1 Latency Injection . . . 46
4.4.2 Jitter Injection . . . 47
4.4.3 Packet Loss . . . 48
4.4.4 Bandwidth Throttle . . . 49
4.4.5 Worker-to-Worker Link Failure . . . 50
4.4.6 Full Network Partition . . . 50
4.4.7 Pod Termination . . . 51

4.4.8 Gradual Degradation 51
4.5 Summary 52

5 Discussion and Conclusion **55**
5.1 Discussion 55
5.2 Limitations 56
5.3 Future Work 58
5.4 Conclusion 59

Bibliography **61**

A Appendix **I**
A.1 CNOM Widgets I

List of Figures

1.1 The 5G CN architecture with a focus on the N4 interface. Adapted from [4]. 2

2.1 An overview of the 5G Service-based architecture [11]. 8
2.2 An overview of the Kubernetes architecture [14]. 10

3.1 Architecture overview of Centralized network monitoring for 5G pods in Kubernetes. 25
3.2 Cluster topology with 10 pods across 4 worker nodes. 37

4.1 A screenshot of the CNOM dashboard during 20ms latency injection, showing the first 10 widgets. 43
4.2 Detection and correlation latencies across all fault types under 0% simulated UE load. 53
4.3 Detection and correlation latencies across all fault types under 20% simulated UE load. 53
4.4 Detection and correlation latencies across all fault types under 50% simulated UE load. 54

A.1 Anomaly History (by type, only last 30s tracked). I
A.2 Anomaly History (by target, last 30s). II
A.3 Active correlations. II
A.4 Worker Health Score (0-100). II
A.5 Pod Health State 5=HEALTHY, 4=INSUFFICIENT, 3=DEGRADED, 2=WARNING, 1=FAILING, 0=DOWN. III
A.6 RTT (ms) grouped by src_worker,dst_worker. III
A.7 Two Lifetime anomalies widgets by target and type. IV
A.8 List of monitored pods containing pod IP, pod name, status, worker node, ready state, number of restarts, and age of pod. IV
A.9 Avg RTT (ms) by Pod. V
A.10 RTT EMA (ms) by Pod. V
A.11 Jitter (ms) by Pod. VI
A.12 Total Probes per Pod. VI
A.13 Total Failures per Pod. VI
A.14 Packets/s by Pod. VII
A.15 Throughput (KB/s) by Pod. VII
A.16 RST% by Pod. VII

A.17 Retransmit% by Pod. VII

List of Tables

2.1 Comparison of network monitoring approaches for Kubernetes environments. 21

3.1 Anomaly types detected by the agent. 29
3.2 Fault injection scenarios and configurations. 38

4.1 Anomaly and correlation rates during normal operation (0% load, no injected faults, 18 hours). 45
4.2 Detection and correlation latency for latency injection at varying load levels. Detection = time to first anomaly involving the faulty worker. Correlation = time to first `WORKER_HIGH_LATENCY`. All values in seconds. 47
4.3 Detection and correlation latency for jitter injection (5ms ± 20ms, normal distribution). Correlation = time to first `WORKER_HIGH_LATENCY`. All values in seconds. 48
4.4 Detection and correlation latency for packet loss injection. Detection = time to first RETX anomaly. Correlation = time to first `WORKER_SOURCE_HOTSPOT`. All values in seconds; - indicates correlation did not fire. 49
4.5 Detection and correlation latency for bandwidth throttling at 0% load. Correlation = time to first `WORKER_SOURCE_HOTSPOT`; – indicates it did not fire. 49
4.6 Detection and correlation latency for worker-to-worker link failure (worker-2 ↔ worker-3) at 0% load. Detection = time to first `PROBE_FAILURE`. Correlation shows both `WORKER_ISOLATED` (first to fire) and `PEER_DOWN`. 50
4.7 Detection and correlation latency for full network partition (worker-2 isolated from all other workers) at 0% load. Detection and correlation fire simultaneously. 51
4.8 Detection and correlation latency for pod termination at 0% load. Detection = time to first `PROBE_FAILURE`. Correlation = time to first `PEER_DOWN`. 51
4.9 Detection and correlation latency for gradual latency increase. Detection = time to first anomaly involving worker-2. Correlation = time to first `WORKER_HIGH_LATENCY`. All values in seconds. 52

1

Introduction

This chapter motivates the need for improved network monitoring in cloud-native 5G deployments and introduces the monitoring system presented in this thesis. Section 1.1 describes the shift toward microservice-based 5G core networks and the observability challenges this creates. Section 1.2 formulates the specific problem addressed. Section 1.3 states the research questions. Section 1.4 summarizes the contributions, and Section 1.5 discusses known limitations. Section 1.6 outlines the structure of the remaining chapters.

1.1 Motivation

Monitoring has long been one of the central challenges in distributed systems, as applications are made up of many interacting components that fail and degrade in subtle and frequently non-obvious ways. As systems scale to greater size and become more dynamic, it becomes increasingly hard to understand whether performance issues originate with application logic, shared resources, or the underlying network. Modern monitoring approaches therefore focus on the need for constant visibility, minimal overhead, and early anomaly detection before any failures become visible [1].

These challenges have become much more pronounced with the shift toward cloud-native architectures in 5G core networks. Larger network functions are giving way to microservice-based components deployed across containerized infrastructure managed by Kubernetes. While this brings improvements in scalability and flexibility, the communication paths also become much more complicated, often making performance diagnosis challenging [2]. Network paths between services change dynamically, and small degradations can propagate across components. When these paths begin to deteriorate, the first hints are subtle: higher jitter, slower tail latency, or a few dropped packets that easily go unnoticed [3]. These signals appear long before anyone would call the situation a fault, yet they often explain why applications suddenly feel slow.

A brief overview of the 5G core can help contextualize these challenges. The 5G core network is broadly split into two planes: Control Plane and User Plane. The control plane is responsible for signaling, decision-making, and coordination between network functions. It includes functions such as the Access and Mobility Management Function (AMF), Session Management Function (SMF), Policy Control Function

(PCF), and supporting data and analytics functions. These control-plane functions communicate frequently with each other using service-based interfaces and transport protocols designed for reliability and fast failover [4].

The user plane, in contrast, is responsible for forwarding actual user data packets. This functionality is primarily implemented by the User Plane Function (UPF), which forwards traffic between the access network and external data networks. Communication between the control plane and the user plane is carried out through well-defined interfaces, such as the N4 interface between the SMF and the UPF as shown in Figure 1.1. Performance issues on these interfaces can directly affect user traffic, even if all individual network functions appear healthy [4].

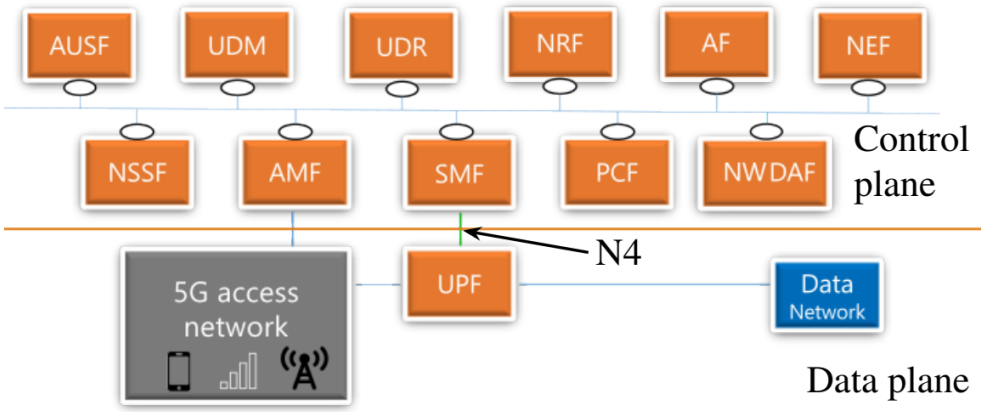

Figure 1.1: The 5G CN architecture with a focus on the N4 interface. Adapted from [4].

Existing monitoring approaches each address part of this problem but leave significant gaps. Active probing can uncover failures but does not scale well in large deployments, since the amount of probe traffic grows quickly with the number of nodes and links that must be tested [5]. Metrics-based monitoring tools such as Prometheus operate on scrape intervals of 15–60 seconds, introducing latency that can miss transient degradations. Logging is inherently reactive, useful for post-mortem analysis but unable to provide early warning. Distributed tracing captures application-level interactions but misses kernel and network layer issues entirely. None of these approaches, on their own, provide the kind of early, low-overhead detection needed to catch subtle degradation before it cascades into visible faults.

This motivates the need for a complementary source of observability at the kernel level. Extended Berkeley Packet Filter (eBPF) enables low-overhead monitoring of transport-level behavior, socket activity, and packet processing directly within the Linux kernel, without modifying application code or injecting additional traffic [6]. By combining this passive kernel-level telemetry with targeted active probing, it becomes possible to detect degradation early while keeping overhead low enough for production environments.

This thesis presents *NetMon*, a hybrid monitoring tool that combines passive kernel-level observation via eBPF with active TCP probing between pods in a Kubernetes cluster. Each pod runs a monitoring agent that continuously probes its peers and observes kernel-measured latency, jitter, retransmission rates, and TCP anomalies. These observations are reported to a central aggregation component that correlates anomalies across the cluster to detect failure patterns such as peer failures, latency sources, worker node isolation, and network partitions. The system is designed to detect degradation before it triggers existing health checks or disrupts operations.

1.2 Problem Statement

The core problem is detecting subtle network degradation such as small shifts in latency, jitter increases, and intermittent packet loss, before it cascades into visible failures, while keeping monitoring overhead low enough for production 5G deployments.

Most monitoring tools only detect issues once they have already started affecting applications. As industry discussions on proactive monitoring highlight, depending solely on visible faults leads to longer outages and higher operational costs, and early anomalies need to be handled before they turn into major issues [7] [3]. Running active probes constantly is not practical either, since probe traffic grows with the size of the system and quickly becomes expensive [5]. On the other hand, passive telemetry can be imprecise and often mistakes normal fluctuations for real degradation, which adds to alert fatigue [7]. Any solution must therefore operate continuously, stay lightweight, and scale to cloud-native 5G environments while distinguishing genuine degradation from normal variation.

In the current AMF architecture, monitoring is primarily focused on liveness and health checks on key control-plane interfaces to ensure that specific paths are functioning correctly. These methods are not effective at detecting the early stages of degradation within communication paths between AMF microservices, including SCTP-based transport links and internal service-to-service flows. By the time existing health checks detect a problem, the degradation has often already affected operations.

This thesis addresses this gap through *NetMon*, a hybrid monitoring tool in which lightweight agents deployed on each pod perform continuous active TCP probing of their peers while simultaneously collecting passive kernel-level metrics via eBPF, including round-trip time, jitter, retransmission rates, and TCP flag anomalies. These per-pod observations are reported to a central aggregation component that correlates anomalies across the cluster to identify failure patterns and localize their source. The decision to use eBPF was made in order to gather kernel-level metrics without modifying any of the existing application code or loading custom kernel modules of the Kubernetes pod [8].

1.3 Goals

This master's thesis aims to design and evaluate a lightweight hybrid monitoring framework for early detection and localization of network degradation in cloud-native 5G systems, with a specific focus on the Access and Mobility Management Function (AMF). The work includes both the design of a hybrid detection model and the implementation of a working monitoring system.

The research is guided by the following main research question:

- How can early-stage network degradation and impending faults in cloud-native 5G systems be detected and localized using a lightweight hybrid approach that combines passive kernel-level monitoring with scalable active probing techniques?

This is supported by the following sub-questions:

- Can kernel-level metrics such as RTT, jitter, and TCP anomalies detect degradation before application-level monitoring or existing health checks?
- How can anomalies observed independently by multiple pods be correlated to distinguish between pod-level faults, worker node issues, and network partitions?
- What is the resource overhead of the hybrid approach compared to existing monitoring solutions?

The thesis demonstrates that kernel-measured RTT, jitter, retransmission rates, and TCP flag anomalies can be correlated across pods to identify peer failures, latency sources, anomaly hotspots, worker isolation, inter-node failures, and pod isolation. By capturing these signals at points that currently lack fine-grained metrics such as internal AMF microservice communication paths, the system detects anomalies before they trigger existing controller-level health checks or disrupt AMF operations.

Overall, the research works toward a practical, data-driven monitoring framework that helps 5G operators detect and localize issues early while keeping the system lightweight and within scalability constraints.

1.4 Contributions

This thesis makes the following contributions:

- **Hybrid monitoring architecture.** A monitoring system that combines passive kernel-level telemetry via eBPF with active TCP probing for early fault detection in Kubernetes clusters, designed as a general-purpose tool and evaluated in a 5G AMF deployment.
- **Cluster-wide anomaly correlation engine.** A correlation layer that aggregates anomaly reports from distributed agents and identifies failure patterns

such as: peer down, latency source, anomaly hotspot, worker isolation, inter-node failure, and pod isolation. This enables distinguishing between pod-level faults, worker node issues, and network partitions from independently observed signals.

- **Real-time visualization pipeline.** Integration with Prometheus and Ericsson's Cloud Native Operations Manager for continuous export and visualization of per-pod and per-link network health metrics, including kernel-measured RTT, jitter, retransmission rates, and TCP anomalies.
- **Evaluation in a live 5G AMF environment.** Assessment of detection timeliness, localization accuracy, and resource overhead in a multi-worker Kubernetes cluster running AMF workloads, extending monitoring coverage to internal communication paths that previously lacked fine-grained metrics.

1.5 Limitations

Several constraints scope the applicability of this work. These are summarized here and discussed in detail in Section 5.2.

- **Elevated Linux capabilities:** NetMon's eBPF programs require `CAP_BPF` and `CAP_NET_ADMIN`, which may not be possible to acquire in hardened production environments [8], [9].
- **Probe-based observation model:** NetMon generates lightweight TCP probe traffic rather than performing deep inspection of all application traffic, meaning its view of network health is based primarily on probe behavior and lightweight kernel packet-inspection.
- **Resource overhead:** The AMF has strict reliability requirements, and any monitoring solution must avoid introducing noticeable performance degradation. Measuring the actual overhead is part of the evaluation.

1.6 Thesis Structure

The remainder of this thesis is organized as follows:

- **Chapter 2: Background** provides the background on cloud-native 5G core networks, Kubernetes networking, eBPF, and anomaly detection techniques. Reviews related work in network monitoring, fault detection, and observability for distributed systems.
- **Chapter 3: Methods** describes the system architecture, including the agent-based monitoring model, the eBPF programs for passive kernel-level telemetry, the active TCP probing mechanism, the central aggregation and correlation engine, and the evaluation methodology including the fault injection scenarios and metrics used.

- **Chapter 4: Results** presents the evaluation results including resource overhead measurements, baseline characterization, detection and correlation latencies across all fault types and load levels, and a summary of key findings.
- **Chapter 5: Discussion and Conclusion** interprets the results, compares the approach against existing monitoring paradigms, discusses limitations, and outlines directions for future work.

2

Background

This chapter provides the theoretical background and context necessary to understand the design and evaluation of the monitoring system presented in this thesis. Section 2.1 introduces the 5G core network architecture and its deployment on Kubernetes, including the failure modes that arise from this combination. Section 2.2 covers eBPF, the in-kernel runtime that enables low-overhead packet inspection. Section 2.3 reviews existing network monitoring approaches and identifies the limitations that motivate a hybrid design. Section 2.4 describes anomaly detection techniques relevant to network monitoring. Finally, Section 2.5 surveys related tools and research, positioning the contribution of this thesis within the existing body of work.

2.1 Cloud-Native 5G Core Networks

This section provides the necessary background on the 5G core network architecture and its deployment model. Section 2.1.1 describes the service-based architecture that defines how network functions communicate. Section 2.1.2 covers Kubernetes as the platform on which these functions are deployed. Section 2.1.3 discusses the failure modes that arise from this combination of architecture and platform.

2.1.1 Service-based Architecture

The architecture of mobile core networks has undergone a change between the fourth and fifth generations. In 3G and 4G systems, the core network consisted of a small number of monolithic network elements connected through well defined point-to-point interfaces. Each element performed a broad set of functions, and communication between them followed rigid, protocol-specific paths. The 5G core network, as specified by 3GPP in TS 23.501 [10], replaces this model with a Service-Based Architecture (SBA) in which network functions expose their capabilities as services through standardized APIs and communicate over a common service bus.

The key network functions in the 5G core include:

- **Access and Mobility Management Function (AMF)**, which handles signaling for every connected device and manages mobility procedures.

- **Session Management Function (SMF)**, which establishes and manages user sessions.
- **User Plane Function (UPF)**, which forwards user data traffic.
- **Policy Control Function (PCF)**, which provides policy rules.
- **Network Repository Function (NRF)**, which enables service discovery between network functions.

Control-plane communication between these functions uses HTTP service-based interfaces (SBI), while user-plane traffic is carried over GTP-U tunnels and managed through the N4 interface between the SMF and UPF [4], [10]. The AMF is the focus of this thesis: it is involved in every UE connection procedure, making it both latency-sensitive and high-throughput.

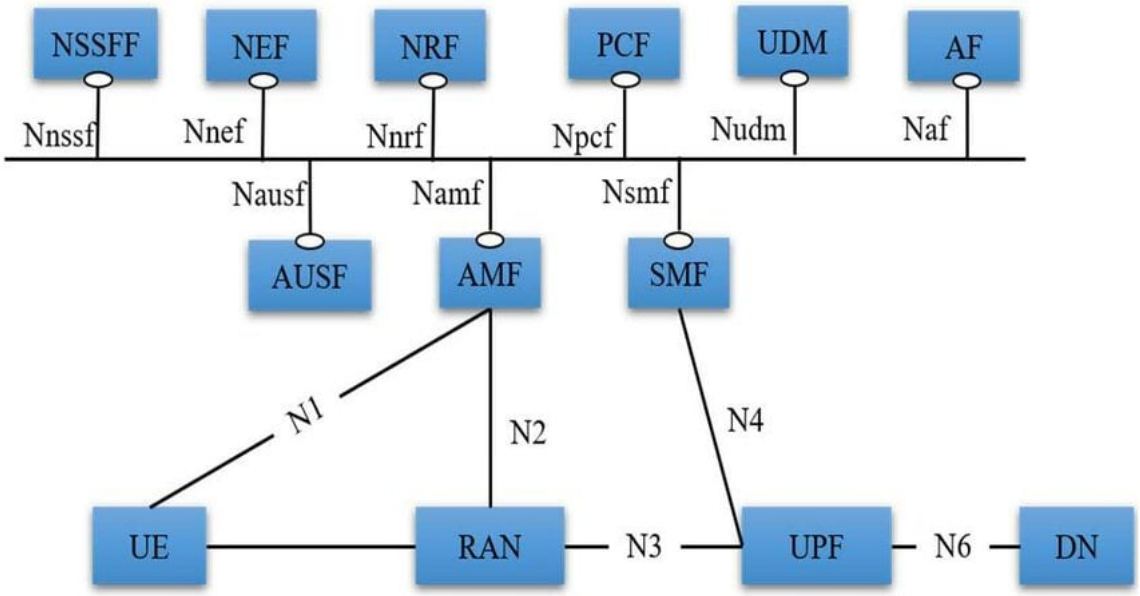

Figure 2.1: An overview of the 5G Service-based architecture [11].

The service-based communication model means that a single procedure, such as a UE attaching to the network, can trigger a chain of service requests across multiple network functions. An attach procedure may involve interactions across the AMF, SMF, UPF, and PCF in sequence, where the end-to-end procedure time is the sum of the individual hop latencies. This creates a system where degradation at any point in the chain propagates to the overall procedure time [3]. A small increase in latency between the AMF and SMF, for example, may not be visible in isolation but becomes significant when multiplied across thousands of concurrent procedures.

2.1.2 Kubernetes as the Deployment Platform

The 5G core network functions described above are deployed as containerized microservices orchestrated by Kubernetes [12]. In this model, each microservice runs

inside one or more *pods*, the smallest deployable unit in Kubernetes. Each pod is assigned a unique IP address within the cluster and can contain one or more containers that share the same network namespace. A single network function such as the AMF may be decomposed into multiple microservices, for example, separate components for mobility management, SCTP transport, and packet forwarding, each running as distinct pods. Kubernetes handles the scheduling of pods onto worker nodes, automatic restart of failed pods, and horizontal scaling based on resource utilization.

The networking model in Kubernetes requires that every pod can communicate directly with every other pod without network address translation. This flat network space is implemented by Container Network Interface (CNI) plugins, which are responsible for assigning IP addresses to pods and configuring routing between them. Different CNI plugins use different mechanisms to achieve this [13]. Within a single worker node, pod-to-pod traffic typically traverses virtual Ethernet pairs and a Linux bridge, staying entirely in kernel space. Across worker nodes, traffic must traverse the physical network, passing through switches and network interface cards. This networking layer is transparent to applications but is where many subtle failures originate, packet loss from encapsulation overhead, added latency from overlay routing, or MTU mismatches causing fragmentation.

