Distributed Multi-Probe Traffic Analysis Architecture

Amoz Emanuel Moitinho Amaral - 62923

Thesis presented to the School of Technology and Management of Bragança to obtain the master's degree in Informatics within the scope of the double degree program with the Federal University of Technology – Paraná.

Supervisors:

Prof. Tiago Miguel Ferreira Guimarães Pedrosa

Prof. Nuno Gonçalves Rodrigues

Prof. Neylor Michel

Bragança

October 2025

Distributed Multi-Probe Traffic Analysis Architecture

Amoz Emanuel Moitinho Amaral - 62923

Thesis presented to the School of Technology and Management in the scope of the Master in Informatics.

Supervisors:

Prof. Tiago Miguel Ferreira Guimarães Pedrosa

Prof. Nuno Gonçalves Rodrigues

Prof. Neylor Michel

Bragança

October 2025

Dedication

First and foremost, I dedicate this work to God. To my family, especially to my mother, whose unwavering support made every step of this journey possible. To my closest friends, for their presence through so many moments and throughout my academic path. And finally, to my dear father; may God keep him and hold him in heaven.

Acknowledgments

I would like to thank God and my family for everything. This journey would not have been possible without their love, strength, and support. I also thank my supervisors, Prof. Tiago Pedrosa, Prof. Nuno Rodrigues, and Prof. Neylor Michel, for their unwavering support and the knowledge they shared throughout this period. This work would not have been possible without their guidance. Finally, I thank all the professors at the Instituto Politécnico de Bragança and the Universidade Tecnológica Federal do Paraná for their help and teaching over the years.

Resumo

Este trabalho descreve, implementa e avalia uma arquitetura distribuída de monitorização de tráfego com múltiplas sondas, concebida para garantir escalabilidade, resiliência e ingestão eficiente de dados. A motivação decorre das limitações de abordagens centralizadas, que degradam sob carga elevada. A solução adota uma arquitetura em camadas: sondas contentorizadas com Suricata e Fluent Bit publicam eventos EVE JSON no Apache Kafka; no nó central, o Logstash normaliza e enriquece os registos, que são indexados no OpenSearch e visualizados no Grafana. A metodologia compara duas topologias em Kubernetes: uma consolidada, com serviços e sonda num único pod para validação rápida, e outra segmentada, que separa hosts, serviços e sonda por namespace utilizando bridge proxy com Multus para aproximar o tráfego de produção. Os ensaios com replays de PCAP e tráfego adversarial escalaram de uma para cinco e dez sondas, medindo latência fim a fim, utilização de recursos e atraso de filas. Os resultados indicam operação estável e consulta próxima do tempo real até cinco sondas, enquanto com dez sondas o nó central satura CPU e I/O, elevando o backlog e o tempo de indexação mesmo com aumento de partições no Kafka e consumidores no Logstash. Conclui-se que a abordagem é viável e reproduzível, e que a sua evolução exige escala horizontal do núcleo analítico, políticas de ciclo de vida de índices e metas operacionais de latência e atraso, a par de validação em rede produtiva via SPAN ou TAP.

Palavras-chave: monitorização de tráfego, NIDS, Kafka, OpenSearch.

Abstract

This work describes, implements, and evaluates a distributed traffic monitoring architecture with multiple probes, designed to ensure scalability, resilience, and efficient data ingestion. The motivation stems from the limitations of centralized approaches, which degrade under high load. The solution adopts a layered architecture: containerized probes with Suricata and Fluent Bit publish EVE JSON events to Apache Kafka; at the central node, Logstash normalizes and enriches the logs, which are indexed in OpenSearch and visualized in Grafana. The methodology compares two topologies in Kubernetes: a consolidated one, with services and probes in a single pod for quick validation, and a segmented one, which separates hosts, services, and probes by namespace using a bridge proxy with Multus to approximate production traffic. Tests with PCAP replays and adversarial traffic scaled from one to five and ten probes, measuring end-to-end latency, resource utilization, and queue delay. The results indicate stable operation and near real-time querying up to five probes, while with ten probes the central node saturates CPU and I/O, increasing backlog and indexing time even with increased partitions in Kafka and consumers in Logstash. It is concluded that the approach is feasible and reproducible, and that its evolution requires horizontal scaling of the analytical core, index lifecycle policies, and operational latency and delay targets, along with validation in a production network via SPAN or TAP.

Keywords: traffic monitoring, NIDS, Kafka, OpenSearch.

Contents

List of Tables

List of Figures

Acronyms

DDoS Distributed Denial Of Service. 23

DPI Deep Packet Inspection. 27, 33

IDS Intrusion Detection System. 18

ILM Intermediate Level Managers. 14, 15

IoT Internet of Things. 13, 14, 23

IPS Intrusion Prevention System. 18

MAC Media Access Control. 6, 7

ML Machine Learning. 22–24

NIC Network Interface Controller. 5–7

NIDS Network Intrusion Detection System. 18, 20, 21, 27, 36, 38

P2P Peer-to-peer. 14

SIEM Security Information and Event Management. 2, 23

SNMP Simple Network Management Protocol. 12–14, 19

VPS Virtual Private Server. 35, 36, 48, 52, 58

Chapter 1

Introduction

The evolution of computer networks has been marked by exponential growth in both complexity and traffic volume [1]. This scenario, coupled with the emergence of increasingly sophisticated security threats, has necessitated the adoption of innovative approaches for network traffic analysis and monitoring [2]. The ability to identify anomalous traffic patterns, detect malicious activities, and optimize network performance has become essential to ensure the security, efficiency, and reliability of modern communication systems [3], [4].

In this context, the implementation of a distributed and scalable traffic analysis architecture emerges as a solution to the problem at hand. These types of architectures enable the collection, processing, and correlation of information at multiple points in the network, providing a comprehensive and detailed view of traffic [5]. However, the effectiveness of these architectures critically depends on the ability to collect data efficiently and process it in real time, especially in large-scale, high-traffic network environments [6].

This project proposes the development of a distributed traffic analysis architecture based on multiple probes (multi-probe), designed to be scalable, efficient, and capable of detecting anomalous traffic patterns. The architecture will be conceived to support both real-time analysis and post-processing techniques, aiming at network security, scalability, and efficiency. One of the main focuses of the project will be enhancing the traffic collection component, ensuring that the distributed probes operate effectively with minimal impact on network performance.

The relevance of this work lies in its potential contribution to the advancement of traffic monitoring and analysis techniques, offering a solution that can be adapted and used scalably in different network scenarios, from small infrastructures to large corporate networks. Moreover, the proposed approach aims to overcome the limitations of centralized systems, which often face challenges related to scalability and latency in high-traffic environments.

In this thesis, the theoretical and practical foundations of the proposed architecture will be addressed, including the use of containerization (Docker), orchestration via Kubernetes, and distributed probes based on Suricata and Fluent Bit. The practical aspects also encompass centralized log aggregation using OpenSearch and Grafana, namespace-based segmentation to simulate diverse environments, and the evaluation of scalability strategies. A message broker layer based on Apache Kafka is also introduced to decouple probes from the central processing pipeline and improve system resilience. In addition, technical challenges related to data flow, system performance, and integration with security tools such as Security Information and Event Management (SIEM) platforms will be discussed throughout the work.

1.1 Objectives

The proposed work aims to design, implement, and test a traffic analysis architecture based on distributed probes. The specific objectives include:

- Defining a modular and scalable architecture for network traffic capture and analysis;
- Implementing a pilot system of distributed probes capable of collecting traffic data and sending it to a centralized analysis system;
- Developing an efficient mechanism for aggregating, correlating, and visualizing the collected data;

- Evaluating the performance of the solution through testing in both simulated environments and real networks.

1.1.1 Document Structure

The remainder of this document is organized into five chapters, each addressing a specific aspect of the study. Chapter 2 presents the theoretical foundation, reviewing relevant works on distributed systems, network monitoring, and traffic analysis tools, while identifying the technological gaps that motivate this research. Chapter 3 details the conceptual design of the proposed architecture, outlining its layered composition, probe orchestration model, and data flow between distributed components. Chapter 4 describes the practical realization of this architecture, including the deployment environment, containerized components, and orchestration strategies that enable scalability and fault tolerance. Chapter 5 reports the experimental results, analyzing system performance, scalability, and detection effectiveness under different probe configurations. Finally, Chapter 6 summarizes the main findings, discusses limitations, and suggests directions for future work.

Chapter 2

Literature Review

This chapter presents the basic concepts of networks and how understanding them is fundamental to solving problems that involve communication between computers. Following this, the protocols used in the traffic analysis work will be introduced along with the available metrics.

2.1 Traffic Analysis

The broadcast storm is one of the most serious problems a network can face. This issue can occur due to incorrect network configuration or even defective devices used in building the topology. At another time, a single user can consume and use all the available bandwidth, affecting other users connected to the network. To address these and many other traffic-related problems in a network, traffic analysis is performed using a tool to determine the root causes of these issues in a network [7].

According to [8], packet sniffing, also known as traffic analysis, refers to the capture of data transmitted across a network. In this method, the Network Interface Controller (NIC) can be configured to intercept all inbound and outbound traffic. Asrodia and Sharma emphasize its relevance in tasks such as network administration, traffic monitoring, and ethical hacking[1].