A Kubernetes cluster consists of multiple worker nodes, called nodes in figure 2.2, each running a kubelet agent and a container runtime [14]. The Kubernetes scheduler distributes pods across workers based on resource availability and scheduling constraints. This means that two pods belonging to the same network function may run on different physical machines, and their communication traverses the inter-node network. The distinction between intra-worker and inter-worker communication is important for fault localization: a network issue between two worker nodes affects all pod-to-pod communication across that link, while a problem with a single pod affects only that endpoint. This topology, pods grouped on workers, workers connected by physical links, is a structure that NetMon's correlation engine exploits to distinguish between pod-level, worker-level, and network-level failures.

Pods discover each other through Kubernetes Services, which provide stable DNS names that resolve to the current set of pod IP addresses backing that service [15]. When a pod is terminated and replaced, the new pod receives a different IP address and the DNS records are updated accordingly. This dynamic nature of pod identities means that the set of peers a monitoring agent communicates with changes over time. NetMon uses this DNS-based service discovery to resolve its peer list on each probe round, which means that when a pod is killed, it will eventually disappear from the peer list. This behavior is relevant to the design of the monitoring system, as discussed in later chapters.

2.1.3 Failure Modes in Cloud-Native Systems

Cloud-native deployments are subject to a wide range of failure modes, including node crashes, storage failures, control plane outages, and certificate expiration, among others. Many of these are fail-stop in nature and are effectively handled

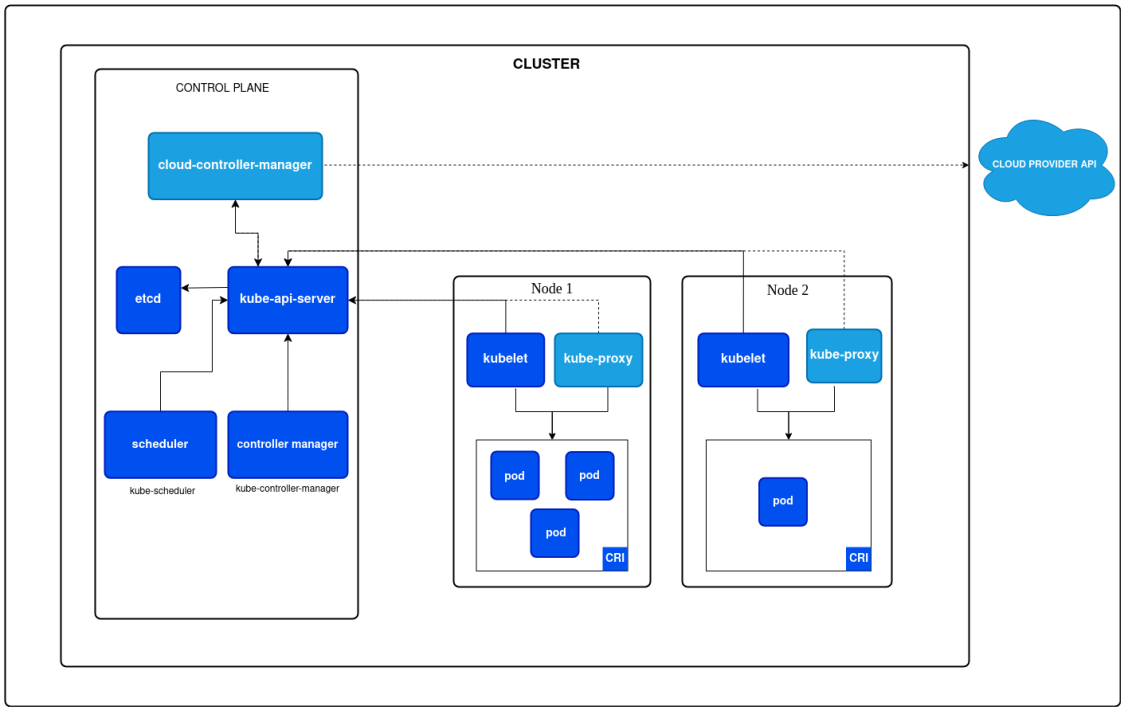

Figure 2.2: An overview of the Kubernetes architecture [14].

by Kubernetes' built-in recovery mechanisms such as liveness probes, node heartbeats, and automatic pod rescheduling [14]. This section focuses on four categories of failures that share a critical property: they manifest as observable changes in network-level behavior: probe reachability, latency, jitter, and packet loss, and are therefore detectable through the hybrid monitoring approach proposed in this thesis. These correspond to what Huang et al. call "*gray failures*": degradations that existing failure detectors often miss because the system appears nominally operational while applications experience degraded service [16].

2.1.3.1 Pod-Level Failures

The most straightforward failure mode is a pod crash caused by out-of-memory (OOM) kills, unhandled application exceptions, or failed liveness probes. Kubernetes automatically restarts crashed pods, but there is a gap between the failure and recovery during which the pod is unreachable. If a pod crashes repeatedly, Kubernetes enters a CrashLoopBackOff state where restart attempts are progressively delayed, leaving the pod intermittently available for extended periods [17]. These failures are the easiest to detect because the pod simply stops responding to probes, but the transient unavailability during restarts can still cause cascading effects on dependent services.

2.1.3.2 Network Partitions

A more complex failure mode is a network partition, where connectivity between parts of the cluster is lost while other paths remain functional. Worker-to-worker

partitions occur when the physical network link between two nodes fails or degrades: all pods on one worker lose connectivity to all pods on the other, but communication within a worker remains healthy [18]. These failures are harder to detect because each individual pod sees only its own perspective, a pod experiencing a partition cannot distinguish between a remote pod being down and the network path being broken. Identifying partitions requires correlating observations from multiple pods, which is one of the core capabilities of NetMon's correlation engine. Prior work on partition detection in similar environments has shown that active probing can identify these failures but faces scalability challenges as the number of nodes grows [5].

2.1.3.3 Resource Exhaustion

Kubernetes enforces resource limits on CPU and memory for each pod. When a pod exceeds its CPU limit, the kernel throttles its execution, causing increased latency in processing requests without any explicit error. Memory pressure on a worker node can trigger the kernel's OOM killer, degrading I/O performance for all pods on that node [19]. Network bandwidth saturation on a worker node's physical network interface affects all pods scheduled on that node simultaneously. These resource exhaustion scenarios cause gradual degradation rather than hard failures: connections remain established, responses are still delivered, but latency increases and throughput drops. This makes them difficult to distinguish from normal load variation using application-level health checks alone [16].

2.1.3.4 Silent Degradation

The most insidious failure mode is silent degradation: small increases in round-trip time, occasional packet loss, rising retransmission rates, or jitter spikes that do not trigger any existing health check or alert. These signals often precede hard failure, and a degrading network link will eventually fail completely, but the degradation phase can last minutes or hours [3]. During this phase, applications may still function but with subtly degraded performance that accumulates across the service chain. Existing monitoring tools that rely on binary health checks or coarse-grained metrics with scrape intervals of 15-60 seconds miss these signals entirely. Industry discussions on proactive monitoring emphasize that detecting these early anomalies before they escalate is critical to maintaining the availability targets required by 5G deployments [7]. This category of failure is the primary target of NetMon's hybrid monitoring approach.

Taken together, these failure modes illustrate why monitoring a cloud-native 5G core requires visibility at the kernel level to observe subtle transport-layer signals, correlation across multiple pods to distinguish local issues from cluster-wide patterns, and topology awareness to separate pod-level faults from worker-level and network-level problems. The following sections describe the technical foundations that NetMon builds upon to address these requirements.

2.2 Extended Berkeley Packet Filter (eBPF)

eBPF is a general-purpose in-kernel runtime in the Linux kernel that allows sandboxed programs to execute at various hook points without modifying kernel source code or loading kernel modules [20]. Originally derived from the Berkeley Packet Filter (BPF), a mechanism for efficient packet capture introduced by McCanne and Jacobson [21], eBPF has evolved into a programmable platform for networking, observability, and security [8]. Programs are written in restricted C, compiled to eBPF bytecode, and verified by an in-kernel static analyzer before being JIT-compiled to native machine instructions. This combination of safety guarantees and near-native performance makes eBPF particularly suited for network monitoring in production environments, where instrumentation must not compromise system stability or introduce significant overhead.

2.2.1 Architecture

An eBPF program is written in a restricted subset of C, compiled to eBPF bytecode, and loaded into the kernel at runtime. Before execution, every program must pass through the eBPF *verifier*, a static analysis component that checks safety properties including memory access validity, bounded execution (no infinite loops), and type correctness [20]. The verifier simulates all possible execution paths and rejects any program that could crash, hang, or access memory outside its permitted range [8]. This verification step is what distinguishes eBPF from traditional kernel modules: it provides safety guarantees without requiring kernel recompilation or reboot.

Once verified, the program is JIT-compiled to machine code and attached to a *hook point* in the kernel. Hook points define where in the kernel's execution flow the program runs. The hook points most relevant to network monitoring are:

- **XDP (eXpress Data Path)**: Executes at the earliest possible point in the network receive path, inside the network driver's receive function, before the kernel allocates a socket buffer (`sk_buff`) for the packet [22]. This allows packet inspection and filtering with minimal overhead, as packets that are dropped or redirected never enter the kernel's networking stack. XDP programs can return verdicts to pass, drop, redirect, or transmit packets.
- **TC (Traffic Control)**: Attaches to the kernel's traffic control layer, which processes packets after socket buffer allocation but before delivery to the application. TC hooks exist for both ingress and egress traffic, making them suitable for inspecting outgoing packets, a capability that XDP lacks, since XDP operates only on the receive path [8]. TC programs have access to the full `sk_buff` structure, enabling richer packet metadata inspection at the cost of slightly higher overhead compared to XDP.

Other hook points include kprobes (dynamic instrumentation of arbitrary kernel functions), tracepoints (static instrumentation points compiled into the kernel), and cgroup hooks (for per-container resource control), though these are not used in the system presented in this thesis [20].

Communication between eBPF programs running in kernel space and userspace applications is facilitated through *BPF maps*, shared data structures that both sides can read from and write to. Maps support various types including hash tables, arrays, ring buffers, and LRU caches. A common pattern is for an eBPF program to aggregate per-packet statistics into a map, which a userspace process periodically reads to produce monitoring reports [8]. Ring buffer maps provide an event-driven alternative, allowing the kernel-side program to push individual events to userspace with minimal latency.

2.2.2 Use Cases

The general nature of the eBPF runtime has led to its adoption across several domains in cloud-native infrastructure:

- **Networking**: Cilium is a CNI plugin for Kubernetes that replaces the traditional iptables-based packet processing path with eBPF programs, providing identity-based network policies and load balancing directly in the kernel [8].
- **Observability**: Pixie, a CNCF project, uses eBPF to automatically capture application-level telemetry, including HTTP, gRPC, and DNS requests, without requiring application instrumentation or sidecar containers [8].
- **Security**: Falco, another CNCF project, attaches eBPF programs to system call tracepoints to detect anomalous runtime behavior such as unexpected process execution, file access, or network connections inside containers [8].

These projects illustrate a common pattern: eBPF enables kernel-level visibility and control that previously required either kernel modifications or userspace proxies, both of which carry significant performance or maintenance costs.

2.2.3 Advantages for Network Monitoring

eBPF offers several properties that make it particularly suited for network monitoring in latency-sensitive environments such as 5G core networks:

- **Zero-copy packet inspection**: XDP programs inspect packet data directly in the driver's receive buffer, without copying it into a separate kernel buffer or to userspace. This eliminates the memory copy overhead that traditional packet capture mechanisms such as `libpcap` incur [22].
- **Low processing overhead**: Because eBPF programs are JIT-compiled to native machine code and execute within the kernel, they avoid the context switches between kernel and userspace that characterize traditional monitoring approaches. Høiland-Jørgensen et al. demonstrated that XDP achieves single-core packet processing rates of up to 24 million packets per second [22].
- **No kernel module compilation**: Unlike traditional kernel modules, which must be compiled against specific kernel headers and loaded with full kernel privileges, eBPF programs are verified for safety at load time and can be deployed without recompiling the kernel or rebooting the system [20]. This is

particularly important in production Kubernetes environments where kernel modifications are typically not permitted.

- **Selective data aggregation**: BPF maps allow monitoring programs to aggregate statistics in kernel space and export only summarized results to userspace, reducing the volume of data that crosses the kernel-userspace boundary. This is in contrast to raw packet capture, where every packet must be copied to userspace for analysis [8].

These properties enable a monitoring approach that operates continuously in production without measurable impact on the workload being observed, a requirement for monitoring latency-sensitive 5G signaling traffic.

2.3 Network Monitoring Approaches

Network monitoring in cloud-native environments can be broadly categorized into passive approaches, which observe existing traffic, and active approaches, which inject traffic to test connectivity. This section reviews both categories and identifies the gap that motivates the hybrid approach presented in this thesis.

2.3.1 Passive Monitoring

Passive monitoring captures and analyzes traffic that is already flowing through the network, without generating additional load. The approaches differ in the detail and overhead of the data they collect.

Packet capture tools such as `tcpdump` and Wireshark operate by copying every packet that matches a filter expression from the kernel's network stack to a userspace buffer, where it can be written to disk or analyzed in real time. While this provides complete visibility into individual packets, the approach incurs significant overhead at high traffic rates: each captured packet requires a memory copy from kernel to userspace. In a Kubernetes cluster running a large number of pods, enabling full packet capture on every node would consume substantial CPU and storage resources, making it impractical for continuous production monitoring.

Flow-based monitoring protocols such as NetFlow [23] and sFlow [24] address the scalability problem by aggregating traffic into flow records: summaries that capture source and destination addresses, ports, protocol, packet counts, and byte counts for each conversation. This reduces the data volume by orders of magnitude compared to packet capture. However, flow records discard per-packet detail: individual RTT measurements, TCP flag sequences, and retransmission events are not preserved. Furthermore, sFlow uses statistical sampling (typically 1 in N packets), which provides accurate aggregate statistics but cannot reliably detect short-lived anomalies affecting individual connections [24].

eBPF-based observability tools represent a more recent approach. Cilium's Hubble component uses eBPF programs attached to the kernel's networking stack to extract per-flow metadata including HTTP request paths, DNS queries, and TCP

connection states without packet capture overhead or application instrumentation [8]. Pixie similarly uses eBPF to capture application-level telemetry automatically. These tools provide rich observability data, but their primary focus is on visualizing service dependencies and debugging application behavior rather than on detecting and correlating network-level faults across a cluster [20]. They do not perform active connectivity testing, and they lack correlation engines that can attribute anomalies to specific infrastructure components such as worker nodes or inter-node links.

2.3.2 Active Monitoring

Active monitoring generates synthetic traffic to test whether network paths are functional and to measure their performance characteristics. Unlike passive monitoring, it can detect failures even when no application traffic is flowing.

Synthetic probing tools such as the Prometheus Blackbox Exporter [25] send HTTP, TCP, ICMP, or DNS probes to target endpoints and record whether the probe succeeded, the response latency, and TLS certificate validity. This provides an external perspective on service availability and is widely used for uptime monitoring. However, synthetic probes test connectivity from the prober's location to the target, which may differ from the actual communication paths between pods. A probe from an external monitoring node may succeed even when pod-to-pod communication within the cluster is degraded, because the probe traverses a different network path. Additionally, probe-based tools typically operate at intervals of seconds to minutes, making them unable to detect sub-second latency spikes or transient failures.

Service mesh health checks, as implemented by Istio and Linkerd, monitor inter-service communication through sidecar proxies that intercept all traffic entering and leaving each pod. The proxy can report request success rates, latency distributions, and retry counts at the application protocol level (HTTP, gRPC). However, because the sidecar operates at the application layer (L7), it cannot observe kernel-level network behavior such as TCP retransmissions, RST rates, or SYN handshake timing. A network issue that manifests as increased kernel-level retransmissions may appear to the sidecar only as slightly elevated latency, without any indication of the underlying cause. Service meshes also introduce resource overhead: each sidecar proxy consumes additional CPU and memory, and the extra network hop adds latency to every request [26].

2.3.3 The Case for Hybrid Monitoring

Each of the approaches described above provides partial visibility into the network health of a Kubernetes cluster. Passive monitoring observes what is happening but cannot test paths that carry no traffic. Active monitoring tests connectivity but generates only point-in-time measurements without continuous traffic analysis. Application-layer tools miss kernel-level symptoms, while kernel-level tools lack the active probing needed to detect failures on idle paths.

A hybrid approach that combines kernel-level passive observation with active probing can address these limitations. By running eBPF programs that continuously monitor

TCP behavior (retransmissions, RST rates, RTT variations) alongside active TCP probes that test connectivity to every peer, such a system can detect both visible degradation and silent path failures. Abranches et al. demonstrated that eBPF-based monitoring can achieve continuous network analytics with significantly lower resource overhead than traditional approaches [27], but their work focused on single-node packet processing rather than cluster-wide fault detection.

To our knowledge, no existing tool combines eBPF-based passive traffic analysis with active inter-pod probing and a central correlation engine that attributes anomalies to specific pods, worker nodes, or inter-node links. This gap, the absence of a cluster-wide, topology-aware, hybrid monitoring system for cloud-native 5G deployments is what the system presented in this thesis aims to fill.

2.4 Anomaly Detection in Distributed Systems

Detecting anomalous behavior in distributed systems is a prerequisite for timely fault identification. In a Kubernetes cluster running dozens of interconnected pods, anomalies may manifest as latency spikes, packet loss, connection resets, or traffic pattern shifts. These symptoms can indicate anything from a failing pod to a degraded inter-node link. Chandola et al. [28] provide a comprehensive taxonomy of anomaly detection techniques while also categorizing them. This section reviews three categories relevant to network monitoring: statistical methods, machine learning approaches, and rule-based detection.

2.4.1 Statistical Methods

Statistical anomaly detection methods model the expected behavior of a metric and flag observations that deviate significantly from that model. Their appeal lies in a solid theoretical foundation and the ability to operate without labeled training data.

The *Z-score* method computes how many standard deviations an observed value lies from the sample mean. Given a time series of measurements with mean μ and standard deviation σ, an observation x is flagged as anomalous if $|x - \mu|/\sigma$ exceeds a threshold, typically between 2 and 3 [29]. While straightforward, the Z-score assumes a stationary distribution, an assumption that rarely holds in production systems where baseline metrics drift over time due to scaling events, traffic pattern changes, or rolling deployments.

The *exponentially weighted moving average* (EWMA) addresses the issue of changing baselines by weighting recent observations more heavily than older ones. The EWMA of a time series is defined recursively as $S_t = \alpha \cdot x_t + (1 - \alpha) \cdot S_{t-1}$, where $\alpha \in (0, 1]$ controls how quickly the average adapts to new data [30]. Anomalies are detected when the current observation deviates from the EWMA by more than a configurable multiple of the estimated standard deviation. EWMA is widely used in network monitoring because it is computationally inexpensive, requires no batch retraining, and naturally adapts to gradual baseline shifts [28].

2.4.2 Machine Learning Approaches

Machine learning methods learn complex patterns from data and can detect anomalies that are difficult to capture with fixed statistical models. Two unsupervised approaches are particularly relevant to network monitoring, as they do not require labeled examples of faults.

Isolation Forest, proposed by Liu et al. [31], detects anomalies by exploiting the observation that outliers are "few and different" and therefore easier to isolate through random partitioning. The algorithm constructs an ensemble of random binary trees, where anomalous points tend to be isolated in fewer splits, resulting in shorter average path lengths. Isolation Forest has linear time complexity and low memory requirements, making it applicable to monitoring data. However, it treats each observation independently and does not capture how metrics evolve over time.

Autoencoders are neural networks that learn to compress and reconstruct their input. When trained on normal traffic, they struggle to reconstruct patterns they have not seen before, making the reconstruction error a signal for anomalous behavior. This allows them to capture complex relationships between metrics such as RTT, retransmission rate, and packet rate. However, autoencoders require substantial training data representative of normal operation, are sensitive to hyperparameter choices, and introduce computational overhead that may be unsuitable for real-time monitoring [28].

2.4.3 Rule-Based Detection

Rule-based (or threshold-based) detection is the simplest and most widely deployed approach in production monitoring systems. A rule defines a condition on one or more metrics such as "RTT exceeds 5 ms" or "probe failure count exceeds 3 within 30 seconds" and triggers an alert when the condition is satisfied. Rules can be static, using fixed thresholds determined by domain expertise, or adaptive, where thresholds are derived from running statistics such as the EWMA described above.

The primary advantage of rule-based detection is *readability*: each alert directly maps to a human-understandable condition, making it straightforward to diagnose why an alert fired and to tune thresholds based on operational experience. Rules also have minimal computational overhead, requiring only a single comparison per metric per evaluation cycle, which makes them suitable for high-frequency, per-peer evaluation in systems that monitor many concurrent connections. Furthermore, rule-based systems require no training phase and can be deployed immediately with domain-informed defaults.

The limitation is expressiveness: fixed thresholds cannot adapt to workload changes, and individual rules cannot capture complex multi-metric correlations. A hybrid approach that combines adaptive statistical baselines (e.g., EWMA) with threshold-based alerting can mitigate these limitations while preserving readability [28].

2.4.4 Correlation and Attribution

Detecting individual anomalies is necessary but not sufficient for fault diagnosis in a distributed system. When a pod fails, multiple monitoring agents may independently report probe failures, latency spikes, and connection resets targeting the same peer. Without correlation, an operator is presented with a flood of individual alerts rather than a single diagnosis. Huang et al. [16] describe this challenge in the context of gray failures, where partial degradation produces ambiguous symptoms across multiple observers.

Correlation engines aggregate anomaly reports across the cluster and apply attribution logic to identify the root cause. Common patterns include: convergence-based attribution, where multiple independent reporters flagging the same target suggests the target is the source of the problem; and topology-aware correlation, where failures concentrated on peers located on a specific worker node suggest a node-level or link-level fault rather than individual pod failures [18].

2.5 Related Work

The monitoring approaches reviewed in Section 2.3 establish that passive observation, active probing, and application-layer monitoring each provide some visibility into cluster health. This section examines research efforts that are closest to the hybrid approach presented in this thesis and identifies the remaining gap.

2.5.1 eBPF-Based Monitoring for 5G

Several recent works investigate eBPF as a monitoring mechanism specifically for cloud-native 5G systems. Soldani et al. [6] position eBPF as a key enabling technology for observability and performance monitoring in cloud-native 5G, highlighting its flexibility, safety model, and low runtime overhead. Their work is a survey and vision paper that identifies eBPF's potential but does not present a concrete monitoring system.

The 5GC-Observer project [32] goes further by demonstrating a working system that uses eBPF to monitor communication between 5G core network functions at the kernel level. 5GC-Observer provides per-flow visibility into HTTP/2 exchanges over the service-based interface, validating that kernel-level observation of 5G signaling traffic is feasible without application instrumentation. However, 5GC-Observer focuses on passive observation of a single node's traffic. It does not combine passive observation with active probing to cover idle paths, and it does not correlate observations across multiple nodes to distinguish pod-level faults from infrastructure-level problems.

Abranches et al. [27] demonstrate that eBPF-based monitoring can achieve continuous network analytics with significantly lower resource overhead than traditional approaches such as packet capture or userspace proxies. Their work validates the performance characteristics of eBPF for network monitoring but focuses on single-node

packet processing pipelines. The step from per-node observation to cluster-wide correlation, aggregating anomaly reports from multiple agents and attributing them to specific infrastructure components, is not addressed.

These works collectively establish that eBPF is a viable and efficient mechanism for kernel-level network monitoring in 5G environments. What they do not address is how to combine this passive visibility with active probing and cross-node correlation into a single deployable system.

2.5.2 Cluster Monitoring and Distributed Telemetry

The architectural challenge of monitoring a cluster of interconnected nodes, collecting telemetry without introducing bottlenecks, and correlating observations across vantage points has been studied in contexts that predate cloud-native systems.

Supermon [33] is an early cluster monitoring system that demonstrated high-frequency, low-overhead sampling at scale. Its architecture of lightweight per-node agents feeding a central aggregator is directly relevant to NetMon's design. The key differences are that Supermon focused on system-level metrics (CPU, memory, network counters) rather than per-connection transport behavior, and it did not perform anomaly detection or fault correlation. It also predates containers and Kubernetes, so it does not address the dynamic topology of pod-based deployments where endpoints appear and disappear as pods are scheduled and rescheduled.

Research on monitoring for 5G network slices [34] emphasizes the need for lightweight, distributed telemetry that avoids central bottlenecks. In a sliced deployment, each network slice may have different performance requirements and must be monitored independently, multiplying the volume of telemetry data. This motivates architectures where per-node agents perform local aggregation and export only summarized results, a principle that NetMon follows: the eBPF programs and probe logic run within each pod, and only summarized anomaly reports and health metrics are sent to the central aggregator.

Bergström and Fredriksson [5] investigate active probing for detecting network partitions in distributed systems. The work demonstrates that probing can reliably identify partitions but highlights the scalability challenge: in a fully connected mesh of n nodes, the number of probe pairs grows as $O(n^2)$. This scalability concern is relevant to NetMon's design, though in practice the probe traffic is minimal (a single short-lived TCP connection per peer per 10-second round).

2.5.3 Gray Failure Detection in Practice

The concept of gray failures, introduced by Huang et al. [16], describes a class of degradations where a system component continues to operate but delivers degraded service: elevated latency, intermittent packet loss, or reduced throughput that does not trigger existing failure detectors. The authors argue that traditional fail-stop detectors, which model components as either alive or dead, are fundamentally unable to capture this intermediate state. To address this, they propose a detection archi-

tecture based on *differential observability*: comparing a component's self-reported health against the observations of its peers. When a component claims to be healthy but multiple peers report degraded interactions with it, the discrepancy signals a gray failure. This principle of multi-perspective observation is conceptually close to NetMon's design, where each agent independently monitors its peers and the central correlation engine compares these perspectives to identify components that are reachable but degraded. However, Huang et al. present differential observability as an architectural principle rather than a deployable system, and their work does not address the specific constraints of containerized environments such as dynamic pod topology, kernel-level metric collection, or integration with Kubernetes orchestration.

Alquraan et al. [18] provide grounding for the gray failure problem through an analysis of 136 real-world network-partitioning failures across distributed systems. Their findings are directly relevant: the majority of partitions they studied were *partial*, affecting some communication paths while leaving others intact, and most were not detected by the affected systems' built-in failure detectors. The study identifies a recurring pattern where individual nodes observe symptoms (timeouts, retries) but lack the cluster-wide view needed to determine whether the problem is local or systemic. This is precisely the attribution challenge that NetMon's correlation engine addresses, distinguishing between a single pod being slow and an entire inter-node link being degraded. However, Alquraan et al.'s contribution is a study of past failures, not a detection system. They characterize what goes wrong and why existing detectors miss it, but do not propose a concrete monitoring architecture.

In Kubernetes environments specifically, the closest equivalent to a gray failure detector is the built-in liveness and readiness probe mechanism [17]. Liveness probes determine whether a container should be restarted, and readiness probes determine whether a pod should receive traffic. Both operate on a binary pass/fail model: the probe either succeeds within a configured timeout or it does not. A pod experiencing a threefold increase in network latency due to a degraded inter-node link will continue to pass its liveness probe, which typically checks whether the application responds to an HTTP request within a generous timeout, while delivering measurably degraded service to its peers. Kubernetes provides no mechanism for expressing or acting on health information such as "this pod's average RTT to its peers has increased from 0.5 ms to 4 ms." This binary model is sufficient for detecting fail-stop failures but leaves gray failures entirely unaddressed.

The gray failure literature thus establishes both the problem and the detection principle, but a gap remains between the theoretical framework and a practical, deployable system. The problem has been characterized (Huang et al.), its prevalence in production has been documented (Alquraan et al.), and the insufficiency of existing Kubernetes health checks for this class of failure is evident from the platform's design. What is missing is a concrete system that implements observability at the kernel level, operates continuously with low overhead, and integrates with the dynamic topology of a Kubernetes cluster. This is one of the central contributions of the system presented in this thesis.

2.5.4 Summary and Research Gap

Table 2.1 summarizes the capabilities of the monitoring approaches reviewed in this chapter. Each existing approach provides partial visibility into the network health of a Kubernetes cluster, but none combines kernel-level passive observation, active probing of idle paths, and topology-aware correlation in a single system.

Approach	Granularity	Overhead	Idle paths	Kernel	Correlation
Packet capture	Per-packet	High	No	Yes	No
Flow-based	Per-flow	Low	No	Partial	No
eBPF observability	Per-flow	Low	No	Yes	No
Synthetic probing	Per-probe	Low	Yes	No	No
Service mesh	Per-request	Medium	No	No	Partial
Hybrid (this thesis)	**Per-packet + probe**	**Low**	**Yes**	**Yes**	**Yes**

Table 2.1: Comparison of network monitoring approaches for Kubernetes environments.

To our knowledge, no existing tool or research system combines the following three capabilities in a single deployable system:

1. **Kernel-level passive observation** of TCP transport behavior, including retransmission rates, RST counts, connection timing, and traffic volume, using eBPF programs that operate continuously with minimal overhead.
2. **Active inter-pod probing** that tests connectivity and measures round-trip time to every peer in the cluster, including paths that carry no application traffic.
3. **Central correlation with topology awareness** that aggregates anomaly reports from all agents and applies attribution logic to distinguish between pod-level failures, worker-node-level degradation, and inter-node link problems.

This gap, the absence of a lightweight, cluster-wide, topology-aware hybrid monitoring system for cloud-native 5G deployments, is what the system presented in this thesis aims to fill. The following chapters describe the design, implementation, and evaluation of NetMon, a system that combines these three capabilities and is deployable as a Kubernetes sidecar container alongside existing 5G core network functions.

3

Methods

This chapter describes the design, implementation, and evaluation methodology of the network monitoring system developed for Kubernetes-based telecom workloads. The system consists of distributed agents running on each pod and a central aggregation server that performs cluster-wide analysis.

3.1 System Architecture

The monitoring system is developed on the basis of a principle where network health checking within a distributed environment requires the aggregation of several observation techniques. No single technique, whether active or passive, is capable of giving a comprehensive view of network health on its own. By using a combination of active and passive techniques and centralizing the aggregation of such information, the system is able to cross-match information obtained from several points of observation within the cluster. This allows it to make a distinction between pod-level issues, worker-level issues, and network partitions. The following sections will explore the two-tiered structure of the system, the goals and objectives of its design, and the techniques it uses.

3.1.1 Architecture overview

The monitoring system follows a distributed central server architecture, designed to provide real-time visibility into network health across a Kubernetes cluster running 5G packet core components. An agent (NetMon) and a central server are the two major components of the proposed system.

Agent (NetMon) is deployed as a sidecar process in every monitored pod. Each agent is responsible for actively probing all peer pods in the cluster to verify TCP connectivity, passively monitoring traffic on the pod's network interface using eBPF programs, performing local anomaly detection on the collected metrics, and periodically reporting its findings to the central server.

Central Server acts as the intelligence center which runs on the controller pod in the cluster. Its functions are as follows:

- Data Aggregation: Collects reports from all active NetMon agents.

- Correlation & Analysis: Analyzes metrics from various pods to determine overall patterns or root causes for network degradation.
- Baseline Learning: Maintains per-peer pair baselines using CUSUM change-point detection and tracks anomaly-to-failure predictive correlations.
- Topology Mapping: Provides a physical layout of the cluster by mapping pods to their respective worker nodes.
- Health Monitoring: Tracks the operational state of both individual pods and worker nodes.

Communication between the agent and the central server occurs through JSON over TCP, with each agent sending reports every 5 seconds containing metrics, anomalies, and traffic flow statistics. The agents also communicate with each other via a lightweight ping-pong protocol on a separate TCP port for active probing and topology discovery. These communications rely on Kubernetes service discovery, where a headless service provides a stable DNS name that resolves to the current set of pod IP addresses. This is necessary because when a pod is terminated and replaced, the new pod receives a different IP address [17]. Service discovery ensures that agents can always locate the central server and their peers without static configuration even as pods are added, removed or rescheduled across worker nodes.

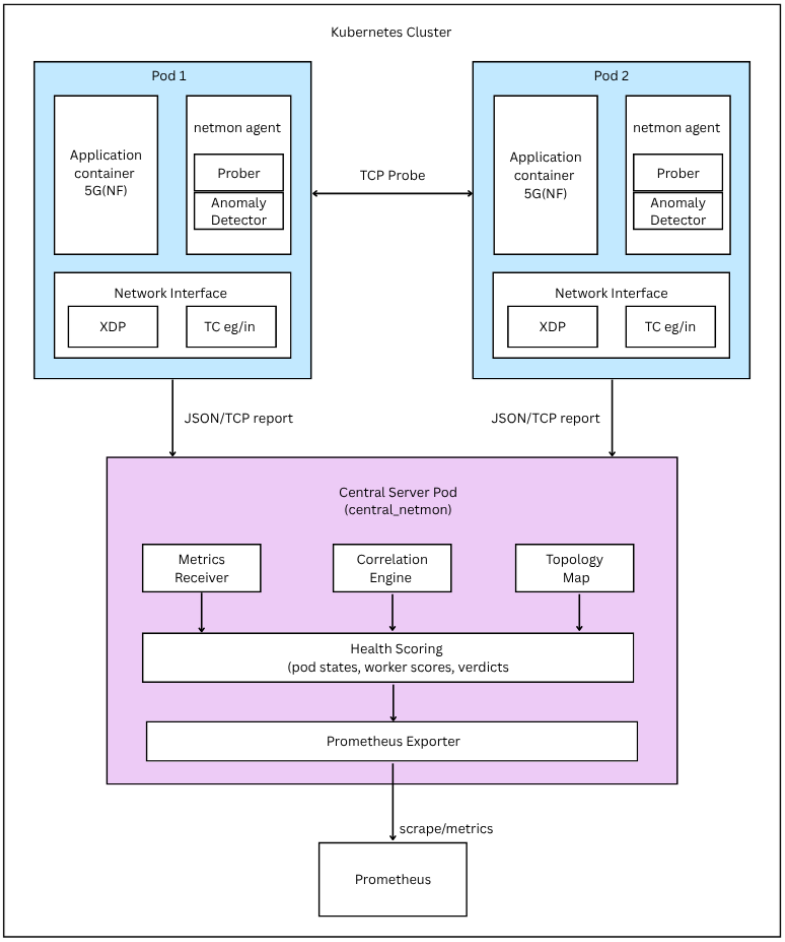

Figure 3.1: Architecture overview of Centralized network monitoring for 5G pods in Kubernetes.

3.1.2 Design Goals

The system is designed around four primary goals. The first is to enable early detection where the system should identify network degradation within seconds, before application-level health checks or scrape-based monitoring would surface the problem. This directly addresses the "gray failure" problem identified by Huang et al. [16], where partial degradation goes undetected by binary health checks. The second goal is to design the system to operate in a lightweight manner where the monitoring overhead in terms of CPU, memory, and network traffic must remain low enough so that it does not degrade the performance of the monitored workload. The third goal is to have cluster-wide correlation, meaning it should be able to aggregate multiple pod-level observations and distinguish between pod-level issues, worker node failures, and network partitions, rather than treating each issue as a separate entity. Lastly, to enable monitoring without modifying the source code of the application, rebuilding the container images, or altering the application-level configurations. The system should be deployed through infrastructure-level changes, such as granting Linux capabilities in the deployment chart and creating a headless service for peer discovery, ensuring workloads run unmodified while monitoring is

introduced transparently.

3.1.3 Active Probing Mechanism

The agent actively runs continuous TCP health checks for all peer pods in the cluster. The probing mechanism is composed of three steps:

Peer Discovery: Agents resolve the Kubernetes headless service name to get the IP addresses for all pods in the same namespace [15]. The list of peers is updated at the start of every probe cycle to handle pods being added or removed due to scaling or restarts [17]. In Kubernetes, when a pod is terminated, it first transitions the pod through a terminating state during which it may still be resolved but is no longer responsive. During this window, probe failures to the terminating pod are expected and are handled by the agent's failure tracking mechanism.

Probe Protocol: The agent sends a TCP request to a dedicated monitoring port on each target pod every 5 seconds. A 3 second connect timeout is used to bound the time spent on unresponsive peers, ensuring that a single unreachable pod does not delay the entire probe cycle. The probe is a simple ping-pong protocol in which both the sender and receiver include their own worker node name in the message. This exchange verifies TCP connectivity to the target and it allows agents to build a topology map by learning which worker node each pod is running on.

Failure Tracking: For every pod in the cluster, the agent tracks the number of total probes and failures as well as consecutive failures. Each failure is immediately flagged as a `PROBE_FAILURE` anomaly. When consecutive failures exceed a certain threshold (default is 3), an additional `CONSECUTIVE_FAILURES` anomaly is flagged to differentiate between connection failures and overall unavailability.

Peer Grace Period: When a pod disappears from DNS resolution (indicating termination), the agent does not immediately stop probing its last-known IP address. Instead, it continues probing for 60 seconds after the IP disappears from DNS. This ensures that terminated pods are detected via probe failures rather than silently dropping out of the monitoring set. Without this mechanism, a pod that shuts down between probe cycles would never generate a `PROBE_FAILURE` anomaly.

3.1.4 eBPF Passive Monitoring

In addition to active probing, the agent attaches eBPF programs to the primary network interface of the pod: one XDP program on the ingress path, and a pair of TC programs on the egress and ingress hooks. These programs enable passive monitoring of the traffic without injecting additional traffic or requiring changes to the application code.

3.1.4.1 XDP Program: Port and Traffic Monitoring

The XDP program runs on the ingress part of the network interface at the earliest point in the Linux networking stack, before the kernel even allocates a socket buffer

for the packet [22]. This means packets can be inspected with minimal overhead before any higher-level protocol processing takes place.

The program only looks at packets coming from known peer IPs and uses a set of shared BPF maps to track both per-peer behavior and aggregate traffic patterns. These maps feed into the local anomaly detection logic and provide data for cluster-wide traffic analysis with selected events forwarded for further processing. The following BPF maps are used:

- **Per-peer counters**: For each monitored peer, the program tracks TCP packets and byte counts, UDP packets, as well as counts of each TCP flag: SYN, SYN-ACK, ACK, PSH, FIN, RST. These counters are also maintained globally across all peers to provide cluster-wide traffic summaries.
- **Retransmission detection**: An LRU hash map of source IP, source port, TCP sequence number to the first seen timestamp stores information on each unique data-bearing TCP segment. Only segments carrying payload are tracked; like pure ACKs, SYNs, FINs, RSTs, keep-alives and window probes are excluded since they reuse sequence numbers. When a segment is seen again with the same key, it is considered a retransmission if at least 50 ms has elapsed since it was first seen. This 50 ms floor avoids false positives caused by the kernel's internal packet batching, which can cause the same segment to appear multiple times in rapid succession without being a true retransmission. An upper bound of 120 seconds filters out stale entries.
- **Event ring buffer**: A 64KiB ring buffer to stream events to userspace for real-time processing. Two types of events are emitted: retransmission events and RST events. In userspace, the agent uses RST events for burst detection that is, if 3 RSTs arrive from the same peer within 5 seconds then an immediate probe failure is recorded without waiting for the next probe cycle.

3.1.4.2 TC Program: Kernel-Level RTT Measurement

A pair of TC programs is attached to the egress and ingress hooks of the same interface. These two TC programs measure the active probe connection at the kernel level. Specifically, the TC programs are as follows:

- **TC Egress**: This TC program intercepts outgoing TCP SYN packets destined for the probe port. For each intercepted TCP SYN packet, it stores a high-resolution kernel timestamp in an LRU hash map with keys (peer IP, local IP, local port). If a timestamp already exists and is not stale (500 ms TTL), it preserves the existing timestamp, as it may be caused by retransmitted SYNs.
- **TC Ingress**: When a data-bearing reply segment is received from a peer on the probe port (i.e., the first segment carrying the "pong" response payload), it looks up the timestamp of the corresponding SYN packet and calculates the latency as the current kernel timestamp minus the stored SYN timestamp. The value, in microseconds, is sent to userspace via a ring buffer event. The timestamp entry is then deleted to prevent stale entry buildup.

This measurement captures the *handshake-to-response latency* of the probe connection, including TCP connection establishment and response generation at the receiver. However, because both timestamps are taken entirely within the kernel, it excludes userspace scheduling delays and instrumentation overhead on the sender side. As such, it provides a high-resolution and consistent signal that reflects network conditions together with minimal kernel and receiver-side processing.

Besides probe packets, the TC egress program maintains real application traffic flows. For every non-probe TCP packet, the TC egress program maintains per-flow counters (packets, bytes, last seen) in a separate LRU hash map keyed by the destination IP, destination port, and protocol. The counters are periodically read by the userspace agent and included in the JSON report sent to the central server which enables it to track real application traffic patterns between pods and detect traffic anomalies such as drops or spikes which is further discussed in Section 3.2.4.

3.1.5 Local Anomaly Detection

The agent includes a real-time anomaly detection capability for the gathered metrics. Two detection strategies are employed depending on the metric type: Z-score based deviation analysis for latency metrics, and relative/absolute threshold checks for traffic statistics.

3.1.5.1 Statistical Methods

The statistical approach is based on two components: The first is the Exponential Moving Average (EMA), defined as:

$$\mathrm{EMA}_t = (1 - \alpha) \cdot \mathrm{EMA}_{t-1} + \alpha \cdot x_t \tag{3.1}$$

where $\alpha = 0.2$ for RTT and $\alpha = 0.15$ for traffic statistics (packets per second, RST percentage, retransmission percentage). The EMA is more heavily influenced by the recent data points while also taking into account the previous observations [30]. This makes it appropriate for detecting changes in a non-stationary environment. The second is the Z-score, which calculates the number of standard deviations that a data point is away from the running average:

$$z = \frac{|x_t - \mathrm{EMA}_t|}{\sigma_t} \tag{3.2}$$

where σ_t is the running standard deviation, computed as the square root of an EMA of squared deviations using a slower smoothing factor ($\alpha_{\mathrm{var}} = 0.05$) to avoid hypersensitivity to individual outliers. A minimum standard deviation floor of 0.5 ms is enforced to prevent division by near-zero values in stable environments. If the Z-score exceeds 5.0, the data point is marked as anomalous. The statistical method is disabled until at least 20 data points are collected for the peer.

For RTT and jitter anomalies, an additional absolute deviation filter requires that the deviation from the EMA exceeds 2 ms before an anomaly is raised. This prevents spurious alerts when the baseline is very low, such as sub-millisecond RTT between pods on the same worker node.