[1]Cybersecurity specialists who operate offensively to find vulnerabilities in a particular company's

In line with this, traffic analysis allows for the detection of anomalies in the network, bottlenecks, identification of defective equipment and cabling, observation of system messages not displayed in applications, and identification of security flaws or defects in services available on the network [9].

As described in [10], packet sniffing is the process of capturing and decoding network data to extract information such as the protocols used and plaintext passwords. Similar to how surveillance cameras monitor places, traffic analysis is an indispensable oversight tool for computer networks. A traffic sniffer is capable of capturing network data and displaying it in real time or logging it based on numerous criteria including the source and destination IP address, port numbers, and protocols used. Such information helps system administrators identify security weaknesses, fix problems, and ensure that audit logs are complete. In addition, the tools help system administrators identify and mitigate the risk of failure caused by the presence of unused accounts, unutilized resources, and low-performance sections of the network, ultimately optimizing the performance of the whole network.

When a packet is transmitted from a source to a destination, it traverses multiple intermediary devices. As described in [11], a computer with a NIC configured in promiscuous mode can intercept all traffic passing through the network, regardless of its intended recipient. Each NIC has a unique physical Media Access Control (MAC), which allows it to differentiate among devices. Figure 2.1 illustrates this interception process, where a host running Wireshark, with its NIC set to promiscuous mode, represented by an ear symbol, is able to capture traffic from other hosts connected via a switch.

systems.

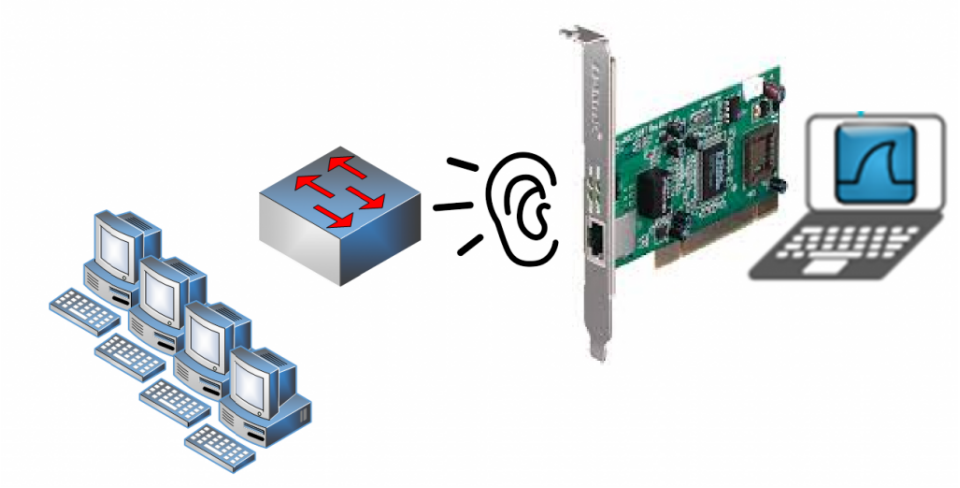

Figure 2.1: Operation of the network card in promiscuous mode

When the NICs receive packets, they are copied to memory and then passed to the operating system's kernel, eventually reaching the user's applications. The sniffing process occurs in two modes: Passive (packet interception in hub-based networks) and Active (packet interception in switch-based networks).

According to [12], a network interface can operate in either promiscuous mode or monitor mode. In promiscuous mode, the NIC of the system used can receive all packets coming from the network, even those not intended for it. In contrast, in normal or non-promiscuous mode, each network interface only processes packets addressed to its own MAC address. For wireless interfaces, monitor mode allows packet capture without associating with an access point. The authors also identified the core components of a network analyzer, which include:

- Hardware: Most traffic analyzers use standard network adapters. Using special hardware can help detect hardware failures, such as *Cyclic Redundancy Check* (CRC)[2], voltage issues, wiring problems, among others;
- Capture Driver: Captures network traffic data and stores it in a buffer;
- Buffer: A location where captured network data is stored for analysis;

[2]Error detection method to identify corrupted data.

- Decoding and Analysis: Responsible for displaying the content of the previously stored network traffic in descriptive text form.

As discussed in [13], packet analyzers represent a fundamental technology for data interception. Their effectiveness is intrinsically linked to the Ethernet protocol, which was designed based on the principle of sharing. In practice, most networks employ broadcast technology, allowing messages intended for one machine to be seen by any computer connected to that network.

In this context, the message is theoretically intended exclusively for the intended computer, being ignored by other connected devices. However, it is possible to configure computers to receive messages even if they are not directly addressed to them, an operation carried out via a sniffer.

2.2 Fundamentals of Distributed Architectures

Distributed architectures are fundamental to the development of modern network analysis and monitoring systems, especially in scenarios where scalability, fault tolerance, and efficiency are critical requirements [14]. This topic explores the basic concepts of distributed systems, their characteristics, and their relevance to traffic analysis in computer networks.

2.2.1 Concepts of Distributed Systems

A distributed system is a collection of autonomous components spread across different machines, communicating via message passing to achieve common objectives. In this thesis, the distributed system is instantiated by a probe fabric that produces telemetry and by a central analytics stack that ingests, processes, and indexes these streams. These systems present several challenges in their design, implementation, and debugging due to their complex infrastructure and the need for coordinated collaboration among independent components [15].

One of the fundamental characteristics of a distributed system is the abstraction of

its distributed resources, offering users a unified and simplified view. Additionally, distributed systems should exhibit properties such as concurrency, consistency, availability, reliability, fault tolerance, openness, and scalability [16]. However, the importance of these properties varies depending on the specific requirements of an application domain [17].

Among the fundamental characteristics of distributed systems, scalability, fault tolerance, concurrency, and decentralization stand out, and these will be explored in detail in the following sections:

- **Scalability:** The capacity of a system to grow in response to demand by allocating work across multiple nodes is known as scalability. For example, Apache Kafka achieves horizontal scalability through topic partitioning, which enables high event rates [17]. In our evaluation, increasing partitions and consumers improved delivery parallelism but did not remove the central-node bottleneck under constrained CPU and I/O, which manifested as sustained backlog growth at ten probes.

- **Fault Tolerance:** The ability of a system to operate in the presence of component failures is known as fault tolerance. Common strategies include message retries in cluster middleware and checkpointing for recovery [16], [18]. Balancing consistency with availability is a key challenge, as illustrated by eventual consistency models [17].

- **Concurrency:** The performance of concurrent tasks is referred to as concurrency. Stream processors such as Apache Flink support real-time pipelines [17], while Kubernetes orchestrates containerized microservices [19]. Typical risks include race conditions in peer-to-peer overlays and the overheads of deadlock detection [15], [20].

- **Decentralization:** A distributed mode of control without a single central authority is known as decentralization. Content Delivery Networks distribute servers across

regions to reduce latency [17], while blockchain networks adopt proof-based consensus [16]. Large-scale leader election and data fragmentation remain practical challenges [18], [21].

Table 2.1 outlines representative techniques used to address scalability in distributed systems, summarizing their typical advantages, disadvantages, and usage contexts. Scalability, or the system's ability to increase its processing power and storage relative to cost and requirements [18], is indispensable for maintaining performance and efficiency in heterogeneous environments.

Table 2.1: Comparative Table of Scalability Techniques

Technique	**Advantages**	**Disadvantages**	**Typical Application**
Efficient Partitioning	Load balancing and parallelism	Synchronization and coordination overhead	Apache Kafka
Data Replication	High availability and resilience	Storage overhead and replication latency	Distributed KV stores
Clustering	Energy savings and locality	Potential load imbalance across clusters	IoT routing protocols

Each technique reflects a balance among efficiency, complexity, and cost:

- **Efficient Partitioning:** Partitioning subdivides data or tasks horizontally to enable parallel processing and load distribution. It is crucial for high-throughput systems such as Kafka [17]. Synchronization and coordination across partitions introduce overheads, particularly where replication or stronger consistency is required [18]. In our setting, increasing partitions and consumer instances raised ingestion parallelism, but throughput remained bounded by the central analytics capacity.

- **Data Replication:** Replicating data across nodes improves availability and recovery, at the cost of storage and potential replication lag [16]. Eventual consistency models trade stronger guarantees for higher availability [16].

- **Clustering:** Cluster-based organization can reduce energy usage and improve locality, though uneven cluster loads may degrade performance in dynamic scenarios [18].

Figure 2.2 illustrates a Kafka cluster with multiple brokers where each broker stores and serves partitions of a topic. Producers publish to the cluster and consumers read from assigned partitions. This multi-node design improves availability and scalability even when individual brokers fail, provided that replication and partition reassignment are correctly configured.