For traffic statistics (RST percentage, retransmission percentage, packets per second), a relative threshold approach is used instead: a metric is flagged as anomalous if it exceeds 5× its per-peer EMA baseline. Additionally, absolute thresholds are applied for RST percentage ($> 5\%$) and retransmission percentage ($> 10\%$), which fire independently of the baseline.

3.1.5.2 Warmup Suppression

To avoid false positives during startup, the agent suppresses anomaly reporting during the first 3 probe rounds. This allows the EMA baselines to stabilize before detection begins.

3.1.5.3 Types of Anomalies

A summary of the different anomaly types identified by the agent can be found in Table 3.1. Each identified anomaly includes information on the peer IP address, type of anomaly, and a string describing the anomaly, which is then included in the next metrics report sent to the central server.

Table 3.1: Anomaly types detected by the agent.

Anomaly Type	**Trigger Condition**
`PROBE_FAILURE`	TCP connection to a peer failed (3-second timeout).
`CONSECUTIVE_FAILURES`	≥ 3 consecutive probe failures to the same peer.
`KERNEL_RTT_SPIKE`	Kernel-measured RTT deviates significantly from EMA baseline ($z > 5.0$ and absolute deviation $\geq 2\,\text{ms}$).
`JITTER_SPIKE`	Jitter (absolute difference between consecutive RTT samples) deviates from its EMA baseline ($z > 5.0$ and absolute deviation $\geq 2\,\text{ms}$).
`HIGH_RST`	RST packet percentage from a peer exceeds 5% (absolute threshold).
`HIGH_RETRANSMIT`	Retransmitted packet percentage exceeds 10% (absolute threshold).
`RST_SPIKE`	RST rate exceeds 5× the per-peer EMA baseline.
`RETX_SPIKE`	Retransmission rate exceeds 5× the per-peer EMA baseline.
`HIGH_PPS`	Packet rate from a peer exceeds 5× its EMA baseline.
`TRAFFIC_SILENCE`	No traffic observed from a previously active peer ($>$ 50 lifetime packets) for more than 30 s.

3.1.6 Metrics Reporting

Every 5 seconds, a metrics report is compiled by the agent and sent to the central server using TCP. The report is serialized to JSON format, which includes:

- **Pod identity**: IP address, name of worker node, pod role (e.g., 'forwarder', 'controller').
- **Local health summary**: Aggregate probe success rate, total/active peers, cumulative probe/failure counts.
- **Per-peer metrics**: Kernel RTT, RTT EMA, jitter, packets per second, throughput, RST percentage, retransmission percentage, cumulative probe/failure counts.
- **Anomalies**: All detected anomalies since the last report, including type, peer IP, description.
- **Traffic flows**: Application traffic flows observed by the TC egress program including destination IP, port, protocol, packet/byte counts.

A push-based reporting model was chosen rather than a pull-based one (e.g. Prometheus scraping) to minimize detection latency and ease deployment. Reports are delivered to the central server within seconds of collection rather than waiting for the next scrape interval, and agents only need to look up the service name of the central server rather than requiring scrape configuration.

3.2 Central Server Design and Implementation

The central server runs on a designated controller pod, and by doing so it uses existing leader election to avoid the complexity of implementing this solely for the central server. This gives the system a single point for aggregating and analyzing cluster-wide data, with each agent periodically sending its JSON report to the server over TCP. In clusters where no dedicated controller pod exists, this design can be adapted by adding a leader election mechanism among the pods.

3.2.1 Pod State Management

The central server manages the state for each reporting pod and stores the following information: A record with lifetime metrics (total probes, failures, and anomalies), the current peer states, and a health state classification. The health states are determined by counting probe failures and total anomalies *targeting* the pod within a 30-second sliding window. The states are defined as follows:

- **DOWN**: No report received for more than 60 seconds.
- **INSUFFICIENT**: Fewer than 50 total probes collected; the pod has not been monitored long enough for a stable baseline.
- **FAILING**: At least 9 probe failures targeting the pod within the last 30 seconds.

- **WARNING**: At least 3 probe failures *or* 30 or more total anomalies targeting the pod within the last 30 seconds.
- **DEGRADED**: At least 1 probe failure *or* 20 or more total anomalies targeting the pod within the last 30 seconds.
- **HEALTHY**: None of the above conditions are met.

Another important design choice is that the health state of a pod is not only based on what it reports, but also on what other pods report *about* it. The central server keeps counters for "anomalies about me" for each pod categorized by anomaly type. These counters are incremented when any agent reports an anomaly with that pod's IP as the source of the anomaly. This kind of cross-referencing helps detect problems with a pod that it cannot see for itself, for example, being unreachable from certain other parts of the cluster.

3.2.2 Correlation Engine

The correlation engine analyzes the aggregated anomaly history within a 30-second sliding window to identify cluster-wide failure patterns. Unless otherwise stated, thresholds are evaluated over distinct reporting sources (i.e., unique pods or workers) and are calibrated for a 10-pod cluster deployment.

1. **PEER_DOWN**: If ≥ 9 probe failure anomalies from distinct reporting pods target the same pod within the window, the pod is classified 'down'. In a 10-pod cluster, this corresponds to all other pods failing to reach the target pod.
2. **LATENCY_SOURCE**: If ≥ 15 RTT spike or jitter anomalies target the same pod, the pod is classified as a source of latency degradation.
3. **ANOMALY_HOTSPOT**: If ≥ 30 anomalies of any type target the same pod, and that pod has not already been classified as **PEER_DOWN** or **LATENCY_SOURCE**, the pod is classified as a hotspot.
4. **WORKER_ISOLATED**: When a pod experiences failures with ≥ 3 distinct peers on the same worker, this indicates a network partition between the worker where the failing pod resides and the target worker. This condition assumes that each worker hosts at least three pods.
5. **INTER_NODE_FAILURE**: When a worker-worker link experiences a success rate of less than 50% while the source worker still experiences success in reaching other workers, a failure in the inter-node link occurs.
6. **POD_ISOLATED**: When a pod experiences a success rate of less than 10% in reaching peers, while sibling pods on the same worker are still healthy, the pod itself is considered isolated, not the worker.
7. **WORKER_HOTSPOT**: When ≥ 2 external workers report anomalies about the pods on the same target worker and that worker's health score (defined in

Section 3.2.4) is ≥ 10 points below the cluster average, the worker is flagged as degraded.

8. **WORKER_HIGH_LATENCY**: When more than 50% of inter-worker links to a target worker have RTT exceeding 10 ms, and at least 3 such links exist, the worker is identified as a latency source.
9. **WORKER_SOURCE_HOTSPOT**: When a single worker reports ≥ 40 total anomalies, $\geq 2\times$ the cluster average anomaly count, about atleast 3 distinct other workers, the problem is localized to the reporter's egress path (e.g., NIC issue, egress packet loss).

The first three patterns identify pod-level issues, patterns 4–6 identify connectivity and isolation issues and patterns 7-9 identify worker-level degradation by cross-referencing reports from multiple vantage points.

3.2.3 Online Change-Point Detection (CUSUM)

In addition to the rule-based correlation engine, the central server implements an online change-point detection system for identifying distributional shifts in per-peer-pair metrics. For each source-destination pod pair, the system maintains running statistics for three metrics: RTT, jitter, and retransmission percentage.

Welford's Online Algorithm maintains a running mean and variance for each metric, updating incrementally with each new sample:

$$\bar{x}_n = \bar{x}_{n-1} + \frac{x_n - \bar{x}_{n-1}}{n} \tag{3.3}$$

$$\sigma_n^2 = \sigma_{n-1}^2 + \frac{(x_n - \bar{x}_{n-1})(x_n - \bar{x}_n) - \sigma_{n-1}^2}{n} \tag{3.4}$$

where x_n is the n-th observed sample, $\bar{x}_n$ is the running mean after n samples, and σ_n^2 is the running variance after n samples. This formulation is numerically stable and requires only constant memory per metric, making it suitable for long-running online computation.

CUSUM (Cumulative Sum) detects change-points by accumulating standardized deviations from the learned mean:

$$S_t^+ = \max(0,\ S_{t-1}^+ + z_t - k) \tag{3.5}$$

$$S_t^- = \min(0,\ S_{t-1}^- + z_t + k) \tag{3.6}$$

where $z_t = (x_t - \bar{x})/\sigma$ is the standardized value and $k = 0.5$ is the slack parameter that controls sensitivity to small shifts. A shift is detected when $S_t^+ > h$ or $S_t^- < -h$, where $h = 4.0$ is the decision threshold. Upon detection, the CUSUM accumulators are reset and the mean is updated to the current value, allowing the system to adapt to the new operating point.

A warmup period of 30 samples is required before detection activates. For RTT and jitter shifts, a minimum absolute deviation of 5 ms from the current mean is required

to suppress noise from sub-millisecond fluctuations. The system generates three anomaly types: `CUSUM_RTT_SHIFT`, `CUSUM_JITTER_SHIFT`, and `CUSUM_RETX_SHIFT`.

Baseline Seeding. For known cluster topologies, pre-computed worker-pair baselines can be loaded from a file to skip the warmup period for new pod pairs. These baselines are extracted from historical Prometheus data and contain the mean and variance for RTT, jitter, and retransmission percentage per worker-pair direction.

Anomaly-to-Failure Tracking. The system tracks whether CUSUM-detected shifts precede probe failures within a 60-second window. For each pod pair, it counts shifts that were followed by a failure (true predictions), shifts not followed by a failure (false alarms), and failures with no preceding shift. The predictive rate quantifies the system's ability to provide early warning before hard failures occur.

3.2.4 Topology Awareness

To construct a topology map of the cluster, the central server groups the pods based on the worker node assignments, which are obtained from the worker node name received during the probe handshake. There are three benefits of the topology map:

Worker Health Scoring: Each worker node is assigned a composite health score that combines two complementary signals: the probe success rate, which captures "hard" connectivity failures, and the anomaly rate, which captures "soft" performance degradation such as latency spikes, jitter, and elevated retransmissions. The health score is defined as:

$$\text{score} = \alpha \cdot \text{success\_rate} + (1 - \alpha) \cdot (1 - \text{anomaly\_rate}) \tag{3.7}$$

where *success_rate* is the fraction of successful TCP probes targeting the worker's pods within the last 30 seconds (computed as $1 - \text{recent\_failures/expected\_probes}$), *anomaly_rate* is the ratio of recent anomalies targeting the worker's pods to a normalized maximum of 30 anomalies per pod per 30-second window (clamped to $[0, 1]$), and $\alpha = 0.6$ is the weighting parameter.

For interpretability in subsequent analysis, the score is expressed on a 0–100 scale by multiplying the value in Equation 3.7 by 100. All thresholds and comparisons (e.g., deviations from cluster average) are defined in this scaled representation.

This weighting prioritizes availability, as a probe failure signifies a terminal loss of connectivity, while an anomaly represents merely degraded communication. The value $\alpha = 0.6$ was selected empirically; values below 0.5 were found to overreact to transient traffic bursts, while values above 0.7 were insufficiently sensitive to sustained issues like rising retransmission rates.

Network Partition Detection. The server maintains a connectivity matrix of worker to worker links, where each link aggregates the probe results between all pods on the respective workers. A worker is said to be partly partitioned if all its outgoing links have a success rate less than 50%, which means it cannot reliably reach any other worker in the cluster.

Traffic Baseline Tracking. Using the worker-to-worker grouping, the server tracks the volume of real application traffic between each pair of workers by maintaining an exponential moving average ($\alpha = 0.2$). After a minimum of 10 samples, the baseline is used to detect traffic anomalies including traffic drops (below 40% of baseline), traffic spikes (above 3 times baseline), and total traffic loss.

3.2.5 Reporting

The central server generates a report that summarizes cluster status every 30 seconds. It contains cluster-wide health metrics, anomaly correlations, anomaly-failure predictive rates, worker health scores, a list of problem pods, and detailed metrics for each pod grouped by worker node. By default, it outputs a compact one-line status summary every 30 seconds containing active/total pod count, worker health scores, and any pods with a health state below HEALTHY. When launched with the `-verbose` flag, it outputs a detailed report including full cluster health metrics, per-pod breakdowns grouped by worker node, anomaly correlations, and CUSUM insights. In addition, the server exposes a Prometheus metrics endpoint on port 9090 that serves metrics in the standard Prometheus exposition format. The exposed metrics include pod health states, probe success rates, per-peer RTT, jitter, packets per second, throughput, retransmission and RST percentages, worker health scores, inter-worker RTT averages, CUSUM baseline statistics, anomaly-to-failure prediction rates, and active correlations. This enables integration with Prometheus for time-series storage and visualization through dashboards. In the Ericsson test environment, these metrics are scraped and visualized in Ericsson's Cloud Native Operations Manager (CNOM), providing operational visibility within the 5G core management infrastructure.

3.3 System Implementation

The system is implemented in C for performance and for direct integration with the `libbpf` library, which provides the C API for loading and interacting with eBPF programs. JSON serialization is done manually using formatted string output. The eBPF programs are compiled with Clang using the BPF target and loaded through the `libbpf` skeleton mechanism, which generates type-safe C headers from compiled BPF object files. This bypasses runtime compilation and ensures that BPF map layouts stay consistent between kernel and userspace. The build system uses GNU Make with a container-targeted build configuration.

3.3.1 Deployment

The system is deployed to a Kubernetes environment using a deployment script that handles binary deployment and process management. The central server binary is first deployed to the controller pod and started. After a short initialization period, agent binaries are deployed to all monitored pods and started. Each agent uses DNS to discover the IP addresses of all other agents and begins probing and monitoring immediately.

The eBPF programs require the Linux capabilities `CAP_BPF` and `CAP_NET_ADMIN` to be granted to the container [9]. This is a significant deployment constraint, since many production Kubernetes clusters restrict elevated capabilities by default. In practice, this means the tool may not be deployable without explicit security policy changes, limiting its use in hardened environments.

3.3.2 Design Trade-offs

This section discusses the key design decisions and their trade-offs.

eBPF over packet capture: Running eBPF programs in-kernel avoids copying packets to userspace and benefits from verifier-enforced safety, resulting in lower overhead. The downside is that it requires Linux 5.8 or later and the elevated capabilities mentioned above.

Active and passive monitoring: Combining active probing with passive eBPF observation means the system can confirm failures immediately, rather than waiting for passive data alone. The RST burst detection mechanism further bridges passive observation to active failure detection. The cost is that probe traffic grows as $O(n^2)$ with the number of pods. In practice, probes are infrequent (every 5 seconds) and lightweight, so this is manageable at the current cluster size.

Push-based metrics: Agents push reports to the central server rather than being scraped, which keeps detection latency low and avoids the need for scrape configuration. The downside is that the central server becomes a single point of failure. In the current deployment, this is mitigated by running the server on the Ericsson controller pod, which provides automatic failover. In a general cluster without a dedicated controller, a leader election mechanism among the agent pods would be needed.

Dual-layer detection: The system employs two complementary detection layers: the agent-side Z-score detector provides immediate per-sample anomaly detection, while the central-side CUSUM detector (described in Section 3.2.3) identifies distributional shifts over time. The Z-score approach is fast and interpretable but can generate false positives during normal variance. CUSUM is more robust to noise due to its cumulative accumulation and slack parameter, but requires a warmup period and is slower to react to sudden step changes. Together, they provide both immediate alerting and trend-aware detection.

Sensitivity over precision: The system is designed to favor sensitivity: missing a real failure is more costly than raising a false alarm in a network monitoring context. The correlation engine mitigates the resulting false positive volume by requiring multiple independent observations before escalating to a cluster-level diagnosis. Individual anomalies are frequent (approximately 2900 per hour during normal operation), and supplementary correlations such as `WORKER_HOTSPOT` also fire due to natural variance. However, the connectivity-loss correlations (`PEER_DOWN`, `WORKER_ISOLATED`) maintained zero activations across 18 hours of baseline monitoring, providing unambiguous signals when they do fire.

Statistical methods over machine learning: Machine learning approaches for anomaly detection were considered during the design phase but ultimately not pursued due to time constraints and the lack of labeled training data in the target environment. The CUSUM change-point detection and Z-score methods were chosen instead as they require no training phase, adapt online to the deployment's baseline, and produce interpretable results. These statistical methods achieve the core goal of detecting distributional shifts without the overhead of model training, hyperparameter tuning, or the risk of overfitting to a specific cluster configuration.

Parameter Selection: The detection parameters were determined through iterative experimentation on the live test cluster. Initial values were chosen from standard recommendations (e.g., z-score threshold of 3.0, CUSUM slack $k = 0.5$) and then adjusted based on observed behavior. The z-score threshold was raised from 3.0 to 5.0 because the lower value produced excessive anomaly volume from normal RTT variance, while 5.0 reliably detected the smallest target fault (10 ms added latency) without overwhelming the correlation engine. Similarly, the absolute deviation floor of 2 ms, the EMA smoothing factors, and the correlation thresholds (e.g., $\geq$9 probe failures for `PEER_DOWN`) were tuned to match the cluster size and baseline characteristics of the test environment. These values encode assumptions about the monitored system's normal behavior and would need to be re-tuned for deployments with different network characteristics or cluster sizes.

3.4 Evaluation Methodology

In this section, the experimental setup and evaluation approach that is adopted to test and evaluate this monitoring system is described. It is to be noted that the behavior of eBPF-based programs and the anomaly detection mechanism is heavily dependent on the actual network environment, which cannot be properly emulated through unit-level testing. Hence, all experiments are conducted on a live Kubernetes test cluster. Controlled fault injection are used to simulate connectivity and performance issues.

3.4.1 Evaluation Environment

The test environment consists of a virtual AMF component of a 5G core network deployed on a Kubernetes-based infrastructure. This infrastructure has a total of four worker nodes. A total of ten pods are part of a workload, which consists of four different types of microservices representing key AMF functionality: control plane, transport, forwarding, and mobility management. Each worker node usually runs two to three pods.

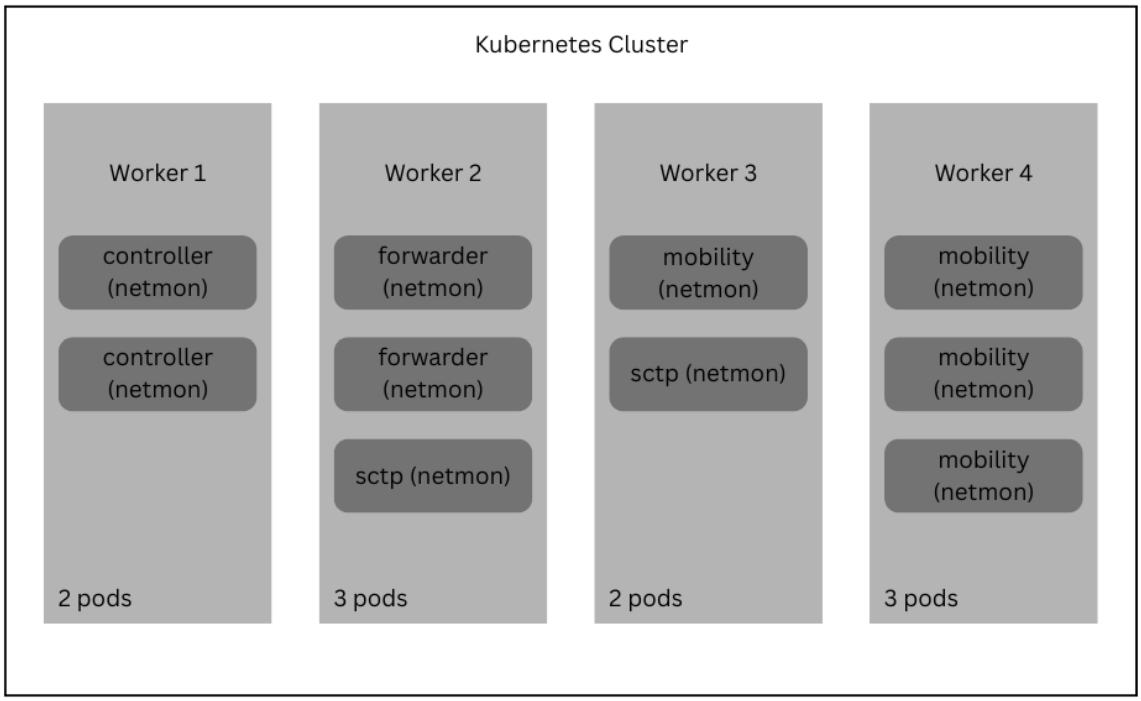

Figure 3.2: Cluster topology with 10 pods across 4 worker nodes.

3.4.2 Fault Injection Scenarios

To evaluate detection and correlation capabilities, controlled fault scenarios were injected into the live test cluster. The scenarios cover the spectrum of network failure modes from subtle performance degradation to complete connectivity loss. Each scenario was repeated multiple times per configuration (a minimum of 5 runs, with up to 10 runs for selected configurations), and experiments were conducted at three traffic load levels (0%, 20%, and 50% simulated UE traffic), where 20% and 50% correspond to approximately 20 and 50 concurrently active UE sessions generating representative AMF control-plane traffic to assess robustness under realistic operating conditions.

All network faults were injected on a single target worker node using Linux Traffic Control (`tc`) and `iptables`, while the central server ran on a separate worker node to ensure uninterrupted metric collection from both sides of a fault. Table 3.2 summarizes the fault scenarios and their configurations.

Table 3.2: Fault injection scenarios and configurations.

Scenario	Injection Method	Configurations	Target Detection
Latency injection	`tc netem delay`	10, 25, 50, 100 ms at 0/20/50% load	`KERNEL_RTT_SPIKE`, `CUSUM_RTT_SHIFT`, `WORKER_HIGH_LATENCY`
Jitter injection	`tc netem delay 5ms 20ms distribution normal`	0/20/50% load	`JITTER_SPIKE`, `CUSUM_RTT_SHIFT`, `WORKER_HIGH_LATENCY`
Packet loss	`tc netem loss`	5%, 10%, 20%, 50% at 0/20/50% load	`RETX_SPIKE`, `CUSUM_RETX_SHIFT`, `WORKER_SOURCE_HOTSPOT`
Bandwidth throttle	`tc tbf rate`	1, 5 Mbit/s at 0% load	`CUSUM_RETX_SHIFT`, `WORKER_SOURCE_HOTSPOT`, `WORKER_HIGH_LATENCY`
Network partition	`iptables` FORWARD	Worker-2 fully isolated at 0% load	`PROBE_FAILURE`, `PEER_DOWN`, `WORKER_ISOLATED`
Worker-to-worker link failure	`iptables` FORWARD	Worker-2 $\leftrightarrow$ worker-3 at 0% load	`PROBE_FAILURE`, `PEER_DOWN`, `WORKER_ISOLATED`
Pod termination	`kubectl delete pod`	Single pod at 0% load	`PROBE_FAILURE`, `PEER_DOWN`
Gradual degradation	`tc netem delay` (stepped)	1, 5 ms steps every 30 s at 0/20/50% load	`KERNEL_RTT_SPIKE`, `CUSUM_RTT_SHIFT`, `WORKER_HIGH_LATENCY`

Each injection lasted 3 minutes followed by a 2-minute recovery period. A false positive baseline was also recorded at each load level with no fault injection to establish the background anomaly rate.

3.4.3 Evaluation Metrics

The evaluation is based on three metrics: detection latency, correlation latency, and resource overhead.

3.4.3.1 Detection Latency

Detection latency is defined as the time between injecting a fault and the first relevant anomaly appearing at the central server that involves the affected worker or its pods. Only anomalies that specifically concern the injected fault target are counted, to avoid measuring coincidental baseline activity.

3.4.3.2 Correlation Latency

Correlation latency is defined as the time between injecting a fault and the first worker-level correlation that correctly localizes the fault to the affected infrastructure component. This measures the end-to-end time from fault occurrence to actionable diagnosis. The relevant correlations depend on the fault type: `WORKER_HIGH_LATENCY` for latency and jitter faults, `WORKER_SOURCE_HOTSPOT` for packet loss and bandwidth throttling, and `PEER_DOWN` or `WORKER_ISOLATED` for network partitions and pod terminations.

3.4.3.3 Resource Overhead

Resource usage for both the agent and central server is measured:

- **CPU usage:** Measured at the container level using `kubectl top` and `bpftool` runtime statistics for eBPF programs.
- **Memory usage:** Measured using the Resident Set Size (RSS) of both the agent and the central server.
- **Network overhead:** Probe traffic ($N \times (N-1)$ connections per cycle) and JSON report traffic (N reports per cycle), quantified in terms of connection count and payload size.

4

Results

This chapter presents the evaluation results of the developed monitoring system conducted on a cloud-native 5G AMF deployment. First, the user experience of the monitoring system is described through its dashboard visualization capabilities and limitations. The resource overhead of the monitoring agents and central correlator is then presented in terms of CPU and memory consumption as well as the network traffic introduced by the probing mechanism. Next, the system's baseline behavior is characterized by measuring anomaly rates and correlation activity during normal operation without injected faults. The fault injection experiments then measure two key metrics: *detection latency* (time from fault injection to the first *relevant* anomaly fired by the system) and *correlation latency* (time from fault injection to the first worker-level correlation that correctly localizes the fault). The chapter concludes with a summary of the key findings across all experiments.

4.1 User Experience Results

The system's monitoring data is visualized through CNOM (Cloud Native Operation Manager), Ericsson's built-in dashboard platform for cloud-native deployments. A single overview dashboard was developed to present the system's detection and correlation outputs in real time. The dashboard contains 19 widgets organized to provide both high-level cluster status and detailed per-pod metrics:

- **Pods (Total / Active)**: a Key Performance Indicator (KPI) widget showing the number of registered and currently reporting pods.
- **Anomaly History**: a time-series graph of anomaly counts by type within the 30-second correlation window, showing the real-time rate of `KERNEL_RTT_SPIKE`, `JITTER_SPIKE`, `CUSUM_RTT_SHIFT`, and other anomaly types.
- **Anomaly History by Target**: anomalies grouped by target pod, showing which pods are being reported as problematic by their peers.
- **Active Correlations**: a timeline of currently active worker-level correlations (`WORKER_HIGH_LATENCY`, `WORKER_SOURCE_HOTSPOT`, `PEER_DOWN`, etc.), providing immediate visibility into localized faults.
- **Worker Health Score**: a 0-100 score per worker node combining probe success rate and anomaly pressure, providing a single metric for worker-level

health.

- **Pod Health State**: per-pod health state over time (HEALTHY, DEGRADED, WARNING, FAILING, DOWN), enabling operators to identify pods transitioning through degradation stages.
- **RTT by Worker Pair**: average RTT grouped by source and destination worker, enabling identification of degraded inter-node links.
- **Lifetime Anomalies**: a KPI widget of total anomalies reported and a timeline of total anomalies reported about each pod.
- **Monitored Pods**: a table listing all registered pods with their current IP, name, worker node, running state, ready state, restarts and the age of the pod.
- **Total Probes / Total Failures per Pod**: cumulative probe and failure counters for each pod, useful for identifying long-term reliability trends.
- **Avg RTT and RTT EMA by Pod**: instantaneous and smoothed round-trip time measurements, showing both current conditions and baseline trends.
- **Jitter, Packets/s, Throughput, RST%, Retransmit% by Pod**: per-pod traffic metrics captured by the eBPF programs, providing visibility into traffic patterns and TCP health indicators.

The dashboard can be seen in figure 4.1 and at the moment of this capture we are displaying the last 15 minutes of data on a 0% load cluster during a 20ms latency injection on worker-2. Notice the spike observed in the bottom right worker-to-worker latency widget which in turn generates spikes of latency related anomalies in the two top left widgets which are anomalies by type and target respectively. We can then observe that several correlations appear in the top right widget, specifically `LATENCY_SOURCE` for the affected pods as well as `WORKER_HIGH_LATENCY` and `WORKER_HOTSPOT` for worker-2, correctly identifying the faulty worker and pods. All of these signals together cause the worker health score and the pod health states to drop in the two bottom left widgets. Finally, we can observe the widgets all returning to baselines after the latency injection is cleared. A comprehensive list of figures for all the widgets currently implemented in CNOM can be found in Appendix A.

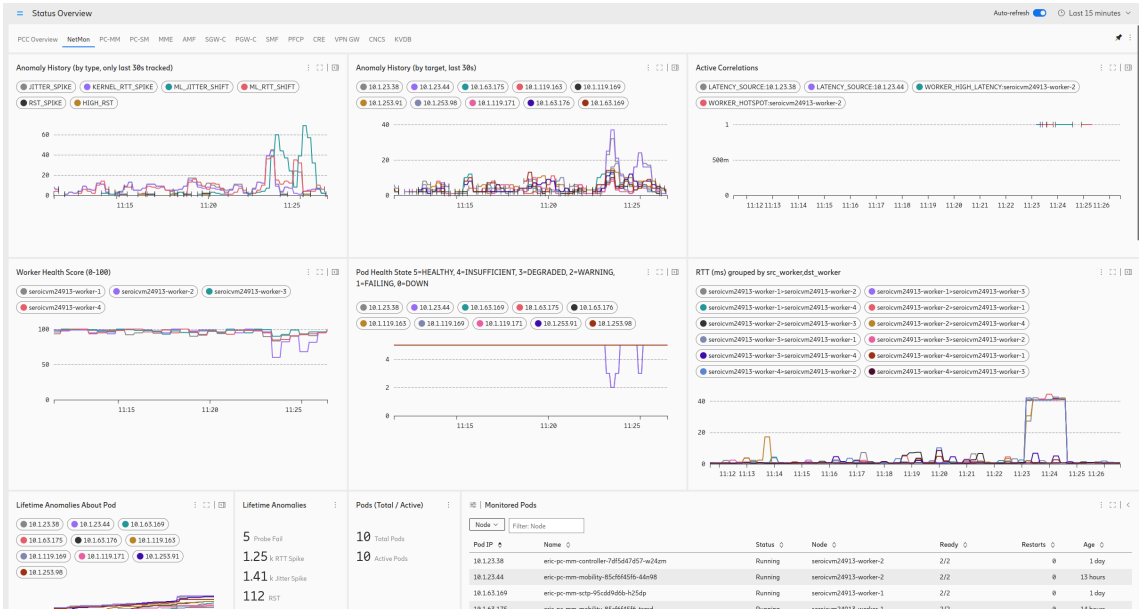

Figure 4.1: A screenshot of the CNOM dashboard during 20ms latency injection, showing the first 10 widgets.

During the evaluation, the dashboard confirmed the system's detection behavior in real time. When a fault was injected, the anomaly history graph showed an immediate spike in the relevant anomaly types, followed by the appearance of a correlation entry in the Active Correlations panel identifying the affected worker. This two-level presentation, raw anomalies for detailed investigation and correlations for actionable alerts, allows operators to quickly identify both the existence and location of a network issue without manually inspecting per-pod metrics.

However, several limitations of CNOM as a visualization platform were encountered during this work, including a 20-line maximum per time-series widget, limited widget types compared to general-purpose tools such as Grafana, and fixed refresh intervals constrained by the Prometheus scrape cycle. These are discussed further in Section 5.2.

Despite these constraints, CNOM provides sufficient visibility for an operator to monitor cluster health and respond to detected faults. The real value of the system lies in the detection and correlation engine rather than the visualization layer, which could be replaced by a more capable platform without modifying the underlying monitoring architecture.

4.2 CPU/Memory Usage and Traffic Overhead

The resource consumption of the monitoring system was measured across all tested conditions: idle operation, active fault injection, and application loads of 20% and 50%. The results are identical across all conditions, with neither application load nor fault injection producing any measurable change in resource usage.

Each agent's userspace process consumes 3 millicores (0.3% of a single CPU core) and 4.5 MiB of memory. The eBPF programs executing in kernel space were measured separately using `bpftool` runtime statistics. At 50% application load, the three eBPF programs (XDP packet filter, TC egress, TC ingress) collectively consume 0.4 millicores per pod while processing approximately 333 packets per second of inter-pod traffic. The total agent overhead is therefore approximately 3.4 millicores per pod: 3m for the userspace probe loop and metrics reporting, plus 0.4m for kernel-side packet processing.

The central correlator consumes less than 1 millicore and 2.6 MiB of memory. Over a 19-hour measurement period, the central process used only 24 seconds of total CPU time, confirming that its processing of incoming reports and correlation computation is negligible. The 4.5 MiB agent memory is dominated by the eBPF map allocations (per-peer counters, sequence tracking for retransmit detection, and ring buffers for event delivery), which are fixed-size regardless of traffic volume.

For a cluster of N monitored pods, the total resource overhead is approximately $N \times 3.4$m CPU and $N \times 4.5$ MiB memory for the agents, plus 0.35m CPU and 2.6 MiB for the central instance. In the evaluated deployment with 10 monitored pods, NetMon consumes 34 millicores of CPU and 48 MiB of memory cluster-wide. By comparison, the application containers in the same cluster consume 393–7048 millicores of CPU and 5259–10447 MiB of memory depending on traffic load (measured at 0% and 50% offered load respectively). This places NetMon's overhead at 0.5–8.7% of application CPU and 0.5–0.9% of application memory. Notably, NetMon's resource usage remains constant at 34m CPU and 45 MiB regardless of application load, as it generates its own lightweight probe traffic rather than scaling with the volume of real traffic.

Beyond CPU and memory, the monitoring system introduces network traffic overhead. Each probe cycle (every 5 seconds), every agent sends a short-lived TCP connection to each of its $N-1$ peers, producing $N \times (N-1)$ probe messages across the cluster. Each agent also sends a single JSON metrics report to the central correlator, adding N report messages per cycle. For the evaluated cluster ($N = 10$), this amounts to 90 probe connections and 10 reports every 5 seconds. Each probe consists of a TCP handshake followed by a brief exchange where the prober sends its worker node identity and the target replies with its own worker node identity (under 200 bytes total). Each report is a JSON document of approximately 4–5 KB containing per-peer metrics, anomaly records, and traffic flow summaries. At these volumes, the monitoring traffic is negligible relative to the application traffic generated by the AMF handling millions of UE connections.

In practice, cloud-native 5G deployments of the type evaluated in this thesis typically operate clusters of 150–200 pods per network function, which has been estimated to be within acceptable bounds for the system's design. For deployments that exceed this scale, techniques such as probe sampling or hierarchical aggregation could be introduced, though this is left as future work.

4.3 Baselines during normal operation

Before evaluating the system's fault detection capabilities, it is necessary to characterize its behavior during normal operation with no injected faults. Table 4.1 presents the anomaly and correlation rates observed during an 18-hour overnight run on the final codebase with no application load and no injected faults.

Table 4.1: Anomaly and correlation rates during normal operation (0% load, no injected faults, 18 hours).

Metric	**Total (18hr)**	**Rate (/hr)**
Anomalies		
CUSUM_RTT_SHIFT	16156	893
JITTER_SPIKE	15812	874
KERNEL_RTT_SPIKE	14383	795
CUSUM_JITTER_SHIFT	3484	192
RST_SPIKE	1628	90
HIGH_RST	1051	58
Total anomalies	**52562**	**2906**
Correlations		
WORKER_HOTSPOT	4084	225
WORKER_SOURCE_HOTSPOT	2770	153
LATENCY_SOURCE	563	31
WORKER_HIGH_LATENCY	197	10
ANOMALY_HOTSPOT	38	2
PEER_DOWN	0	0
WORKER_ISOLATED	0	0
Total correlations	**7652**	**423**

The system generates approximately 2900 anomalies per hour during normal operation. These represent real measurable deviations in network behavior, meaning the system is functioning as designed. The virtualized test environment exhibits RTT variance of 2–5ms between probe cycles due to scheduling, shared physical infrastructure, and normal TCP behavior. When these fluctuations exceed the z-score threshold (5σ above the adaptive baseline) or the CUSUM change-point threshold, an anomaly is correctly emitted. Each anomaly reflects a genuine statistical deviation that occurred on the network, the system does not fabricate signals but they require further interpretation in order to actually be useful from a monitoring perspective.

The correlation layer similarly reflects real patterns in the underlying anomaly data. When anomalies concentrate on a specific worker or link, the correlation engine reports this observation. During normal operation, the natural variance of the virtualized environment is sufficient to occasionally trigger correlation thresholds, producing the rates shown in Table 4.1. These correlations are not false. They accurately report that, for example, one worker experienced more anomalies than average during a given 30-second window. Whether such a signal warrants operator

attention is a judgment that depends on the operational context and severity of the observed deviation.

This baseline behavior presents a trade-off inherent to any sensitive monitoring system. Reducing the anomaly and correlation volume during normal operation would require raising detection thresholds, which would directly increase the detection latency for real faults or cause subtle degradation to go undetected entirely. The thresholds used in this evaluation were chosen to reliably detect most of the smallest tested faults (e.g., 10ms added latency) within seconds, while some packet loss/gradual tests could take up to around two minutes. Filtering out the baseline variance that these thresholds also capture would risk hiding early indicators of genuine problems, precisely the signals the system is designed to detect.

From a user experience perspective, the role of the operator is to interpret the signals the system provides. The dashboard presents anomaly counts and active correlations as time-series data, allowing the operator to distinguish between the steady background rate and a sudden spike that coincides with degraded application performance. A sustained elevation in correlation activity, or the appearance of correlation types not normally present (such as `PEER_DOWN` or `WORKER_ISOLATED`, which produced zero activations during 18 hours of baseline monitoring), provides a clear signal that warrants investigation. Improving the system's ability to automatically distinguish operationally significant variance from normal fluctuations, for example, through learned per-environment baselines or operator-defined severity tiers is identified as future work.

4.4 Experiment Results

The evaluation covers six fault categories: latency injection, jitter injection, packet loss, bandwidth throttling, worker-to-worker link failure, network partition, and pod termination. Additionally, a gradual degradation experiment tests the system's sensitivity to slowly worsening conditions. Each experiment was repeated across multiple runs and, where applicable, at three load levels (0%, 20%, and 50% simulated UE traffic) to assess robustness under realistic operating conditions.

As established in the baseline analysis, the system generates anomalies and correlations during normal operation due to real network variance. To ensure that measured latencies reflect genuine detection of the injected fault rather than coincidental baseline activity, only anomalies and correlations that specifically concern the injected worker or its pods are counted toward the detection and correlation latency measurements.

4.4.1 Latency Injection

Table 4.2 presents the detection and correlation latencies for fixed latency injection at four severity levels. The system consistently detects added latency within 0–10 seconds across all delay magnitudes and load conditions. A clear trend emerges: higher injected delays produce faster and more consistent detection. At 0% load,

100ms injection is detected in a consistent 3–4s across all runs, while 10ms injection averages 6s with more variance (2–10s). This is expected: a larger deviation produces a stronger z-score signal on the first affected probe, reducing dependence on probe cycle timing. Nevertheless, even the smallest tested increase (10ms) is reliably detected within 10s in all cases, demonstrating that the system can identify early-stage degradation well below levels that would typically impact reliability.

The correlation engine correctly identifies the affected worker node via the `WORKER_HIGH_LATENCY` correlation in all runs. Correlation latency follows the same trend: 13–14s for 100ms injection versus 17–20s for 10ms at 0% load. Under 20% and 50% load, correlation latency decreases further (4–11s for 100ms, 10–18s for 10ms). This improvement under load occurs because the higher baseline RTT under load means the injected delay pushes the absolute RTT further above the 10ms correlation threshold, allowing the condition (RTT >10ms on >50% of inter-worker links) to be satisfied with fewer probe cycles.

A notable finding is that detection latency at higher delay levels (50ms, 100ms) is remarkably stable at 3s regardless of load, suggesting that the system reaches a floor determined by the probe interval (5s) and the time for the first affected probe to complete. The 10ms injection, being closer to the detection threshold, shows more variance as it depends on the exact timing of the probe relative to the injection and the current baseline variance.

Table 4.2: Detection and correlation latency for latency injection at varying load levels. Detection = time to first anomaly involving the faulty worker. Correlation = time to first `WORKER_HIGH_LATENCY`. All values in seconds.

	0% load					20% load		50% load	
Delay	R1	R2	R3	R4	R5	R6	R7	R8	R9
Detection latency (s)									
10ms	10	2	7	6	5	2	4	2	2
25ms	3	9	0	7	3	4	3	2	7
50ms	0	5	3	3	7	3	3	3	3
100ms	4	3	3	3	3	5	2	3	3
Correlation latency (s)									
10ms	20	19	17	20	17	14	18	10	10
25ms	13	15	19	17	16	14	14	8	10
50ms	15	18	15	18	16	7	15	8	8
100ms	14	13	14	13	13	11	4	6	6

4.4.2 Jitter Injection

Table 4.3 shows results for variable latency injection (5ms ± 20ms with normal distribution), which simulates network jitter rather than a fixed delay increase. Detection latency is 0–5s across all runs and load levels, with the system detecting the jitter pattern within a single probe cycle in most cases.

The `WORKER_HIGH_LATENCY` correlation fires within 15–19s at 0% load and improves to 6–8s under 20% and 50% load, correctly identifying the affected worker in all

runs. Under load, the WORKER_SOURCE_HOTSPOT correlation also fires (within 5–7s), providing an additional localization signal. The jitter experiment generates the highest anomaly volume of all experiments because the variable delay causes continuous threshold crossings. However, the correlation engine filters this volume effectively, producing correct localization rather than being overwhelmed by the anomaly count.

Table 4.3: Detection and correlation latency for jitter injection (5ms ± 20ms, normal distribution). Correlation = time to first WORKER_HIGH_LATENCY. All values in seconds.

	0% load					20% load		50% load	
	R1	R2	R3	R4	R5	R6	R7	R8	R9
Detection (s)	5	1	2	2	2	0	5	0	1
Correlation (s)	15	18	19	16	17	8	6	7	8

4.4.3 Packet Loss

Table 4.4 presents results for packet loss injection at four severity levels. Unlike latency faults, packet loss does not directly increase RTT, instead, it causes TCP retransmissions that the system detects via the CUSUM change-point detector (CUSUM_RETX_SHIFT) and the relative spike detector (RETX_SPIKE).

Detection latency ranges from 5-16s at 5-20% loss, increasing to 17-58s at 50% loss. The higher latency at 50% loss is caused by the fault itself: half of the metric reports from the affected worker's pods are dropped, reducing the rate at which anomaly data reaches the central correlator.

The WORKER_SOURCE_HOTSPOT correlation correctly identifies the source worker in all runs at 5-20% loss. At 50% loss, one run at 0% load failed to correlate (indicated by -) due to insufficient anomaly density within the 30-second correlation window. Under 20% and 50% application load, correlation succeeds in all runs because increased traffic generates more retransmission events, compensating for the packet loss.