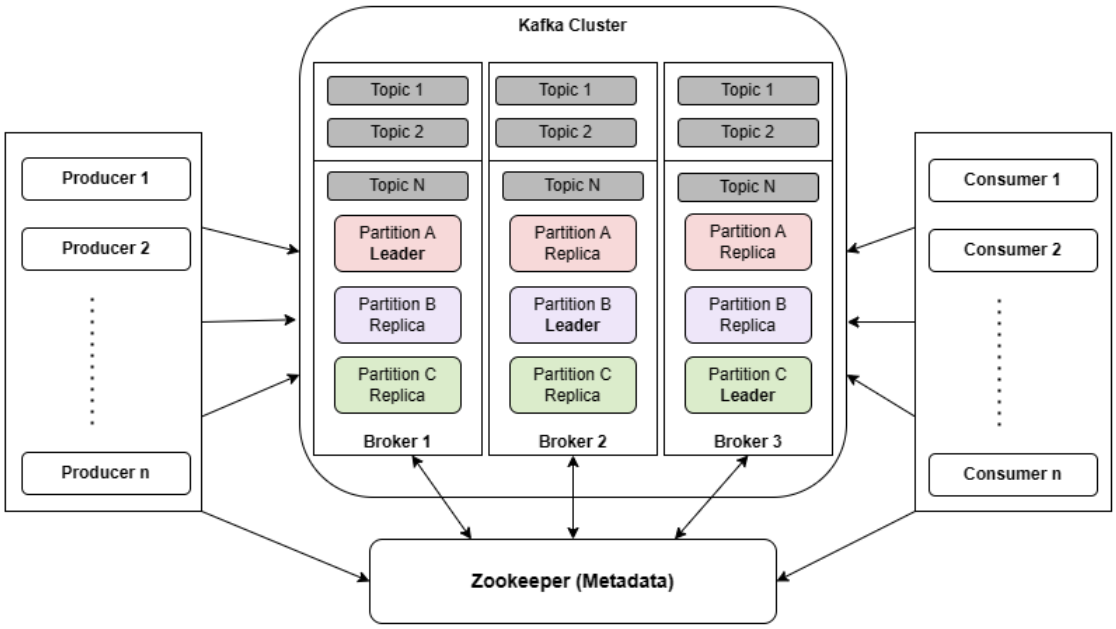

Figure 2.2: Example using Kafka in cluster and partitioning

The Zookeeper is responsible for coordinating the entire cluster, helping the brokers to communicate and maintain system stability. The data is sent to Kafka by the producers and consumed by the consumers. This architecture with several nodes and replicas ensures that even if one of the servers fails, the operation of the system will not be interrupted, guaranteeing high availability and scalability.

2.2.2 Distributed Multi-Probe Network Monitoring

In the context of distributed systems, network monitoring plays a crucial role in ensuring the system's reliability, performance, and scalability. As discussed in the previous section, distributed systems are composed of independent components that communicate and coordinate to achieve a common goal. To maintain the health and efficiency of such systems, it is essential to implement robust monitoring mechanisms that can handle the complexity and scale of distributed environments.

The Internet, as a distributed system, exemplifies this complexity, with its diverse technologies, communication links, and traffic types. This even includes communication methods that utilize fiber optics. As expected, this complexity results in many different forms of traffic on the network [22]. Network administrators face the challenge of detecting and resolving issues within this intricate infrastructure. To achieve this, traffic monitoring tools must leverage the fundamental characteristics of distributed systems, such as scalability, fault tolerance, and decentralization, to provide relevant and reliable insights into network performance and status.

In distributed systems, scalability is a critical factor for tools designed to oversee network performance. As the number of nodes and the volume of traffic grow, these solutions must be capable of scaling horizontally by distributing the workload across multiple probes or agents [23]. For instance, Simple Network Management Protocol (SNMP) based systems can be expanded by deploying several management stations and sharing tasks among them. Similarly, fault tolerance is essential to ensure that the failure of a single agent or probe does not disrupt the entire oversight process [24]. Techniques such as data replication and redundancy in agents can help achieve this. Decentralization, another key characteristic of distributed systems, is reflected in architectures where data collection and analysis are performed locally at each node, reducing the reliance on a central management station and improving the system's resilience [25].

In the context of network monitoring, tools for this activity can be divided into two categories. Node-oriented tools collect information from devices connected to the network,

such as routers, switches, and hosts. These tools must be scalable to handle the potentially large number of nodes in a distributed system. The second category, known as path-oriented tools, focuses on measuring connectivity and latency using utilities like ping, traceroute, and skitter [26] and are especially valuable in distributed environments as they require minimal deployment effort and help localize network issues without relying on monitoring agents at every node. These tools are essential for ensuring the system's fault tolerance and performance, as they help identify and diagnose issues in the network's communication paths.

In this sense and using some of these tools, the main goal of a network administrator in monitoring such an intricate infrastructure is to detect the highest number of issues occurring on the network and subsequently provide some form of guidance or even address these failures. To carry out such monitoring and identify these issues, a widely used protocol by professionals is SNMP. This protocol is simple to operate and easy to implement, as it is an industry-standard protocol [27].

When it comes to the SNMP architectural model, it highlights the need for the existence of management stations and network components. These stations run applications that monitor and control the network components. In distributed systems, the decentralization of these management stations is crucial to avoid single points of failure and to ensure that the monitoring system itself is scalable and resilient. Network components may include hosts, gateways, terminal servers, among others. The management stations send SNMP messages to management agents to acquire information about the network [28], [29].

In practice, distributed multi-probe network monitoring is widely used in large-scale environments such as cloud computing platforms, Internet of Things (IoT) networks, and Content Delivery Networks (CDN)s [30]. For example, in a cloud environment, monitoring tools must track the performance of virtual machines, storage systems, and network links across multiple data centers. Similarly, in IoT networks, where thousands of devices are distributed across wide geographic areas, monitoring solutions must be lightweight,

scalable, and capable of operating in low-bandwidth conditions. These real-world applications highlight the importance of leveraging distributed systems principles in network monitoring [31].

Synchronization Between Probes

In distributed monitoring systems, synchronization policies maintain coherence in data collection and aggregation. For instance, PeerMon [25] utilizes a Peer-to-peer (P2P) architecture in which each probe periodically shares resource data from the entire system to three peers. The peers selected are further heuristically to balance the freshness of the data with network connectivity. This approach eliminates centralized bottlenecks while cyclic updates allow some degree of dependent inter process parallelism while preserving synchronization. Likewise, SIMONE [23] and other SNMP based systems employ Intermediate Level Managers (ILM) who hierarchically divide monitoring work into sub-tasks enabling seamless integration of subordinate pieces at multiple control layers. ILM function as manager-agents (middle managers), allowing cross-layer integration scalability. Control in Kubernetes-orchestrated environments is done through pods [31]. Collectors and probes are bound by ReplicaSets and StatefulSets, which regulates synchronization during dynamic scaling operations.

Filtering and Normalization

For an in-depth investigation to occur, probes must sort and standardize diverse data sources. In the AT&T PacketScope system [22], modified tools tcpdump are used to capture packet headers. Traffic is filtered dynamically with mmdump and IP addresses are anonymized. Correlation of the raw traces with call logs, SNMP statistics, and other files is done at the WorldNet Data Warehouse, serving as a normalization hub. In PeerMon [25] each probe gathers data on CPU, RAM, and users at the local level. They transform these metrics to hashMap entries and share them with peers. In IoT middleware [31],

agents using Protocol Buffers and gRPC[3] serialize metrics like CPU usage and network throughput providing lightweight, structured data exchange which can be integrated into big data frameworks like ElasticSearch.

Handling Packet Loss in High-Speed Networks

Packet loss occurs due to congestion or hardware constraints in high-speed environments. In PeerMon [25], each hashMap fragment is encapsulated in an independent UDP message and sent to prevent blocking if a packet isn't received, which avoids loss. Transient inconsistencies are permitted by the system and eventually, stale data expires (TTL mechanism). Load balanced collectors along with Kafka message queues provides redundancy in Kubernetes based systems [31] to overcome probed failure, ensuring fault tolerance. Buffered disk storage and tape robots in the AT&T measurement infrastructure, prompt uninterrupted trace capture for weeks during peak traffic overloads which needs to be gracefully managed.

Some of the challenges in implementing this type of architecture can be seen below:

- Scalability vs. Overhead: The distribution of probes increases scalability (e.g., ILMs in [23]), but over decentralization increases overhead network traffic. PeerMon [25] limits each probe to three peers, which maintains a bandwidth usage of approximately 120 Kb/s within 500-node networks.

- Dynamic Environments: In cloud-native probes [31], the need arises to adapt with changing microservices. Probes are orchestrated as stateless ReplicaSets within Kubernetes, while isolation is handled by Docker.

- Data Consistency: In the majority, distributed systems focus on eventual consistency rather than absolute. Buga's ASM-based model [24], for instance, tolerates slight discrepancies between clusters and thus aggregates metrics hierarchically.

[3]Is a modern open source high performance Remote Procedure Call (RPC) framework that can run in any environment. https://grpc.io/

Comparative Studies: Centralized vs. Distributed Monitoring Architectures

Table 2.2: Comparison of Centralized vs. Distributed Monitoring Architectures

Criteria	Centralized Architecture	Distributed Architecture
Scalability	Limited due to single-point bottlenecks; struggles with large-scale networks [23].	High scalability via decentralized data processing (e.g., P2P designs in PeerMon [25], Kubernetes ReplicaSets [31]).
Fault Tolerance	Low resilience; central manager failure disrupts monitoring [23].	High resilience through redundancy and self-healing (e.g., automated recovery in Kubernetes [31], data aging in PeerMon [25]).
Precision	Microsecond-level accuracy (software-based tools like Ethereal [32]).	Nanosecond-level accuracy using hardware timestamps (e.g., FPGA probes [32]).
Complexity	Low implementation complexity; suitable for simple networks.	High complexity due to synchronization and specialized hardware (e.g., IEEE1588/GPS in FPGA probes [32]).
Cost	Low cost (commodity hardware and software).	Moderate to high cost (e.g., $400/probe for FPGA solutions [32]).

2.3 Technologies and Tools for Traffic Capture and Analysis

To perform traffic analysis, well-established tools widely used by professionals in the field should be employed. This section introduces key technologies, ranging from traditional tools such as packet analyzers to active network analyzers. By evaluating these tools, the section highlights their role in supporting network monitoring, performance optimization, and rapid fault diagnosis in modern architectures.

2.3.1 Passive Capture Tools

Passive capture tools enable the observation and recording of network traffic without introducing interference or additional packets into the data stream. These tools are essential in distributed systems, as they offer visibility into communications while maintaining minimal system intrusion [33], [34].

Packet analyzers such as Wireshark and TCPdump are commonly used in distributed monitoring environments. They collect network traffic at critical points, like routers or switches, by leveraging port mirroring (SPAN) or network taps. These tools support promiscuous mode, allowing probes to capture traffic not originally destined for them, thus supporting comprehensive data collection across multiple network segments.

Wireshark offers deep inspection capabilities with support for over 1,100 protocols and can operate in both graphical and command-line modes (via Tshark) [35]. Its features such as filtering, geolocation support, and cross-platform compatibility make it widely adopted in network forensics and monitoring pipelines [36].

TCPdump provides a lightweight alternative suitable for low-resource environments or CLI-based automation. It captures packets in raw format for analysis, making it ideal for probing environments where graphical tools are impractical [37].

2.3.2 Active Analysis and Security Tools

Active analysis and security tools are designed to inspect network traffic in real time, detect anomalies, and prevent malicious user activities on the network through the creation of probes or the analysis of traffic patterns. Unlike tools classified as passive analysis, these solutions often interact with the network to identify vulnerabilities, enforce security policies, and respond to threats. These active tools are essential in distributed systems to ensure security, compliance, and performance optimization. Below, two widely used tools in this category, Suricata and Snort, are explored in greater depth.

Suricata

Suricata is an open-source Intrusion Detection System (IDS), Intrusion Prevention System (IPS), and Network Security Monitoring (NSM) tool. It operates in real-time, leveraging multi-threading to analyze traffic across high-speed networks efficiently. Suricata supports protocols such as HTTP, TLS, DNS, and IPv6, and it uses signature-based detection alongside anomaly-based heuristics to identify threats [38], [39].

One of Suricata's standout features is its ability to perform Deep Packet Inspection (DPI) and extract files from network streams for malware analysis. It integrates with platforms like Elasticsearch and Kibana for log management and visualization, making it suitable for distributed architectures where centralized monitoring is essential [40]. Suricata is compatible with Linux, Windows, and macOS, and it uses the pcap library for packet capture.

Snort

Snort is a widely used Intrusion Detection and Prevention System (IDS/IPS). It employs a hybrid approach, combining signature-based detection (defined by customizable rules), protocol analysis (such as header inspection and connection state tracking), and complementary techniques to identify threats [39], [41].

Snort operates in three main modes: sniffer, packet logger (traffic storage for later analysis), and Network Intrusion Detection System (NIDS) [41]. Its modular architecture and efficiency in Unix-like environments (such as Linux or FreeBSD) and Windows make it a flexible solution for networks of various sizes, ranging from small infrastructures to complex corporate environments [42].

2.4 Probe-based Traffic Analysis Architectures

This section discusses network probes, how they are used and implemented, as well as their use in a decentralized multi-probe architecture, in a similar way to distributed systems,

taking advantage of the main and most advantageous points of this type of architecture, as mentioned in section 2.2.

2.4.1 Network Probes Concept

The probes in this architecture operate in three stages (collection, pre-processing, and transmission), aligning with the distributed system principles discussed in Section 2.2. For instance, using Apache Kafka for data transmission ensures scalability and fault tolerance key requirements for high-traffic environments. Moreover, adopting architectural concepts like programmable telemetry filtering [43] and reconfigurable probe modules [44] further enhances the adaptability of the monitoring system.

- **Collection**: Using network interfaces configured in promiscuous mode or techniques such as port mirroring (SPAN), the probes intercept packets in real time, capturing everything from protocol headers to payloads. This collection process requires optimization to minimize network overload, especially in high-flow environments, where packet loss can compromise the analysis. Recent developments in programmable telemetry suggest that in-network pre-filtering (e.g., P4-based INT detection) can drastically reduce collection overhead by discarding redundant data at the source [43].

- **Pre-Processing**: To reduce the volume of data captured and optimize the final analysis, which will be carried out by a central node, the distributed probes are filtered (e.g. by protocol, IP address or port). They can also perform data normalization and preliminary anomaly detection based on each probe's local rules. ReProbe's adaptive model proposes dynamically reconfigurable collectors that can modify their sampling rate and collected metrics based on system behavior, enabling local decision-making and resource optimization without redeployment [44].

- **Sending Data**: Once the data has been pre-processed, it is sent to the central node or distributed main analysis systems, using SNMP or messaging (e.g., Apache

Kafka). Efficiency at this stage depends on tolerable latency and the robustness of the communication channel between the probes and the central node, guaranteeing integrity even in unstable conditions. As demonstrated in programmable INT architectures, telemetry reports can be selectively transmitted based on per-flow conditions and pre-configured thresholds, leveraging AF_XDP for high-speed report parsing and Kafka integration for distributed ingestion[4] [43].

2.5 Data Processing and Aggregation in Distributed Environments

Network Traffic Monitoring and Analysis (NTMA) at scale relies on distributed pipelines that decouple edge collection from transport and indexing at the core. Prior work shows that stream-oriented backbones and search-friendly storage enable both real-time triage and long-horizon forensics under high volume and heterogeneity [45], [46]. In our design, probes emit structured events; a log transport layer provides reliable, high-throughput delivery; and a search backend sustains low-latency queries over time-partitioned indices.

2.5.1 Data Aggregation and Correlation Techniques

At the collection layer, Suricata exports structured telemetry via EVE JSON, covering alerts, flows, and protocol records. Using structured events reduces parsing ambiguity and enables downstream schema-aware processing and joins across event types, for example linking alerts to flows through stable identifiers and the five-tuple within time windows [47].

For transport and decoupling, a distributed commit log (Kafka) offers partitioned, replicated streams with at-least-once delivery and back-pressure so producers and consumers scale independently [48]. Empirical evaluations of distributed NIDS pipelines built

[4]AF_XDP is the Linux user-space socket family for zero-copy packet I/O. `https://docs.kernel.org/networking/af_xdp.html`

on Zeek/Kafka/Logstash/Elasticsearch confirm predictable throughput and clarify bottlenecks at parsing, classification, and indexing; the same design principles transfer to Elasticsearch-compatible backends [49].

Indexing and interactive search are handled by an Elasticsearch-compatible engine. In our case, OpenSearch provides near-real-time indexing and queryability for time-partitioned indices, while Logstash performs minimal transformation and bulk ingestion. Recent academic theses document Fluent Bit forwarding into OpenSearch with authenticated, TLS-protected ingestion and role-based access control, reinforcing operational viability in production-like settings [50]. This arrangement maintains a clear separation of concerns: probes focus on capture and local enrichment; transport ensures durability and fan-out; and the index layer optimizes queryability and lifecycle.

Pragmatic correlation across distributed probes

Correlation is operational and query-driven rather than predictive. Records are linked along stable join keys and windows: (i) flow-centric joins via flow or session identifiers present in EVE; (ii) alert-to-context joins that relate event_type=alert to preceding or succeeding flow or protocol transactions for the same five-tuple; and (iii) multi-probe joins across deployment labels to reveal coordinated activity. Because enrichment is applied at ingestion time, these joins remain simple in the query layer and avoid heavyweight reconstruction [47], [49].

Fault tolerance and performance considerations

End-to-end resilience derives from replication at each layer: forwarders can buffer locally; transport replicates partitions; and the index layer shards and replicates data. Micro-batching and idempotent bulk indexing mitigate transient overloads, while time-based index partitioning avoids hot-shard pathologies on busy days. Prior evaluations of large-scale NTMA systems and distributed NIDS pipelines motivate capacity planning around partitioning and consumer parallelism, which we apply here [45], [46], [49].

2.5.2 Continuous Flow Processing Systems

In open systems found in cybersecurity witnessing, for example, unattended network anomaly detection continuous flow processing configurations are needed for genuine real-time analysis of high-speed data streams. The low-latency, high-throughput profiles of such frameworks enable defenders to track emerging threats and respond to incidents in both limited and boundless data streams. Within this processing arena, engineers generally choose between two architectural models: native streaming and micro-batching.

Native streaming engines such as Apache Flink and Apache Storm work on a per-event basis, handling records one after the other, which gives them extremely low end-to-end latency [51]. In contrast, Spark Streaming partitions a steady data feed into time-windowed micro-batches that the system processes as brief, discrete chunks. Though this batching strategy raises overall throughput, it consequently adds a layer of latency that native frameworks avoid.

Automation systems benefit from modular analytical frameworks that integrate real-time data processing components. As proposed in [52], Suricata can be deployed alongside Redis, serving as an in-memory cache to accelerate data exchange between sensors and back-end systems. Elasticsearch is used for storing and indexing network traffic metadata, supporting efficient querying during anomaly detection workflows. This integration enables near real-time analysis of network events with reduced latency across distributed monitoring nodes.