The correlation latency for packet loss (10–133s) is notably higher than for latency injection (4–20s). This reflects a fundamental difference: latency injection immediately affects every RTT measurement on every link to the affected worker, providing instant multi-source evidence for the threshold-based WORKER_HIGH_LATENCY correlation. Packet loss, on the other hand, appears as a probabilistic increase in retransmissions across TCP flows. The WORKER_SOURCE_HOTSPOT correlation requires accumulating ≥ 40 anomalies from the affected worker about ≥ 3 targets within a 30-second window, a higher bar that takes longer to satisfy when the underlying signal is based on probability.

Table 4.4: Detection and correlation latency for packet loss injection. Detection = time to first RETX anomaly. Correlation = time to first `WORKER_SOURCE_HOTSPOT`. All values in seconds; - indicates correlation did not fire.

	0% load					20% load		50% load	
Loss	R1	R2	R3	R4	R5	R6	R7	R8	R9
Detection latency (s)									
5%	7	7	7	10	5	11	10	10	6
10%	4	16	5	10	12	3	6	4	5
20%	7	9	8	13	12	6	17	7	5
50%	22	58	19	17	26	19	14	6	17
Correlation latency (s)									
5%	35	37	35	37	71	22	37	39	71
10%	13	46	42	22	34	21	32	11	10
20%	24	10	22	25	25	14	20	11	16
50%	133	98	19	47	-	24	129	24	43

4.4.4 Bandwidth Throttle

Table 4.5 shows results for bandwidth throttling, which simulates network congestion by limiting the egress rate from a worker node. At 5 Mbit/s, the system detects the fault within 0–9s and correlates within 7–27s. At 1 Mbit/s, detection remains fast (5–8s) but correlation is less reliable as it only fires in 3 of 5 runs.

The anomaly signature produced by bandwidth throttling is similar to packet loss: TCP congestion from the limited egress rate causes retransmissions, while queuing delay increases RTT. The combination triggers the `WORKER_SOURCE_HOTSPOT` correlation. At 1 Mbit/s, the throttle is severe enough that metric reports from the affected worker are delayed or lost, reducing the central correlator's visibility, a self-limiting effect similar to 50% packet loss.

Experiments under 20% and 50% application load were attempted but produced invalid results: the bandwidth constraint was insufficient to carry both application traffic and monitoring reports, causing the affected worker's pods to become effectively unreachable and destabilizing the cluster. The implications of this are discussed in Section 5.2.

Table 4.5: Detection and correlation latency for bandwidth throttling at 0% load. Correlation = time to first `WORKER_SOURCE_HOTSPOT`; – indicates it did not fire.

Rate	R1	R2	R3	R4	R5
Detection latency (s)					
5 Mbit/s	0	7	2	0	9
1 Mbit/s	6	8	5	5	–
Correlation latency (s)					
5 Mbit/s	7	27	27	20	21
1 Mbit/s	–	36	20	38	–

4.4.5 Worker-to-Worker Link Failure

Table 4.6 presents results for a link failure between two specific worker nodes (worker-2 and worker-3), implemented via iptables rules blocking all pod-to-pod traffic between those two workers while leaving all other paths intact.

The system detects the link failure within 5–9s via `PROBE_FAILURE` anomalies. Two distinct correlations fire: `WORKER_ISOLATED` (14–19s) detects that pods on one worker cannot reach multiple peers on the other, while `PEER_DOWN` (22–25s) identifies specific unreachable pods. `WORKER_ISOLATED` fires first because it requires fewer failures to trigger (3 failing peers on the same worker) compared to `PEER_DOWN` (9 probe failures about the same peer).

Unlike a full network partition, both sides of the link failure can still communicate with the central correlator (which runs on a third worker), so the system receives reports from both perspectives. This enables it to observe that pods on worker-2 report failures only to worker-3 peers and vice versa, while all other paths remain healthy.

Table 4.6: Detection and correlation latency for worker-to-worker link failure (worker-2 ↔ worker-3) at 0% load. Detection = time to first `PROBE_FAILURE`. Correlation shows both `WORKER_ISOLATED` (first to fire) and `PEER_DOWN`.

	R1	R2	R3	R4	R5
Detection (s)	5	6	8	9	8
`WORKER_ISOLATED` (s)	14	17	19	17	19
`PEER_DOWN` (s)	22	22	24	25	25

4.4.6 Full Network Partition

Table 4.7 presents results for a complete network partition where one worker node is entirely isolated from all other workers. This is implemented by dropping all inter-worker pod traffic to and from worker-2, leaving only intra-worker communication intact. Critically, the partitioned worker's agents can no longer report to the central correlator (which runs on worker-3), so detection relies entirely on the non-partitioned workers observing that worker-2 pods are unreachable.

The system detects the partition within 9–11s and produces a `WORKER_ISOLATED` correlation simultaneously with the first detection. This occurs because a single probe round from any pod on the non-partitioned side already contains 3 failed probes to worker-2 pods (one for each pod on that worker), immediately satisfying the `WORKER_ISOLATED` threshold. `PEER_DOWN` also fires for each individual worker-2 pod as the 9-failure threshold is reached.

Compared to the worker-to-worker link failure experiment, the full partition produces a faster correlation because the failure is more severe: every pod on the non-partitioned side sees all worker-2 peers as unreachable simultaneously, rather than only a subset. The loss of reporting from the partitioned side does not impair

detection, demonstrating that the system can identify a partition even when one side is completely silent.

Table 4.7: Detection and correlation latency for full network partition (worker-2 isolated from all other workers) at 0% load. Detection and correlation fire simultaneously.

	R1	R2	R3	R4	R5
Detection (s)	11	9	10	10	10
`WORKER_ISOLATED` (s)	11	9	10	10	10
`PEER_DOWN` (s)	11	9	10	10	10

4.4.7 Pod Termination

Table 4.8 shows results for pod termination, simulating a pod crash or OOM kill. Detection latency is 3-9s in 9 of 10 runs, with one outlier at 29s caused by a forwarder pod that maintained its listener socket during an extended shutdown period.

The `PEER_DOWN` correlation fires within 10-15s in most runs. Two runs targeting forwarder pods show higher correlation latency (32-33s) which is suspected to be due to a longer Kubernetes termination grace period, during which probes may still partially succeed.

A key implementation detail enabling fast detection is the *peer grace period*: when a pod disappears from DNS (indicating deletion), the system continues probing its last-known IP for 60 seconds. This ensures that even gracefully terminated pods are detected within one probe cycle rather than waiting for the Kubernetes replacement pod to be scheduled.

Table 4.8: Detection and correlation latency for pod termination at 0% load. Detection = time to first `PROBE_FAILURE`. Correlation = time to first `PEER_DOWN`.

	R1	R2	R3	R4	R5	R6	R7	R8	R9	R10
Detection (s)	6	4	8	6	6	6	29	3	9	6
Correlation (s)	11	14	13	10	11	32	33	15	14	11

4.4.8 Gradual Degradation

Table 4.9 presents results for gradually increasing latency, testing the system's ability to detect slow-onset degradation that might evade threshold-based monitoring. Two step sizes were tested: 5ms steps (0-50ms over 5 minutes) and 1ms steps (0-20ms over 10 minutes).

With 5ms steps, the system detects the degradation within 2-7s of the first step and correlates within 6-9s. The z-score detector fires immediately because even a 5ms step change is statistically significant relative to the sub-millisecond baseline variance.

With 1ms steps, individual anomalies fire at the first step (0-8s), but these are indistinguishable from baseline noise. The `WORKER_HIGH_LATENCY` correlation fires consistently at the 5ms accumulated step (133-135s from start) across all nine runs at all three load levels. This represents the minimum detectable gradual degradation via correlation: the point where cumulative latency crosses the 10ms RTT threshold on a majority of inter-worker links.

Table 4.9: Detection and correlation latency for gradual latency increase. Detection = time to first anomaly involving worker-2. Correlation = time to first `WORKER_HIGH_LATENCY`. All values in seconds.

	0% load					20% load		50% load	
5ms steps (0-50ms)	R1	R2	R3	R4	R5	R6	R7	R8	R9
Detection (s)	2	4	2	2	7	5	4	5	5
Correlation (s)	8	8	8	6	9	7	8	7	8
1ms steps (0-20ms)	R1	R2	R3	R4	R5	R6	R7	R8	R9
Detection (s)	4	1	2	2	0	7	8	6	1
Correlation (s)	133	135	133	134	134	133	135	133	133

4.5 Summary

The evaluation demonstrates that the hybrid monitoring approach can detect and localize most tested fault types within seconds of injection, at negligible resource cost, and without requiring application instrumentation. Latency and jitter faults are detected in 1–10 seconds and correctly localized to the affected worker within 6–20 seconds. Packet loss detection is slower (5–58 seconds) and more variable due to the probabilistic nature of the fault, but correlation still succeeds in the vast majority of runs. The worker-to-worker link failure experiment demonstrates detection of a partial connectivity loss between two specific workers (5–9s detection, 14–19s correlation), while the full network partition experiment shows that complete isolation of a worker is detected and correlated within 9–11 seconds even when the partitioned side cannot report to the central correlator. Pod termination was correctly detected and correlated but with a few outliers that are suspected to have been due to extended graceful shutdown periods. The gradual degradation experiment confirms that the system detects slow-onset latency increases from the first 5ms accumulated step, regardless of load level.

Figures 4.2, 4.3, and 4.4 present the detection and correlation latencies across all fault types at 0%, 20%, and 50% simulated UE load respectively.

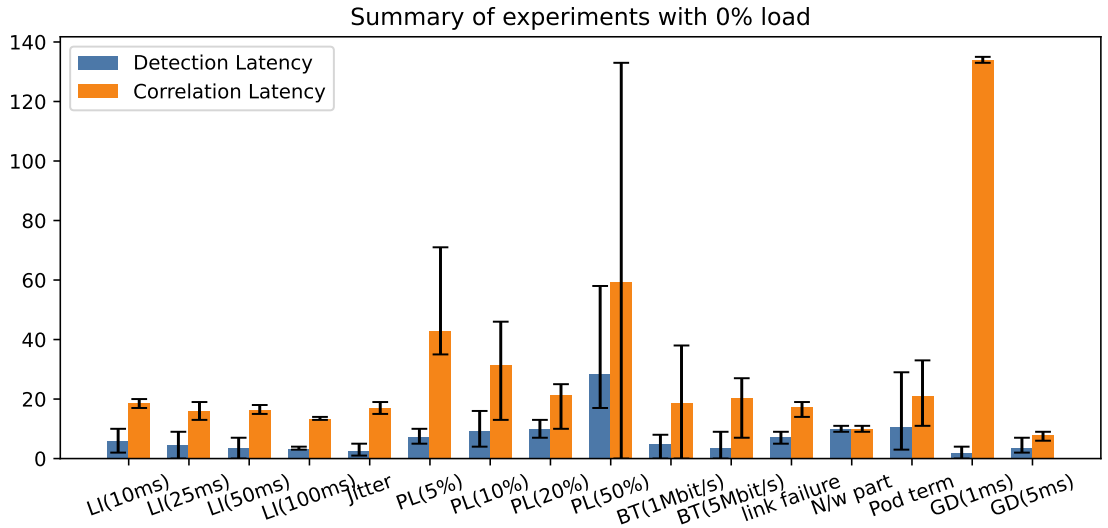

Figure 4.2: Detection and correlation latencies across all fault types under 0% simulated UE load.

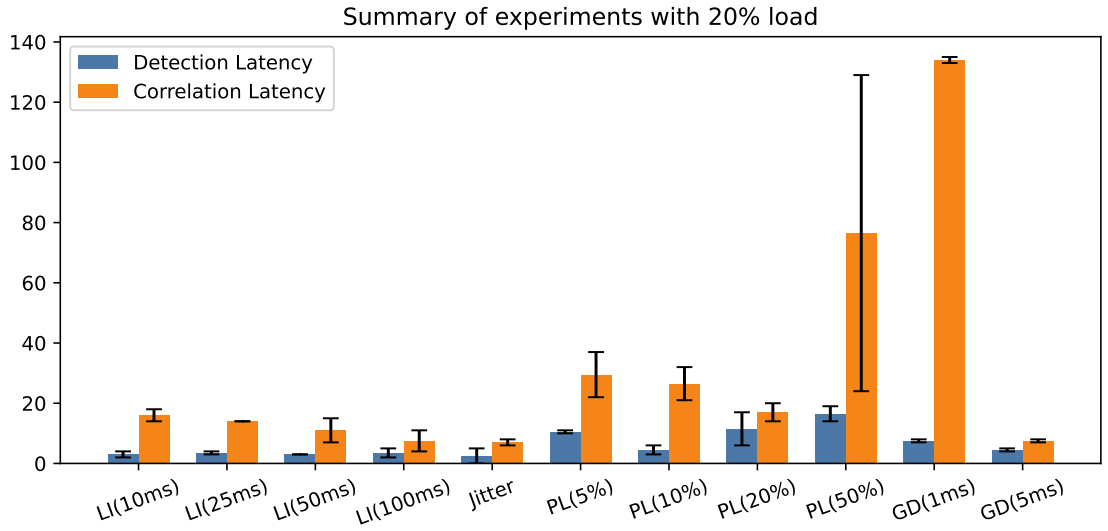

Figure 4.3: Detection and correlation latencies across all fault types under 20% simulated UE load.

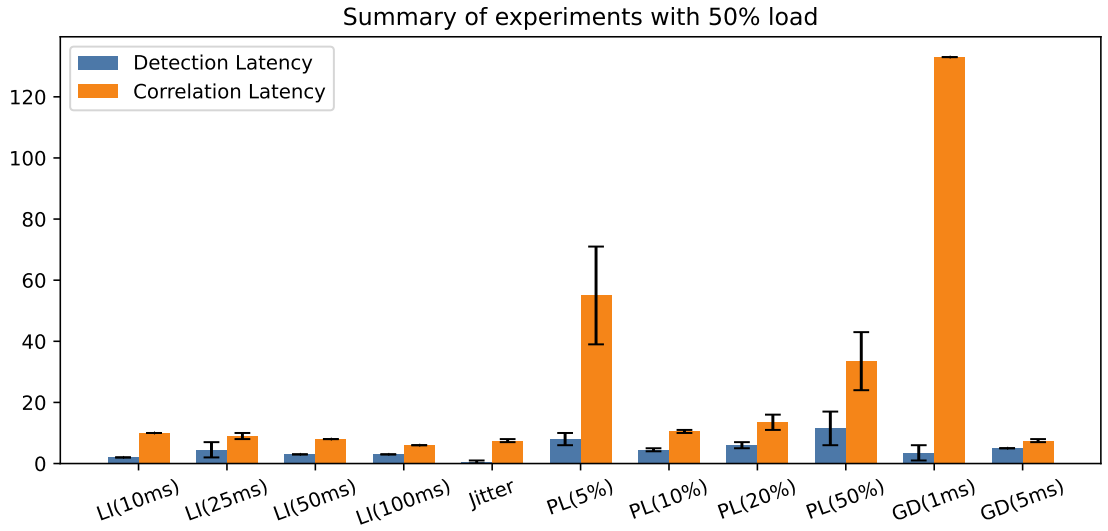

Figure 4.4: Detection and correlation latencies across all fault types under 50% simulated UE load.

Increased application load does not degrade detection performance. Correlation latency generally improves under load because the higher baseline RTT means injected faults push absolute measurements further above correlation thresholds with fewer probe cycles. Detection latency remains stable at 2–10 seconds regardless of load level for latency and jitter faults, confirming that the system operates reliably under realistic production conditions.

The monitoring system introduces negligible resource overhead: 3.4 millicores of CPU (3m userspace + 0.4m eBPF kernel-side) and 4.5 MiB of memory per monitored pod, with the central correlator consuming less than 1 millicore and 2.6 MiB. These values remain constant regardless of application load or fault state, and the total resource consumption by the whole monitoring system represents less than 1% of the total application workload when running at 50% simulated load.

The evaluation also identified limitations. The virtualized test environment produces baseline anomaly rates of approximately 2900/hr due to inherent RTT variance, which in turn produces supplementary correlations that require operator judgment to distinguish from fault-induced signals. Bandwidth throttling under application load is self-limiting: the throttle starves the monitoring system's own metric reports, reducing visibility at the moment it is most needed. Pod termination detection depends on probe timing relative to the Kubernetes graceful shutdown period, introducing variance in detection latency for this fault type. Finally, the $O(N^2)$ probe scaling limits applicability to clusters of larger size without modifications.

In summary, the system detects faults as subtle as 10ms of added latency or 5% packet loss, correctly attributes them to the affected infrastructure component, and maintains this capability under realistic application loads up to a 50% simulated load.

5

Discussion and Conclusion

This chapter interprets the evaluation results in the context of the research questions, compares the approach against established monitoring tools, identifies limitations, and concludes with directions for future work.

5.1 Discussion

The evaluation results demonstrate that the hybrid monitoring approach, combining eBPF-based passive kernel-level observation with active TCP probing and centralized correlation, is effective at detecting and localizing network degradation in a cloud-native 5G deployment. This section interprets the key findings and compares the approach against established monitoring tools.

Detection and localization performance. The system consistently detects latency and jitter faults within 2–10 seconds and localizes them to the correct worker node within 6–20 seconds. These detection times are well below the typical Kubernetes liveness probe interval of 10–30 seconds [35], meaning the system detects degradation before the orchestrator would even begin to suspect a problem. This is particularly relevant for the "gray failure" scenario identified by Huang et al. [16], where a component remains alive but performs poorly, standard liveness probes would never detect such conditions regardless of their interval.

Comparison with existing approaches. Compared to Prometheus with Blackbox Exporter [25], which operates on a pull-based scrape model which is commonly configured at 15–60 seconds [36], the push-based architecture of NetMon provides fundamentally lower detection latency. A Prometheus-based system would require at minimum two scrape cycles to detect a change, and additional evaluation time for alerting rules to fire. Furthermore, Blackbox Exporter probes from an external vantage point, whereas NetMon agents observe the network from within each pod, capturing the actual data-plane perspective experienced by the application. The per-peer kernel-level RTT measurements provided by the TC eBPF programs are not available through external probing. Purely passive eBPF-based observability tools [6] provide deep kernel-level visibility but cannot verify connectivity on idle paths, network routes where no application traffic is currently flowing. In a 5G packet core where traffic patterns are bursty and load-dependent, silent network

partitions between low-traffic pods would go undetected by passive-only approaches. The active probing component of NetMon ensures that every pod-to-pod path is verified every 5 seconds regardless of application traffic patterns.

Resource overhead. The resource overhead measurements confirm that the monitoring system is viable for production deployment. At 3.4 millicores of CPU (including eBPF kernel-side processing) and 4.5 MiB of memory per pod, the system consumes less than 1% of the resources used by the application workload at 50% simulated load. The overhead remains constant regardless of application load or fault state, meaning the monitoring system does not compete with the application for resources during periods of stress when monitoring is most critical.

Correlation accuracy. The correlation engine successfully distinguishes between pod-level, worker-level, and network-level issues in all tested scenarios. The high-confidence correlations (`PEER_DOWN`, `WORKER_ISOLATED`) maintained zero false positives across 18 hours of baseline monitoring, while the softer correlations (`WORKER_HOTSPOT`, `WORKER_SOURCE_HOTSPOT`) provide useful signals but require operator judgment to interpret in the presence of baseline noise. The full network partition experiment further validates the system's robustness: even when the partitioned worker's agents cannot communicate with the central correlator, the system correctly identifies the isolated worker within 10 seconds based solely on observations from the non-partitioned side.

Threshold sensitivity and portability. The system's detection and correlation thresholds encode significant domain knowledge about the monitored environment. Values such as the z-score threshold (5.0), the CUSUM slack parameter (0.5), the minimum absolute deviation (2 ms for agents, 5 ms for central CUSUM), and the correlation trigger counts (e.g., $\geq$9 failures for `PEER_DOWN`, $\geq$3 unreachable peers for `WORKER_ISOLATED`) were tuned iteratively against the specific characteristics of the test cluster. A deployment on different infrastructure, for example bare-metal nodes with sub-millisecond baseline RTT, or a larger cluster with more pods per worker would likely require different values. This is a property of threshold-based detection: the thresholds are interpretable and directly tunable by operators, but they do not adapt automatically to new environments. The CUSUM component partially addresses this by learning per-pair baselines online, but other parameters (warmup period, slack, threshold) remain fixed.

5.2 Limitations

Several limitations were identified during the evaluation:

- **Baseline noise in virtualized environments:** The virtualized test infrastructure produces inherent RTT variance of 2–5 ms between worker nodes, generating approximately 2900 anomalies per hour during normal operation. While these reflect real network behavior rather than false detections, they produce supplementary correlations that an operator must distinguish from

fault-induced signals. On bare-metal infrastructure with more stable network characteristics, this noise floor would likely be significantly lower.