Applications in Real-Time Traffic Analysis

The analysis of network traffic in real time has transformed into one of the key components of cybersecurity and the enhancement of networks due to the implementation of integrated platforms, Machine Learning (ML), centralized log management, and distributed systems. These developments are in response to the increasing sophistication of cyber threats and the demand for fast and easily expandable interventions.

A notable example is the Security Onion platform which features integration with Intrusion Detection/Prevention System (IDPS) such as Suricata, Network Behavior Analysis (NBA) tool such as Zeek, and the ELK stack (Elasticsearch, Logstash, Kibana) for comprehensive traffic monitoring and surveillance. In improved simulated environments this platform was able to detect SYN flood attacks, brute-force, and DNS tunneling attacks by utilizing distributed probes. Reinforcing that, Suricata's hybrid detection philosophy based on signature detection rules and anomaly behavioral detection, as well as live monitoring dashboards in Kibana enable security teams to visualize threats and take proactive steps to eliminate them [53]. Such integrated systems underscore the benefits of multi-probing structures in extensive networks where centralized log aggregation and correlation is inevitable for the recognition of complex attack strategies.

Anomaly detection that adapts in real time is now possible thanks to the advancements made by machine learning. For instance, Random Forest classifiers were able to achieve accuracy rates of over 97% for recognizing harmful traffic on the UNSW-NB15 dataset using parameters such as packet size, protocol type, and TTL values. Edge computing further specializes this ability by distributing computing responsibilities. Lightweight ML algorithms installed on edge devices conduct initial analyses locally, minimizing latency and bandwidth usage. This is important in the IoT ecosystem where edge nodes analyze traffic and send only the questionable data to central servers for further scrutiny [54].

Log management tools such as Splunk and IBM QRadar SIEM illustrate the importance of centralized log management. SIEM systems monitor the security features of firewalls, IDS/IPS, and endpoints, managing logs for Distributed Denial Of Service (DDoS) and credential-stuffing attacks. Sophisticated SIEM platforms provide threat intelligence feeds which give useful context that help prevent exploits. As an illustration, correlation engines can detect pattern recognition indicative of synchronized assaults like consistent authentication failures or abnormal level port scanning and access geolocation discrepancies scrutinized against rules set by human analysts or algorithms fed external intelligence data [55]. These functionalities underscore the emerging importance of SIEMs in providing networks with defense mechanisms that are proactive and adaptive.

The adoption of distributed structures guided by scalable communication paradigms and edge computing facilitate improvements in both scalability and fault tolerance within expansive network settings. The installation of lightweight monitoring agents at certain points enhances system resilience and reduces burden on central servers. As described in [56], the Elasticsearch clusters have the ability to ingest and analyze high rate distributed streams of metrics with near real time responsiveness, providing horizontal scalability and rapid querying functionality. This architecture supports effective resource utilization across infrastructures with high availability requirements.

Regardless of these efforts, problems persist including incorrect positive identifications, intentional threats to ML algorithms, and the intensive resource demands of processing in real-time. Researchers stress the importance of incorporating explainable Artificial Intelligence (AI) to boost transparency and more privacy preserving frameworks of federated learning to maintain data confidentiality in a distributed system. There is increasing interest in hybrid models that combine signature detection with ML algorithms for anomaly analysis to achieve increased accuracy without losing flexibility. Other significant works focus on enhancing detection efficiency using ML-based models that simulate and learn traffic behavior, especially in distributed or resource-constrained scenarios [54]. Once data is processed and aggregated, its efficient visualization becomes critical. Section 2.6 addresses traffic data visualization techniques, emphasizing how tools like Grafana and Elasticsearch can be incorporated into the multi-probe architecture to provide real-time dashboards.

2.6 Traffic Data Visualization

Traffic data visualization is a critical pillar for a multi-probe architecture because it enables correlation across geographically distributed sensors and supports timely operator decisions. Dashboards aggregate time-series metrics and events to reveal communication patterns, outliers, and evolutions that are hard to perceive in raw logs alone (see Fig. 2.3). As networks grow in scale and heterogeneity, visualization must cope with visual clutter,

real-time processing costs, and the trade-off between detail and interpretability [57], [58].

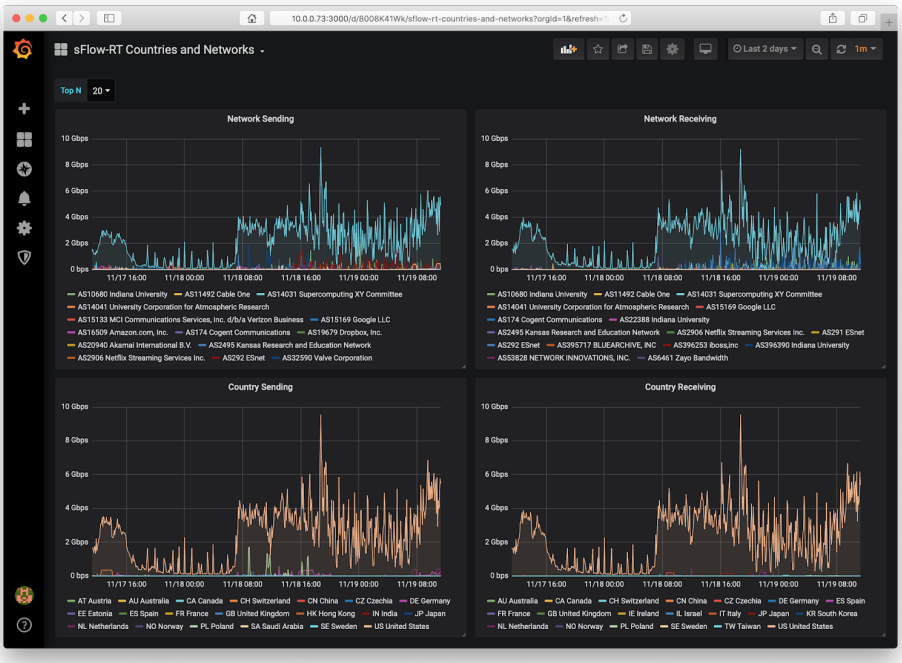

Figure 2.3: Network traffic dashboard in Grafana, showing send and receive metrics in real time.

2.6.1 Role and challenges in multi-probe settings

In distributed deployments, each probe contributes partial, time-stamped evidence. Visualization becomes the integrative layer that aligns these streams, helping analysts spot bursts, sustained drifts, and cross-probe correlations. Prior work shows that modular, containerized stacks (e.g., Snort3 or Suricata for rule-based detection, coupled with Grafana for metrics) can be orchestrated with lightweight services and exposed via APIs or message buses, achieving a pragmatic balance between scalability and usability [59]. However, as node counts and link diversity increase, interfaces must prevent overplotting, keep refresh costs bounded, and preserve context as users zoom from global to local views [57], [58].

2.6.2 Independent pipeline and panel compositions

Modern pipelines follow a clear separation of concerns: edge shippers (e.g., Filebeat *or* Fluent Bit) collect logs and metrics; a streaming backbone such as Apache Kafka provides ordered, durable transport; processors like Logstash normalize and enrich records; indexers (Elasticsearch *or* OpenSearch) store searchable documents; and front-ends (Kibana with Elasticsearch, or OpenSearch Dashboards with OpenSearch) and/or Grafana render interactive visualizations [60], [61]. In practice, Kafka's producer–consumer model absorbs short-term bursts and smooths ingestion while the search cluster allocates shards to sustain query throughput under changing load profiles [61].

For network-security use cases, dashboards combine traffic baselines (throughput, flow counts, service access frequency) with IDS alerts emitted by *Suricata or Snort3.* Suricata's EVE JSON and Snort3's alert streams can be indexed alongside telemetry so that bar gauges, thresholds, and anomaly panels highlight, for instance, multi-port scanning bursts or unexpected service spikes; analysts can then pivot from a spike to the corresponding alert set and underlying flows to prioritize investigation [59]. When Filebeat or Fluent Bit ship structured records into Kafka during peaks—the exact rates are environment-dependent—the buffered flow prevents loss while the search tier maintains interactive updates at short dashboard refresh intervals [61].

2.7 Critical Evaluation of Related Works and Architectural Choices

This section critically compares the main architectural paradigms, tools, and supporting technologies identified in the literature, and examines how their characteristics inform the design of the proposed monitoring framework. The evaluation focuses on scalability, operational efficiency, resilience, and secure service exposure (TLS/RBAC and reverse proxy) as criteria that are essential for high-throughput, distributed network monitoring in container-orchestrated environments.

2.7.1 Distributed vs. Centralized Monitoring Architectures

Centralized network monitoring approaches, including classical SNMP-based systems [23], [28], are straightforward to deploy and manage but exhibit well-documented scalability bottlenecks in large and heterogeneous networks [14]. These architectures typically suffer from single points of failure and constrained polling frequencies, which can lead to incomplete or delayed visibility during traffic peaks.

In contrast, distributed architectures such as PeerMon [25] and ReProbe [44] decentralize the collection process, placing probes closer to traffic sources and enabling localized decision making. This decentralization improves resilience by isolating failures and reduces backhaul bandwidth usage. The proposed system aligns with this distributed vision, but departs from traditional hierarchical agent–manager patterns by leveraging Kubernetes-native constructs (e.g., StatefulSets, namespaces) for elasticity, pod-level autonomy, and rapid recovery. This choice ensures that the addition or removal of probes does not require global reconfiguration, directly supporting horizontal scalability and fault isolation.