- $O(N^2)$ **probe scaling:** The full-mesh probing topology generates $N \times (N-1)$ probe connections per cycle. While this is manageable at the evaluated scale of 10 pods and estimated to be within acceptable bounds for real-world deployments of 150–200 pods, clusters of a larger size would require modifications such as probe sampling or hierarchical aggregation. The probe-based approach also means that NetMon's view of network health is based primarily on probe behavior and lightweight kernel inspection rather than deep inspection of application traffic.
- **Single point of failure:** The central correlator runs as a single instance. If the controller pod hosting it fails, cluster-wide correlation is lost until the pod is rescheduled. Individual agents continue monitoring and detecting anomalies locally, but cross-pod correlation is unavailable during this window. The full partition experiment demonstrated that this limitation does not prevent detection of the partition itself, as the non-partitioned side provides sufficient observations, but it does mean that the partitioned side's local anomaly data is lost until connectivity is restored.
- **Bandwidth throttle visibility:** Under application load, bandwidth throttling starves the monitoring system's own metric reports, reducing visibility at the exact moment it is most needed. Experiments under 20% and 50% load confirmed this: the bandwidth constraint was insufficient to carry both application traffic and monitoring reports, causing affected pods to become effectively unreachable. This is a limitation of any monitoring approach when the fault affects the monitoring path itself.
- **Elevated Linux capabilities:** The eBPF programs require `CAP_BPF` and `CAP_NET_ADMIN`, which may conflict with security policies in hardened production environments. In the test cluster these capabilities were available, but production AMF deployments may enforce stricter policies that prevent granting them to monitoring workloads. This limits immediate deployability, though the monitoring architecture itself is not affected.
- **Resource overhead:** The AMF is a highly performant system with strict reliability requirements. The probe frequency, packet size, and eBPF program complexity were kept minimal throughout development. As shown in Section 4.2, the measured CPU and memory overhead remained within acceptable bounds, but ensuring that monitoring does not degrade the system it protects remains a constraint for any production deployment.
- **Visualization platform constraints:** CNOM restricts time-series graphs to 20 lines per widget, offers fewer widget types than general-purpose tools such as Grafana (no heatmaps, topology views, or bar charts), and its refresh rate is bound to the Prometheus scrape interval. These constraints required splitting anomaly timelines into separate views by type and by pod, and prevented displaying the full richness of correlation data in a single dashboard. How-

ever, the detection and correlation engine is independent of the visualization layer and could be paired with a more capable frontend without architectural changes.

5.3 Future Work

Several directions for future development are identified:

- **Expanded detection patterns.** The collected data contains rich information that is not yet fully exploited. Additional anomaly patterns could be introduced, such as detecting asymmetric latency (where path A→B has different characteristics than B→A), correlating traffic flow changes with probe failures, or identifying gradual capacity exhaustion from traffic volume trends. Similarly, the correlation engine could incorporate temporal patterns, such as recurring degradation at specific times or correlations between pod scheduling events and network anomalies.
- **Adaptive threshold learning.** The current system relies on manually tuned thresholds that encode assumptions about the deployment environment. A natural extension would be to infer appropriate threshold values from observed data, for example setting the z-score threshold based on the measured baseline variance of each worker-pair, or adjusting correlation trigger counts based on the cluster size and pod distribution. This could take the form of a calibration phase during initial deployment, where the system observes normal operation and derives environment-specific parameters, reducing the need for manual tuning when deploying to new clusters.
- **Machine learning and AI integration.** The system currently relies on statistical methods (Z-score, CUSUM) for anomaly detection. Machine learning approaches could be explored to improve detection accuracy, such as using neural networks to learn complex baseline patterns that simple statistical methods cannot capture, or using classification models to automatically distinguish between fault-induced correlations and baseline noise. Large language model integration could also be explored for automated root cause analysis, translating the system's correlation outputs into human-readable diagnostic summaries.
- **Probe scaling strategies.** For large-scale deployments, hierarchical aggregation where per-worker aggregators reduce the reporting load on the central server could extend the system to clusters of hundreds of pods while keeping the traffic overhead low. One practical implementation of this would be to change the probing mesh to instead be a mesh between all worker nodes and where each pod in a worker probe the others. This would effectively change the scaling to $O(W^2)$ where W denotes the number of workers in the system. The main challenge of implementing this would be a per-worker leader election algorithm that decides which pod in a worker should send probes to other workers and reports to the central server.

- **Leader election for the central server.** Implementing a leader election mechanism among the agents would remove the dependency on a dedicated controller pod and improve fault tolerance of the monitoring system itself.

5.4 Conclusion

This thesis presented the design, implementation, and evaluation of a hybrid network monitoring system for Kubernetes-based 5G packet core deployments. The system combines eBPF-based passive kernel-level traffic observation with active TCP probing and centralized multi-source correlation to detect and localize network degradation within seconds.

The evaluation on a live 5G AMF deployment demonstrated that the system detects faults as subtle as 10 ms of added latency or 5% packet loss, correctly attributes them to the affected infrastructure component, and maintains this capability under application loads up to 50% of the cluster's capacity. Detection latencies of 2–10 seconds and correlation latencies of 6–20 seconds were achieved for latency and jitter faults. Worker-to-worker link failures are detected in 5–9 seconds and correlated within 14–19 seconds, while a full network partition is both detected and correlated within 9–11 seconds. The high-confidence correlations produced zero false positives across 18 hours of baseline monitoring. The total resource overhead of 3.4 millicores CPU and 4.5 memory per pod confirms that the approach is viable for production deployment without impacting the monitored workload.

The hybrid approach addresses a gap in existing monitoring tools: standard health checks cannot detect partial degradation, scrape-based systems introduce detection delays measured in tens of seconds, and purely passive tools cannot verify idle network paths. By combining these complementary techniques and centralizing the analysis, the system provides the early detection and fault localization capabilities required for maintaining service quality in cloud-native 5G infrastructure.

Bibliography

[1] "Distributed systems monitoring," GeeksforGeeks. Section: System Design, Accessed: Jan. 19, 2026. [Online]. Available: https://www.geeksforgeeks.org/system-design/distributed-systems-monitoring/.

[2] Ericsson, "Building a cloud-native infrastructure," 2025. Accessed: Jan. 19, 2026. [Online]. Available: https://www.ericsson.com/497b81/assets/local/core-network/guides/cloud-native-infrastructure.pdf.

[3] P. Willars et al., "Enabling time-critical applications over 5g with rate adaptation," Ericsson, White paper, 2021. Accessed: Jan. 19, 2026. [Online]. Available: https://www.ericsson.com/49bc82/assets/local/reports-papers/white-papers/26052021-enabling-time-critical-applications-over-5g-with-rate-adaptation-whitepaper.pdf.

[4] G. Nunziati, C. Fiandrino, L. Foschini, and P. Bellavista, "Monitoring 5g core networks vulnerabilities with eBPF," *IEEE Networking Letters*, vol. 7, no. 3, pp. 220–223, Sep. 2025, ISSN: 2576-3156. DOI: 10.1109/LNET.2025.3577184. Accessed: Jan. 19, 2026. [Online]. Available: https://ieeexplore.ieee.org/document/11027604/.

[5] H. Bergström and O. Fredriksson, "Detecting network partitioning in cloud native 5g mobile network applications," 2022. Accessed: Jan. 19, 2026. [Online]. Available: https://odr.chalmers.se/items/3f122906-5e46-46c6-8031-f878eacc502a.

[6] D. Soldani et al., "eBPF: A new approach to cloud-native observability, networking and security for current (5g) and future mobile networks (6g and beyond)," *IEEE Access*, vol. 11, pp. 57 174–57 202, 2023, ISSN: 2169-3536. DOI: 10.1109/ACCESS.2023.3281480. Accessed: Jan. 19, 2026. [Online]. Available: https://ieeexplore.ieee.org/document/10138542/.

[7] TailWind. "5 proactive monitoring tools to stay ahead in 2025," Accessed: Jan. 19, 2026. [Online]. Available: https://www.tailwindvoiceanddata.com/blog/5-proactive-monitoring-tools-to-stay-ahead-in-2024.

[8] L. Rice, *Learning eBPF*. "O'Reilly Media, Inc.", 2023. [Online]. Available: https://isovalent.com/books/learning-ebpf/.

[9] "Capabilities(7) - linux manual page," Accessed: Jan. 22, 2026. [Online]. Available: https://www.man7.org/linux/man-pages/man7/capabilities.7.html.

[10] 3rd Generation Partnership Project (3GPP), "System Architecture for the 5G System (5GS)," 3rd Generation Partnership Project (3GPP), Technical Specification (TS) 23.501, version 20.2.0, Jun. 2026, 3GPP TS 23.501, Release 20.

[Online]. Available: portal.3gpp.org/desktopmodules/Specifications/SpecificationDetails.aspx?specificationId=3144.

[11] "3gpp 5gs service-based architecture," ResearchGate. Figure from: 5G Mobile Communication Applications: A Survey and Comparison of Use Cases, Accessed: Jun. 23, 2026. [Online]. Available: https://www.researchgate.net/figure/GPP-5GS-service-based-architecture_fig6_352802141.

[12] "Kubernetes overview," Kubernetes, Accessed: Jan. 27, 2026. [Online]. Available: https://kubernetes.io/docs/concepts/overview/.

[13] "Kubernetes cluster networking," Kubernetes. Section: docs, Accessed: Mar. 18, 2026. [Online]. Available: https://kubernetes.io/docs/concepts/cluster-administration/networking/.

[14] "Kubernetes cluster architecture," Kubernetes, Accessed: Mar. 18, 2026. [Online]. Available: https://kubernetes.io/docs/concepts/architecture/.

[15] "Kubernetes services," Kubernetes. Section: docs, Accessed: Mar. 18, 2026. [Online]. Available: https://kubernetes.io/docs/concepts/services-networking/service/.

[16] P. Huang et al., "Gray failure: The achilles' heel of cloud-scale systems," in *Proceedings of the 16th Workshop on Hot Topics in Operating Systems*, 2017, pp. 150–155.

[17] "Kubernetes pod lifecycle," Kubernetes. Section: docs, Accessed: Mar. 18, 2026. [Online]. Available: https://kubernetes.io/docs/concepts/workloads/pods/pod-lifecycle/.

[18] A. Alquraan, H. Takruri, M. Alfatafta, and S. Al-Kiswany, "An analysis of {network-partitioning} failures in cloud systems," in *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, 2018, pp. 51–68.

[19] "Kubernetes resource management for pods and containers," Kubernetes. Section: docs, Accessed: Mar. 18, 2026. [Online]. Available: https://kubernetes.io/docs/concepts/configuration/manage-resources-containers/.

[20] B. Gbadamosi et al., "The eBPF runtime in the Linux kernel," *arXiv preprint arXiv:2410.00026*, 2024.

[21] S. McCanne and V. Jacobson, "The BSD packet filter: A new architecture for user-level packet capture," in *Proceedings of the USENIX Winter 1993 Conference*, USENIX Association, 1993.

[22] T. Høiland-Jørgensen et al., "The eXpress data path: Fast programmable packet processing in the operating system kernel," in *Proceedings of the 14th International Conference on Emerging Networking Experiments and Technologies (CoNEXT '18)*, ACM, 2018.

[23] B. Claise, "Cisco systems netflow services export version 9," IETF, Tech. Rep., 2004.

[24] P. Phaal, S. Panchen, and N. McKee, "Inmon corporation's sflow: A method for monitoring traffic in switched and routed networks," IETF, Tech. Rep., 2001.

[25] Prometheus Authors, *Blackbox exporter*, https://github.com/prometheus/blackbox_exporter, Accessed: 2026-03-19.

[26] A. B. Barr, O. Lavi, Y. Naor, S. Rampal, and J. Tavori, "Performance comparison of service mesh frameworks: The mTLS test case," in *NOMS 2025-2025 IEEE Network Operations and Management Symposium*, IEEE, 2025, pp. 1–6.

[27] M. Abranches, O. Michel, E. Keller, and S. Schmid, "Efficient network monitoring applications in the kernel with eBPF and XDP," in *2021 IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN)*, IEEE, 2021, pp. 28–34.

[28] V. Chandola, A. Banerjee, and V. Kumar, "Anomaly detection: A survey," *ACM Computing Surveys*, vol. 41, no. 3, pp. 1–58, 2009. DOI: 10.1145/1541880.1541882.

[29] F. E. Grubbs, "Procedures for detecting outlying observations in samples," *Technometrics*, vol. 11, no. 1, pp. 1–21, 1969. DOI: 10.1080/00401706.1969.10490657.

[30] N. A. Heckert et al., "Handbook 151: Nist/sematech e-handbook of statistical methods," 2002.

[31] F. T. Liu, K. M. Ting, and Z.-H. Zhou, "Isolation forest," in *Proceedings of the Eighth IEEE International Conference on Data Mining (ICDM)*, 2008, pp. 413–422. DOI: 10.1109/ICDM.2008.17.

[32] A. Khichane, I. Fajjari, N. Aitsaadi, and M. Gueroui, "5gc-observer: A non-intrusive observability framework for cloud native 5g system," in *NOMS 2023-2023 IEEE/IFIP Network Operations and Management Symposium*, ISSN: 2374-9709, May 2023, pp. 1–10. DOI: 10.1109/NOMS56928.2023.10154433. Accessed: Jan. 19, 2026. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/10154433.

[33] M. Sottile and R. Minnich, "Supermon: A high-speed cluster monitoring system," in *Proceedings. IEEE International Conference on Cluster Computing*, Sep. 2002, pp. 39–46. DOI: 10.1109/CLUSTR.2002.1137727. Accessed: Jan. 19, 2026. [Online]. Available: https://ieeexplore.ieee.org/document/1137727/.

[34] L. Sanabria-Russo and C. Verikoukis, "A cloud-native monitoring system enabling scalable and distributed management of 5g network slices," in *2021 IEEE International Mediterranean Conference on Communications and Networking (MeditCom)*, Sep. 2021, pp. 42–46. DOI: 10.1109/MeditCom49071.2021.9647568. Accessed: Jan. 19, 2026. [Online]. Available: https://ieeexplore.ieee.org/document/9647568/.

[35] "Kubernetes liveness, readiness, and startup probes," Kubernetes. Section: docs, Accessed: May 25, 2026. [Online]. Available: https://kubernetes.io/docs/concepts/configuration/liveness-readiness-startup-probes/.

[36] "Prometheus configuration," Prometheus, Accessed: May 25, 2026. [Online]. Available: https://prometheus.io/docs/prometheus/latest/configuration/configuration/.

A

Appendix

A.1 CNOM Widgets

What follows is a comprehensive list of all the widgets implemented in CNOM at the time of writing. The screenshots were taken during normal operation with no load and no injected faults.

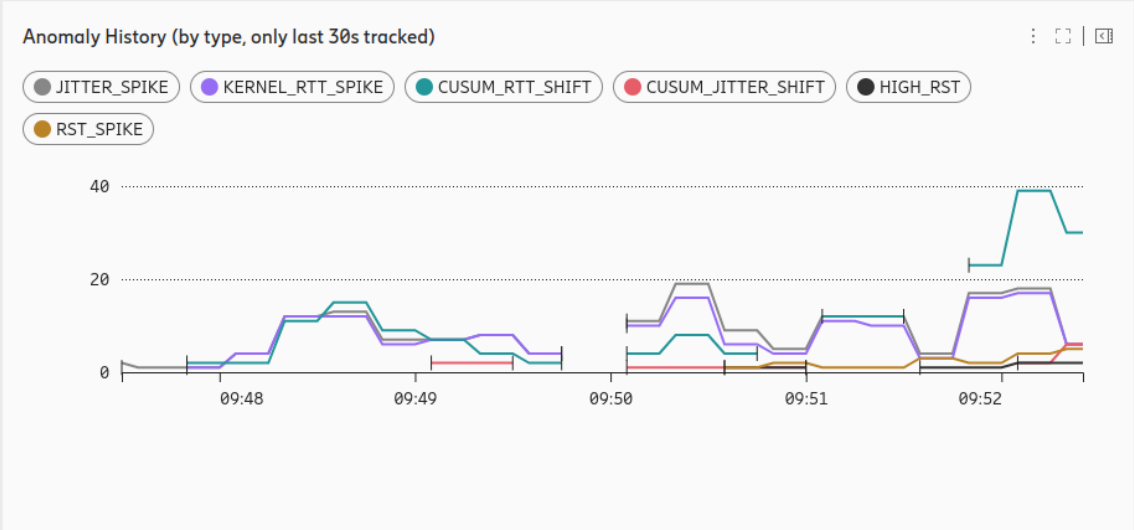

Figure A.1: Anomaly History (by type, only last 30s tracked).

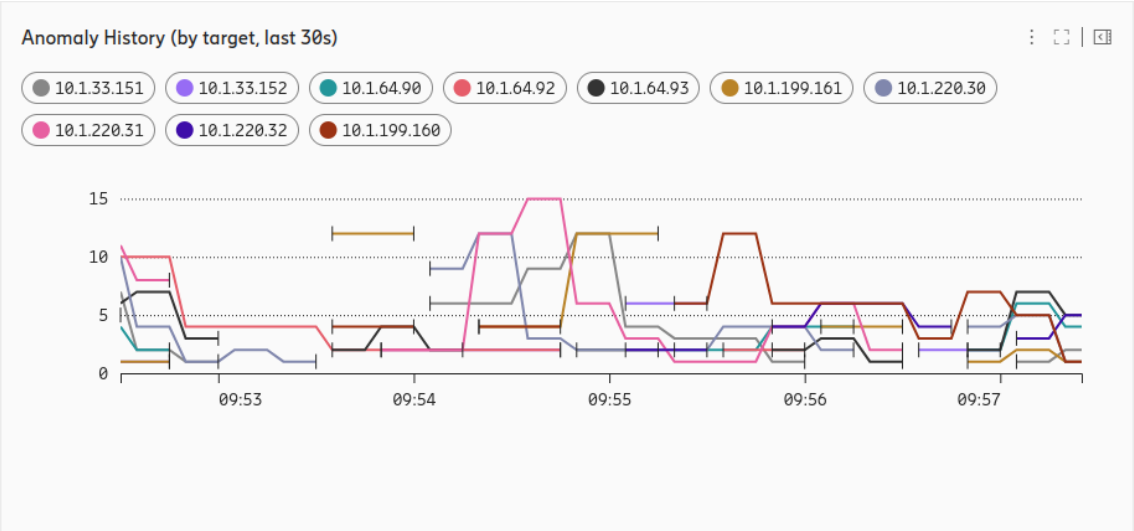

Figure A.2: Anomaly History (by target, last 30s).

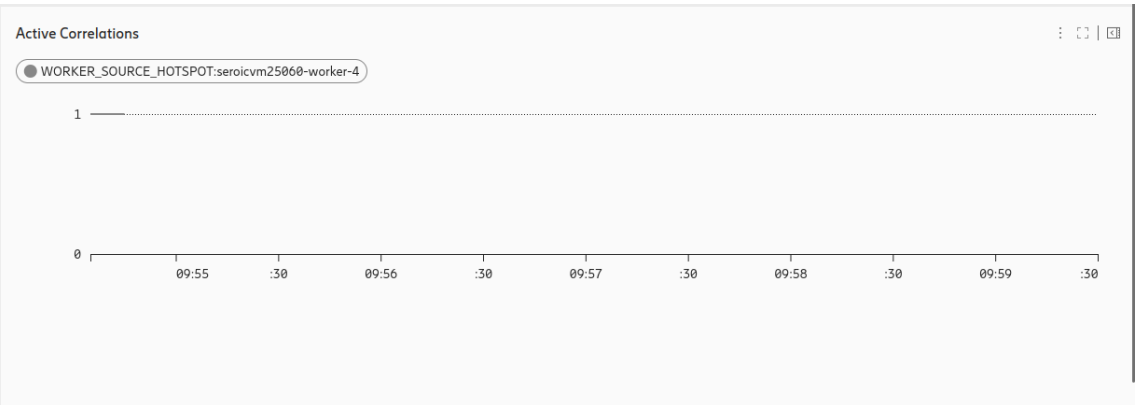

Figure A.3: Active correlations.

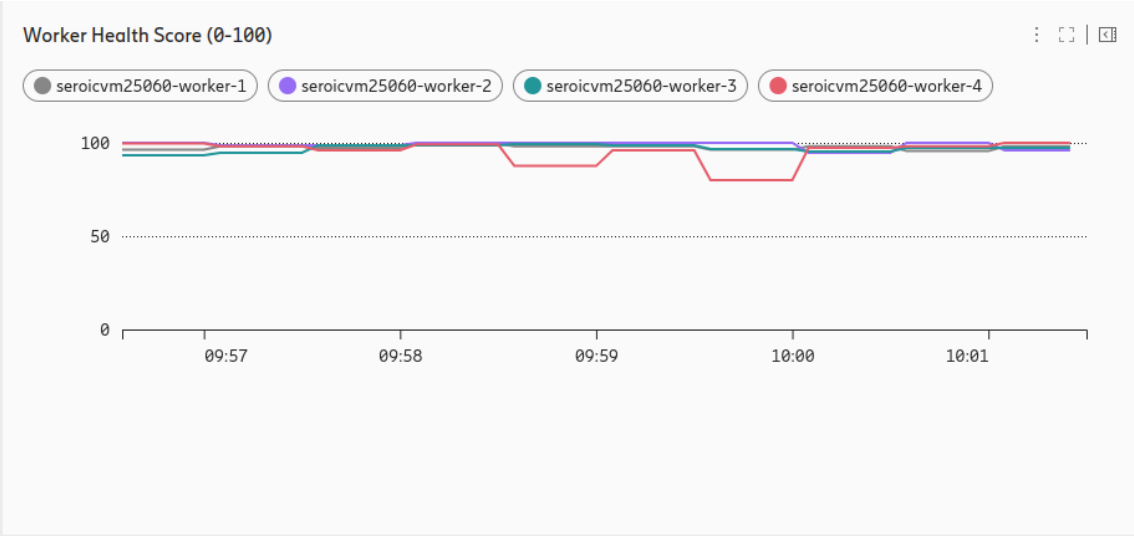

Figure A.4: Worker Health Score (0-100).

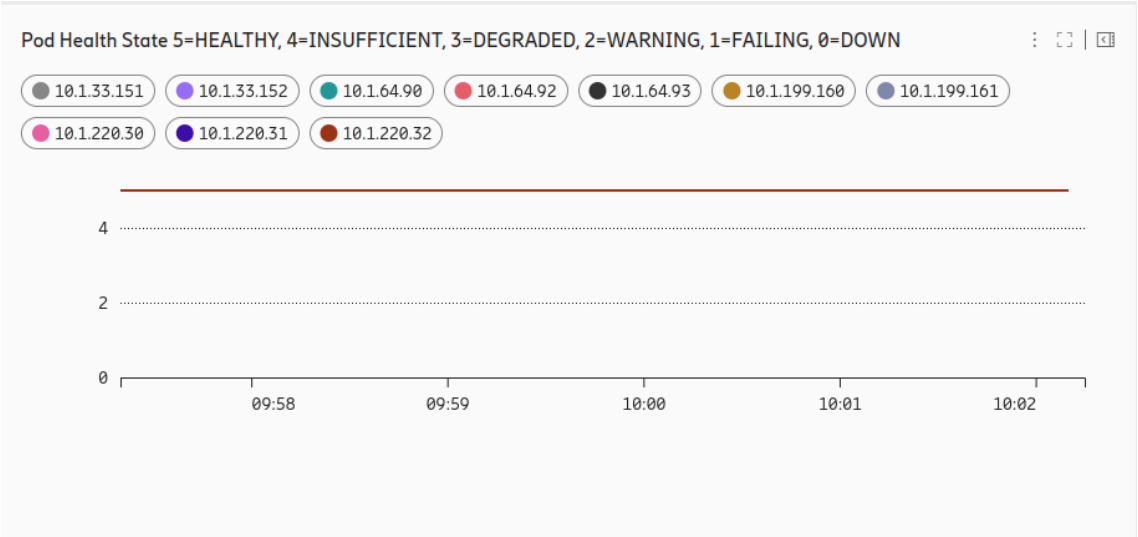

Figure A.5: Pod Health State 5=HEALTHY, 4=INSUFFICIENT, 3=DEGRADED, 2=WARNING, 1=FAILING, 0=DOWN.

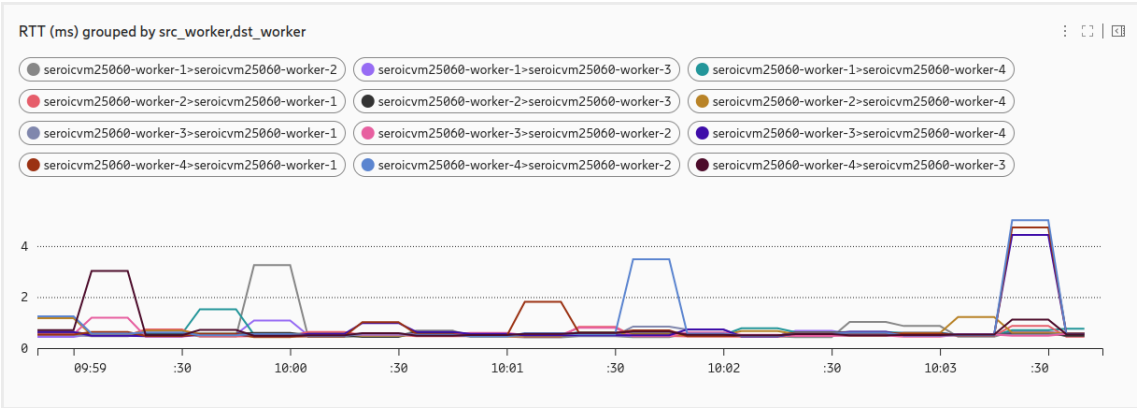

Figure A.6: RTT (ms) grouped by src_worker,dst_worker.

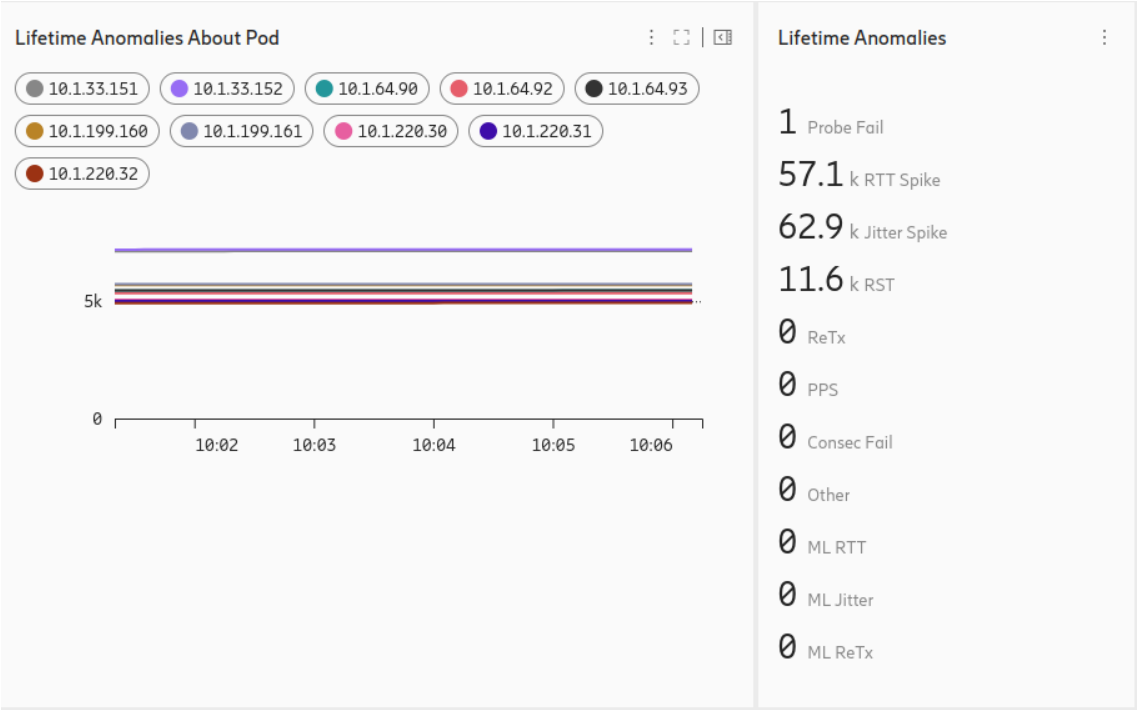

Figure A.7: Two Lifetime anomalies widgets by target and type.

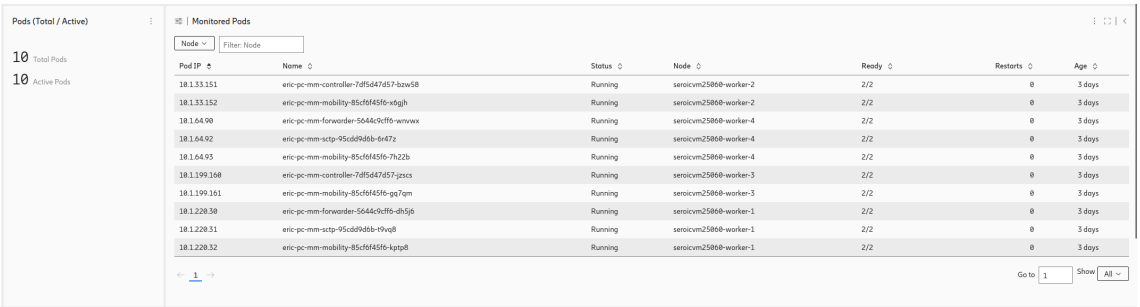

Figure A.8: List of monitored pods containing pod IP, pod name, status, worker node, ready state, number of restarts, and age of pod.

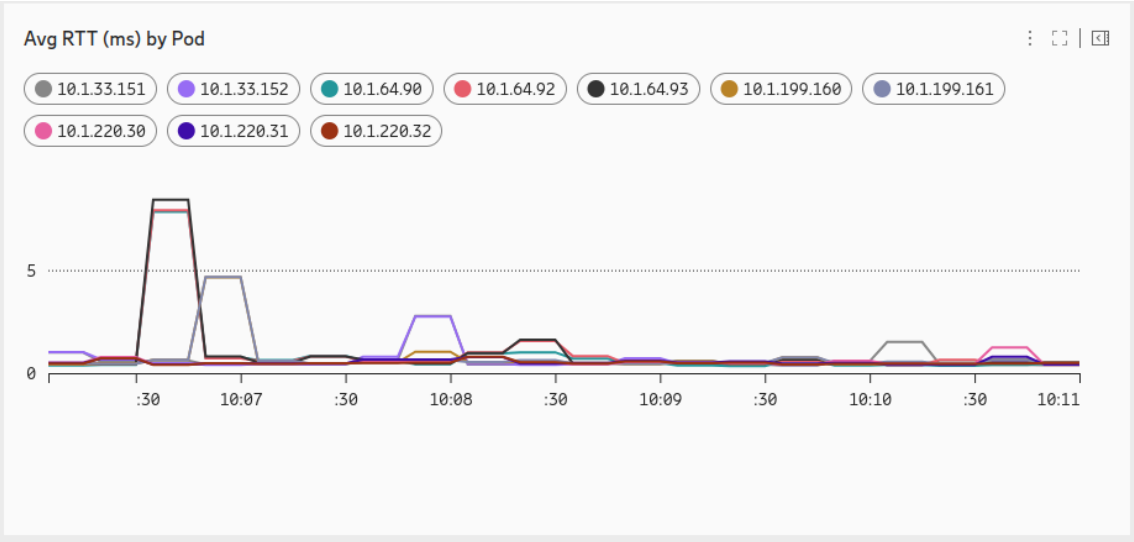

Figure A.9: Avg RTT (ms) by Pod.

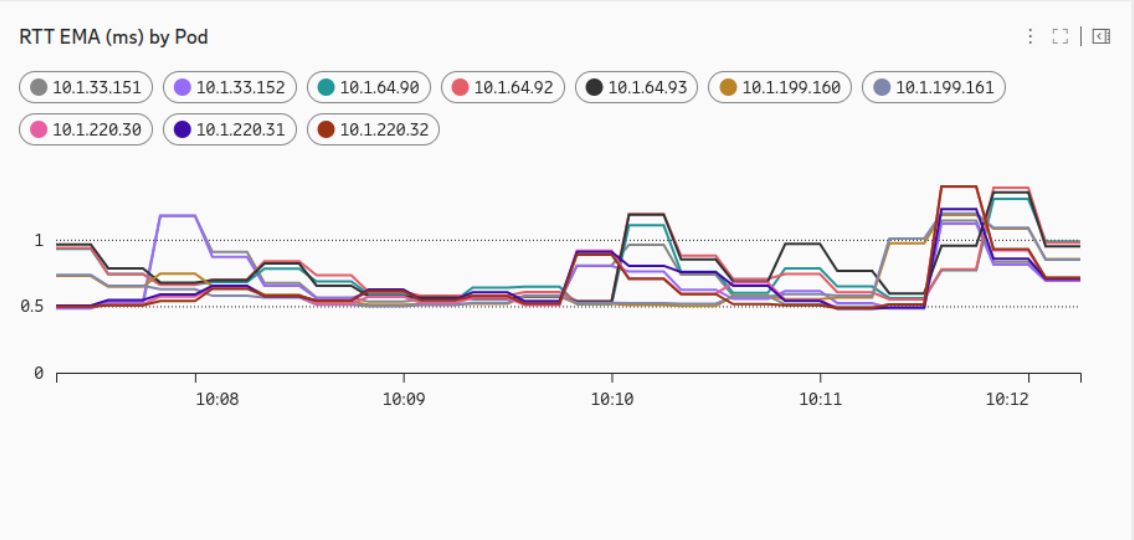

Figure A.10: RTT EMA (ms) by Pod.

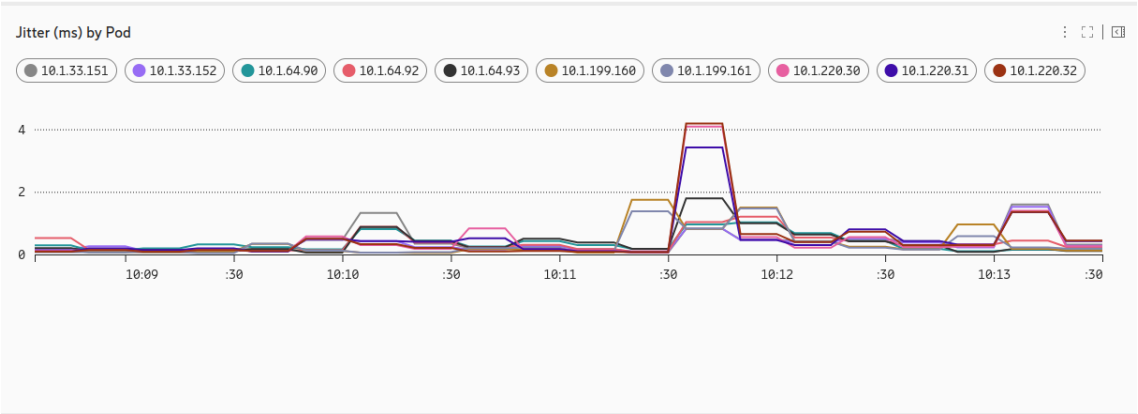

Figure A.11: Jitter (ms) by Pod.

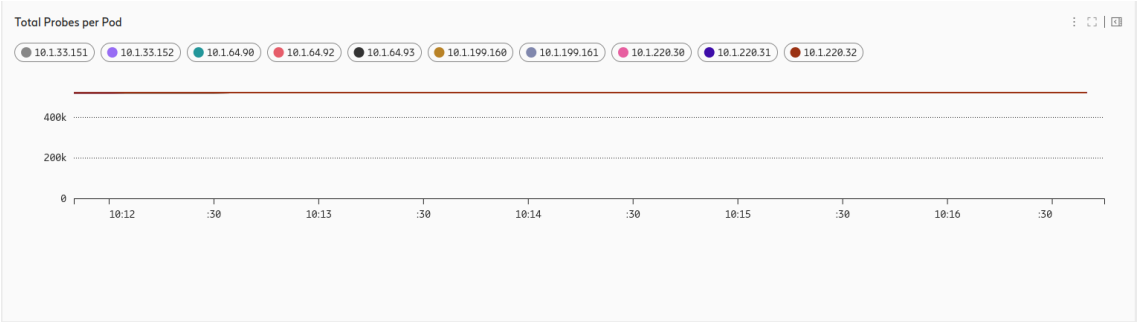

Figure A.12: Total Probes per Pod.

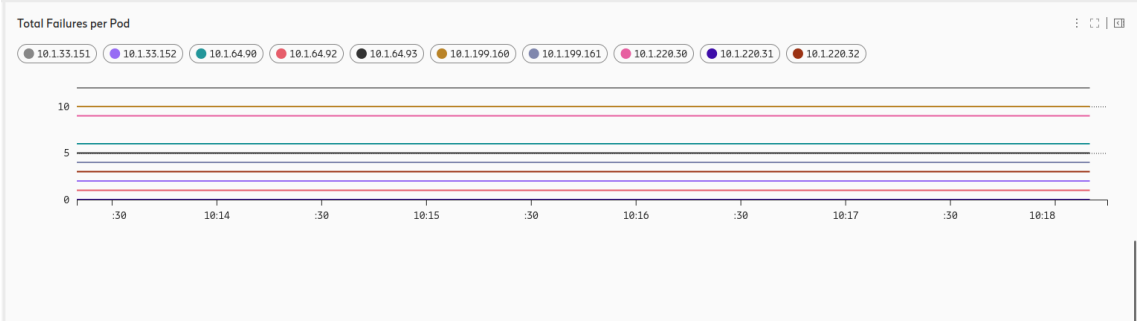

Figure A.13: Total Failures per Pod.

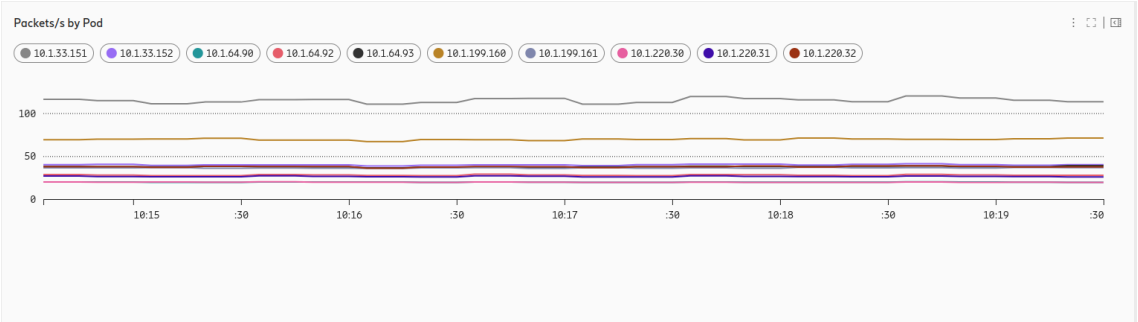

Figure A.14: Packets/s by Pod.

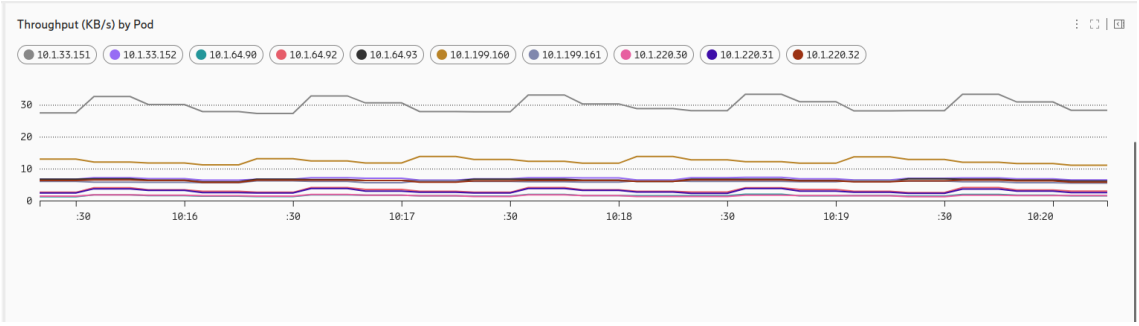

Figure A.15: Throughput (KB/s) by Pod.

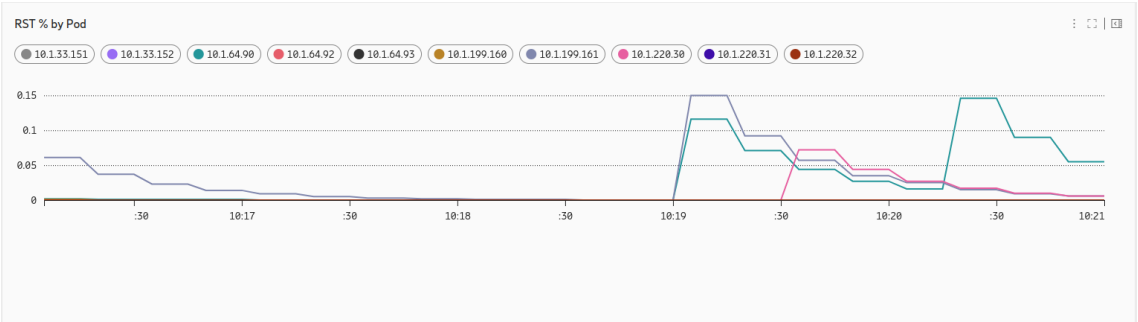

Figure A.16: RST% by Pod.

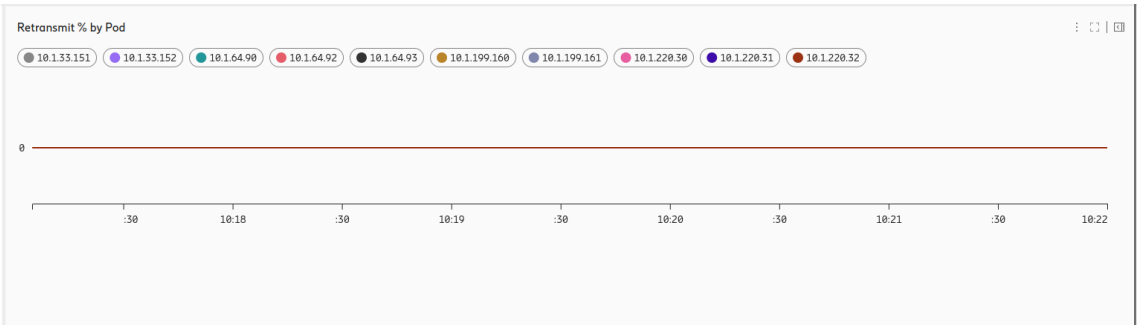

Figure A.17: Retransmit% by Pod.