2.7.2 Data Capture and Traffic Analysis Tools

Packet capture utilities such as Wireshark and TCPdump [35], [36] remain invaluable for manual inspection and forensic analysis, yet they are not designed for continuous, automated operation at distributed scale. Modern NIDS platforms like Snort and Suricata address these limitations by combining Deep Packet Inspection (DPI) with rule-based or anomaly-driven detection [38], [41].

Suricata was selected for the proposed system due to its multi-threaded packet processing, high-throughput performance, and rich EVE JSON output, which integrates seamlessly with log aggregation pipelines. Compared to Snort, Suricata offers broader protocol parsing, a flexible rule syntax, and metadata output that is directly indexable by search engines such as OpenSearch. This design choice minimizes the need for intermediate parsing layers and supports enrichment within the probe itself, reducing downstream processing overhead. Zeek was also considered for its rich flow and application-layer logs;

however, the present work prioritizes DPI and EVE JSON produced by Suricata at the sensor, leaving Zeek outside the implementation scope.

2.7.3 Logging, Aggregation, and Visualization

The Elastic Stack (Filebeat, Logstash, Elasticsearch, Kibana) remains a standard reference for centralized logging [40], [60]. In high-frequency, containerized settings, however, Filebeat's integration path toward non-Elasticsearch back-ends may require additional components, and its per-event footprint can be relatively higher. In this context, Fluent Bit is a lightweight forwarder designed for constrained environments; it provides native OpenSearch output, built-in Kubernetes metadata enrichment, and flexible filtering, aligning closely with distributed probe deployments. This mirrors strategies seen in systems like PeerMon [25], where context enrichment at the source simplifies downstream correlation.

For visualization, Grafana was chosen for multi-source time-series dashboards and low overhead in interactive panels, while OpenSearch Dashboards serves as the document-level explorer. Empirical studies [61] indicate that Grafana performs well for real-time performance monitoring and cross-source aggregation, which matches the requirements of this work.

2.7.4 Message Bus and Stream Processing Considerations

Message brokers such as Apache Kafka are widely recognized for their ability to decouple producers and consumers, absorb ingestion bursts, and provide delivery guarantees [62]. Architectures like DDAS [63] and in-band telemetry systems [43] leverage such intermediaries to enhance fault tolerance and throughput.

Early smoke-tests in this project employed direct ingestion into OpenSearch for simplicity. The implemented system, however, adopts Kafka between producers (Fluent Bit) and consumers (Logstash) to handle bursty loads, enable consumer parallelism, and maintain delivery under transient failures. This staged evolution reflects operational reality:

direct paths reduce moving parts for initial bring-up, whereas Kafka becomes necessary as probe concurrency and replay intensity increase [31], [45].

2.7.5 Probe Synchronization and Adaptability

Adaptive probe coordination, as implemented in ReProbe [44], enables dynamic adjustment of capture parameters in response to workload changes, improving scalability without full redeployment. The present implementation does not yet perform runtime parameter tuning, but employs pre-configured environment variables and initialization scripts that streamline consistent deployment across multiple probes.

By embedding configuration flexibility at the orchestration layer (via StatefulSets and namespace-specific parameters), the architecture is already positioned to incorporate adaptive behaviors. Future work may integrate Kubernetes operators or sidecar controllers to perform dynamic reconfiguration, bringing the system closer to the self-optimizing models described in adaptive monitoring literature.

2.7.6 Synthesis and Implications

The comparative analysis reveals a clear pattern: each technology choice in the proposed system is informed by operational constraints and the scalability, resilience, observability, and security requirements outlined for distributed monitoring. The selected stack (Suricata and Fluent Bit at the edge, Kafka and Logstash in transit, and OpenSearch with Grafana at the core) balances near-term feasibility with long-term adaptability. Secure exposure of services is achieved by deploying TLS on Kafka and HTTPS for OpenSearch/-Dashboards and Grafana behind a reverse proxy with access control.

As will be shown later in Chapter 5, consumer throughput in Logstash becomes a dominant contributor to end-to-end latency under high probe concurrency, which validates the decoupling via Kafka and motivates consumer-group parallelism and topic partitioning in production-like runs.

2.7.7 Summary Comparison Table

Table 2.3 summarizes the main trade-offs between the alternatives discussed in this section, emphasizing the rationale for the choices implemented in the proposed system.

Table 2.3: Comparison of alternative technologies and design choices

Aspect	**Alternative(s)**	**Advantages**	**Limitations / Trade-offs**
Monitoring Architecture	Centralized (SNMP, hierarchical)	Simple operations; established protocols	Scalability bottlenecks; single point of failure; polling latency
	Distributed (PeerMon, ReProbe)	Fault tolerance; elastic scaling; localized capture	Higher orchestration/coordination complexity
Packet Capture	Wireshark, TCPdump	Detailed inspection; mature tooling	Manual/post-mortem focus; not suitable for continuous distributed use
	Snort	Mature rule base; strong signature detection	Limited parallelism; less rich metadata output
	Suricata	Multithreaded; rich EVE JSON; broad protocol coverage	Slightly higher baseline resource usage
Log Forwarding	Filebeat	Tight Elastic integration; mature ecosystem	Relatively higher footprint in high-frequency containerized settings; non-Elasticsearch backends may require additional components
	Fluent Bit	Lightweight; native OpenSearch output; K8s metadata enrichment	Smaller built-in module set than Logstash
Visualization	Kibana	Deep Elastic integration; powerful document search	Less flexible for multi-source scenarios; heavier rendering
	Grafana	Multi-source dashboards; low overhead; quick setup	Less advanced document-level querying
Messaging	Direct ingestion	Simpler path; minimal moving parts; low baseline latency	No buffering on outages; tighter coupling between producers and consumers
	Kafka	Decoupling; buffering; scalable consumers; delivery guarantees	More components; small added latency; operational overhead

Chapter 3

Approach

This chapter presents the proposed architecture developed to address the challenges of scalable and efficient distributed network traffic monitoring. The architecture enables real-time traffic flows with minimal network overhead through the use of containerized probes, centralized data analysis, and message-based communication.

3.1 System Architecture Overview

The proposed architecture is designed to enable distributed, high-throughput network traffic monitoring with a focus on scalability, modularity, and operational resilience. It is composed of multiple lightweight probes deployed across simulated or real network segments, each responsible for capturing, pre-processing, and forwarding network telemetry to a central analysis node. These probes operate independently, ensuring that localized failures or overloads do not propagate across the system, while Kubernetes orchestration guarantees consistent configuration and automated recovery.

At the core of each probe lies a two container composition: Suricata, which performs real-time DPI and generates structured alerts, and Fluent Bit, which processes the resulting logs, enriches them with contextual metadata, and forwards them to Apache Kafka over TLS as the producing component. This separation of concerns optimizes resource usage and isolates detection logic from log transport, allowing each to be updated, tuned,

or replaced independently.

The orchestration backbone is provided by Kubernetes, using a lightweight distribution selected for its reduced footprint and simplified installation process while preserving full API compatibility. Probes are packaged as StatefulSets to ensure stable network identities and persistent storage mappings when required. This facilitates controlled scaling, as the number of active probes can be increased or reduced using native Kubernetes scaling commands without impacting the central processing pipeline. The resulting probe scalability is illustrated in Figure 3.1.

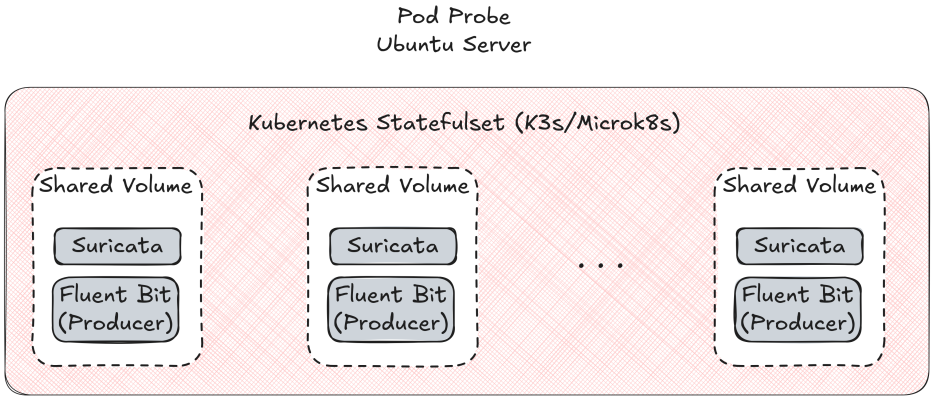

Figure 3.1: Probe scalability using a Kubernetes StatefulSet. Each replica runs Suricata and Fluent Bit (producer) sharing a local volume

The central node aggregates and processes the telemetry received from the probes. Apache Kafka serves as the decoupling mechanism between data producers and consumers, smoothing ingestion spikes and providing fault tolerance through the `nids-logs` topic. Logstash operates as a consumer group that can be scaled horizontally, applies parsing, normalization, and enrichment rules, and inserts the structured events into OpenSearch over HTTPS. OpenSearch is configured as a secure multi-node cluster, with shard replication and TLS encryption ensuring data integrity and availability under sustained load.

For operational visibility, Grafana is integrated directly with OpenSearch over HTTPS to provide real-time dashboards. These dashboards present aggregated traffic flows, alert distributions by severity, protocol usage patterns, and other performance metrics. They are designed with interactive filtering capabilities, enabling analysts to drill down into

data by probe identifier, namespace, or event category.

In the deployed environment the probe pod runs in the simulation network, while the message broker layer and the central analysis services execute on the Virtual Private Server (VPS) that represents the production network. The diagram marks this separation with a vertical dashed boundary. Suricata and Fluent Bit publish JSON records to Kafka over TLS across this boundary, and downstream communication between Logstash and OpenSearch uses HTTPS.

Figure 3.2 provides a high-level overview of the end-to-end data path with this network separation. A single probe pod is depicted for clarity, and the message broker layer shows Logstash as a horizontally scalable consumer group, indicated by the ellipsis.

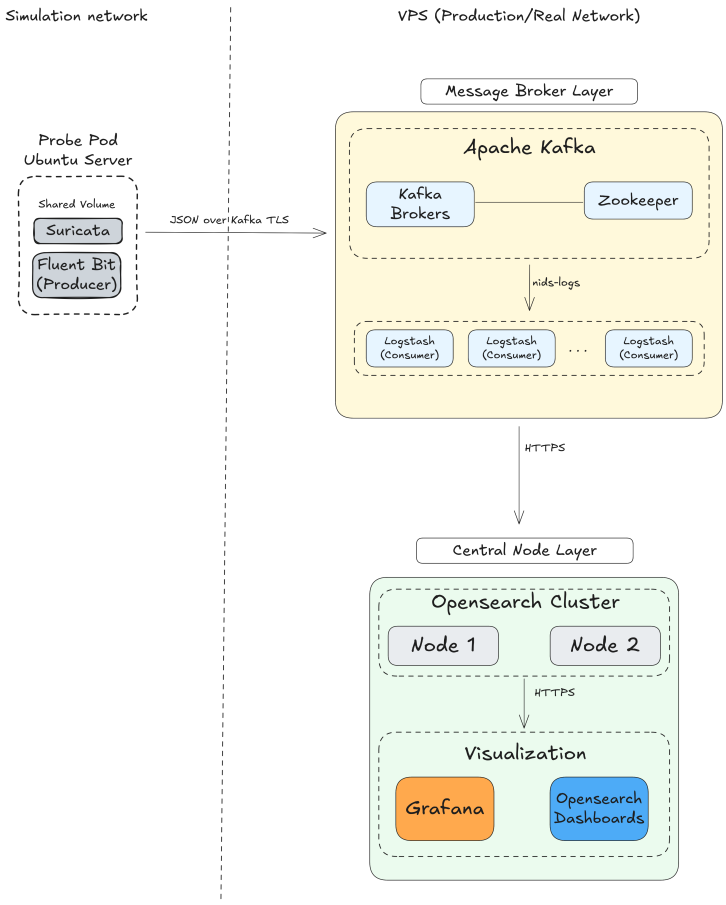

Figure 3.2: General architecture with a single probe pod in the simulation network and the Kafka, Logstash, and OpenSearch stack on the VPS. The vertical dashed line marks the boundary between networks. All inter-site flows use TLS (Kafka and HTTPS).

3.2 Simulation Architectures for Probe Deployment

To validate the monitoring approach under progressively more realistic networking conditions, two simulation architectures were defined. Each variant positions hosts, services, and probes differently in the network stack, controlling how packets are produced, propagated, and ultimately captured by the NIDS. While the central analytics pipeline remains unchanged, the probe-side topology evolves from a compact, single-pod setup to a segmented multi-pod environment with bridge-based traffic mirroring.

3.2.1 Single-Pod Consolidated Topology

The baseline simulation consolidates hosts, application services, and the probe within a single Kubernetes pod. All containers in the pod share the same network namespace, therefore they see the same virtual interface and routing table. In practice, this means traffic between co-located "hosts" and "services" never traverses a node-level bridge or a CNI-managed virtual switch; instead, inter-container flows are exchanged over the pod's local stack. Suricata operates alongside these containers and inspects the resulting traffic directly from the shared interface, capturing both intra-pod communications and any external connections initiated or received by the pod.

In this simulation the probe pod runs in the simulation network on an Ubuntu Server host with K3s, while the message broker layer and the central node execute on the VPS that represents the production network. The diagram marks this separation with a vertical dashed boundary. Suricata and Fluent Bit send JSON records to Kafka over TLS across this boundary, and Logstash writes to OpenSearch over HTTPS.

Figure 3.3 illustrates this layout with the explicit network separation. A single *Probe Pod* encapsulates the host and service containers, the Suricata sensor, and Fluent Bit (producer). All components share the same eth0 inside the pod and exchange logs through a shared volume. The right-hand side shows Kafka with brokers and Zookeeper, a horizontally scalable Logstash consumer group, and a two-node OpenSearch cluster that feeds the visualization stack.

This topology accelerates functional validation by minimizing moving parts: name resolution, service discovery, and logging are simplified because all components observe the same IP address and link. It is also resource efficient, allowing rapid iteration on rules, parsers, and log enrichment without the overhead of emulating multi-hop paths. The main trade-off is scalability. Because hosts, services, and the probe are co-located in a single pod, resources such as CPU, memory, and I/O are shared and cannot be scaled independently; increasing test load (e.g., more users or services) competes directly with the sensor's capture and analysis capacity. Likewise, the single network namespace limits experimental flexibility for adding endpoints, introducing contention, or evaluating failure domain separation. As a result, this architecture is ideal for early pipeline tests and correctness checks, but it is not intended for high fan-out scenarios or for experiments that require independent horizontal scaling of endpoints and sensors.

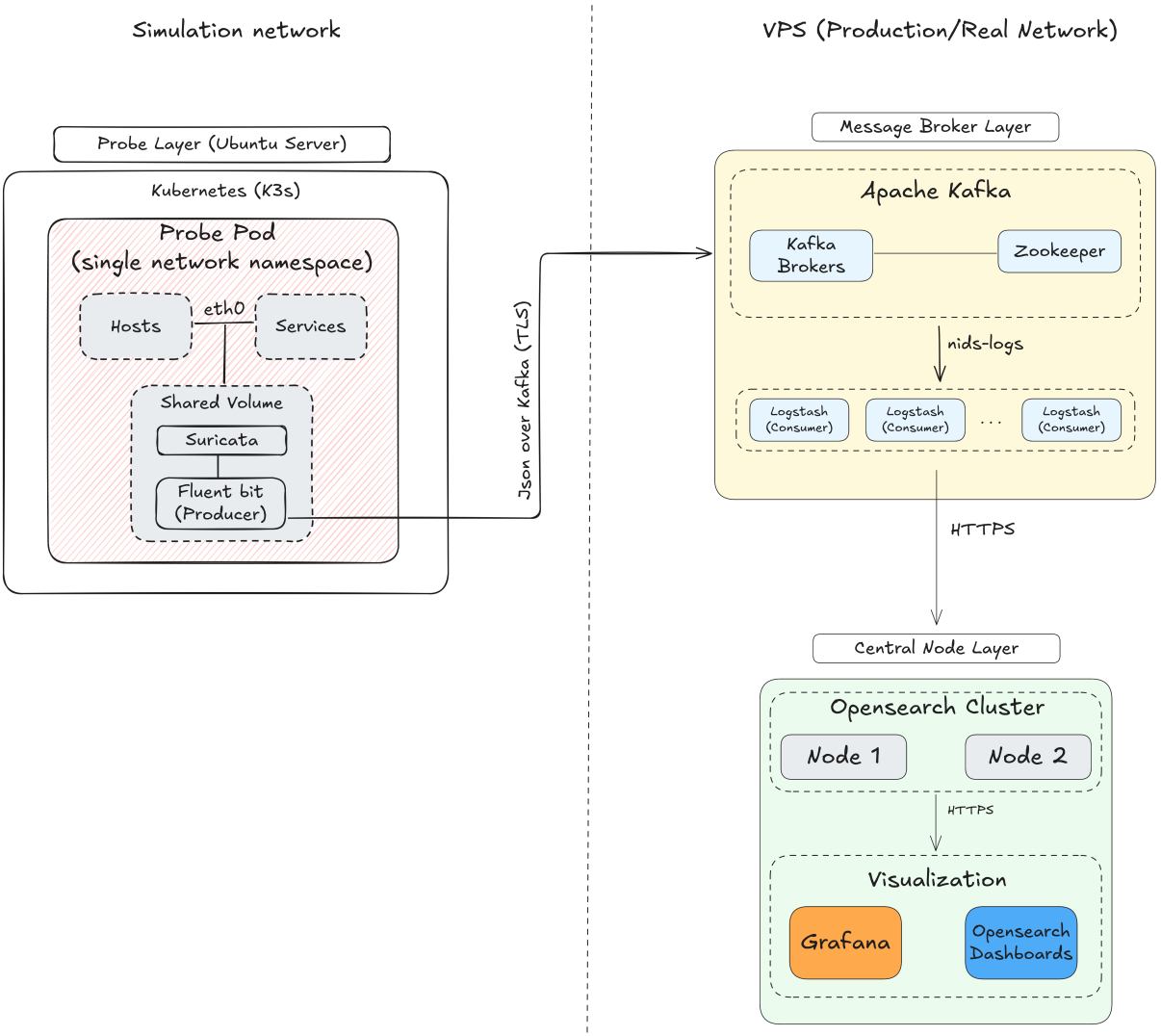

Figure 3.3: Baseline single-pod consolidated topology

3.2.2 Multi-Pod Segmented Topology with Bridge Proxy

The second architecture increases realism by separating hosts, services, and the probe into distinct pods, each with its own IP address and kernel network stack. Communication now traverses the cluster's virtual switching fabric, and the probe does not share an interface with the producers. To ensure the NIDS observes the relevant flows, a bridge-like tap mechanism is introduced between the endpoint pods and the probe. Conceptually, this

"proxy bridge" plays the role that a SPAN port or a network TAP would serve on a physical switch: it replicates frames or streams and delivers a faithful copy to the probe, without altering the original path between endpoints.

Figure 3.4 depicts this layout. A *Pod Host* and a *Pod Service* communicate over the cluster's CNI virtual switch, while a dedicated *Pod Probe* (Suricata + Fluent Bit) receives a mirrored feed via the proxy-bridge, represented as a SPAN/TAP-like link. Inside the probe, Suricata writes EVE JSON to a shared volume consumed by Fluent Bit, which forwards structured events to the common central pipeline (Kafka → Logstash → OpenSearch) over TLS. This visual mapping makes explicit that the sensor inspects traffic from a vantage point equivalent to a monitored link, not from within the endpoints.

With this segmentation, Suricata inspects traffic as if positioned on a monitored link rather than inside an endpoint. The topology exposes realistic behaviors such as ARP resolution, per-pod addressing, and routing decisions by the CNI, and it decouples failure domains so that a probe restart does not affect service availability. The proxy bridge keeps the production path untouched and provides a deterministic data feed into the sensor even as pods scale horizontally, because each new host or service receives its own IP and the mirroring policy simply replicates the corresponding flows to the probe. Compared to the single-pod model, this design is materially closer to field deployments, enabling evaluation of capture coverage, packet timing, and back pressure effects introduced by the virtual fabric, while remaining fully containerized and allowing endpoints and sensors to scale independently.

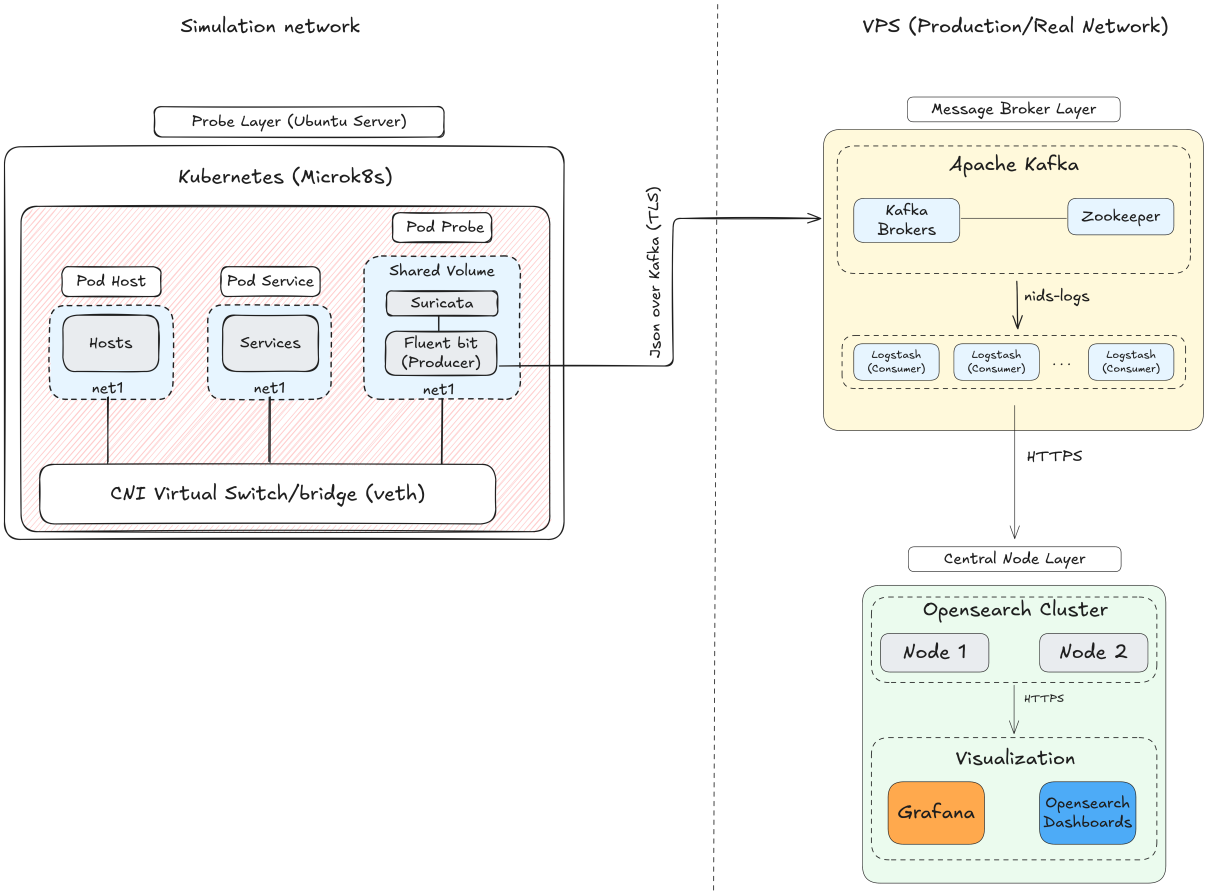

Figure 3.4: Multi-pod segmented topology with a proxy-bridge (SPAN/TAP-like)

3.2.3 Discussion and Rationale

Taken together, the two architectures form a progression from fast, compact validation to more realistic segmentation. The single-pod model accelerates early functional testing of the detection and logging chain. The multi-pod segmented model introduces per-endpoint stacks and an explicit mirroring path into the sensor, surfacing behaviors and bottlenecks of the virtual fabric. Across both variants, the central analytics stack remains constant, isolating the effect of topology on capture coverage, latency, resource usage, and alert rates metrics that, in the subsequent chapters, will underpin the experimental validation of scalability and detection effectiveness.

3.3 Probe Lifecycle and Operation

Each probe's operational life cycle up to a specific timestamp follows a uniform deployment and execution pattern which provides uniformity and predictability across diverse network segments. Each probe is implemented as a Kubernetes pod with a minimum of two master containers. One of the master containers executes Suricata and is responsible for the receipt, inspection, and capture of network packets as well as for the issuance of alerts. The second master container executes Fluent Bit and is responsible for log file processing and transmission. This design of container integration not only achieves a clear separation of work, but also enables the network detection architecture to function independently of the mechanisms to transport logs.

Suricata is processing packets in real-time, in the pod, using its multi-threaded engine to process packets in parallel. In production-oriented deployments, packet capture is accomplished by port mirroring on network switches or through capture ports set to promiscuous mode. In the simulated laboratory setting, this is done by containerized virtual interfaces which emulate the data path without altering the physical infrastructure.

Suricata's captured events, as well as metadata generated, are written to a shared volume that is mounted within the pod. This configuration is a better approach because it eliminates container-to-container networking log transfer requirements. This significantly

reduces overhead. Fluent Bit continuously tails the log files within the shared volume, transforming Suricata's EVE JSON output to be within a structured record format which is easier to use in later processes.

Fluent Bit adds pod name, namespace, and other relevant labels as log entry metadata for context. This enables fine-grained filtering and enhanced correlation during the later stages of the visualization and analysis layers. At this point in time, lightweight filtering is performed to eliminate redundant log entries which leads to the minimization of transmitted data volume. The filtered data preserves the absolute integrity of security-relevant information.

Logs are sent in real time to the Apache Kafka message broker and Fluent Bit is a Kafka producer in this context. This model is essential to boosting system performance. Probes are not made to wait for central node acknowledgments, which makes it possible to take advantage of Kafka's capacity for buffering and retention during bursts of event creation. This approach separates the probes from central and decouples the probes from the central systems for indexing and visualization, which allows them to fail or be scaled in any order.

On the consumer end, Logstash fetches the events from Kafka, performs parsing and enrichment filters, and finally saves them into OpenSearch. The data is processed and turned into the final output, normalized, enriched, and indexed within milliseconds. This enables near instantaneous querying and visualization through Grafana dashboards.

The integration of Kubernetes orchestration, containerized probe components, and a pipeline supported by asynchronous message brokers drives the ability of probe components to operate under varying loads without a drop in performance. This architecture also offers operational resilience: the capture of data in Kafka acts as a buffer during the capture of data in Kafka acts as a buffer during the central node outages or maintenance windows, enabling probes to be individually scaled up and down without losing data.

3.4 Message Broker Integration

To enhance the scalability and resiliency of the data pipeline, a messaging broker layer was introduced between the distributed probes and the central analysis node. Apache Kafka was deployed as a distributed log queue to decouple data producers (probes) from consumers (the central indexing system). In the updated architecture, each probe's logging agent (Fluent Bit) acts as a Kafka producer, asynchronously publishing Suricata event logs to a Kafka topic instead of sending them directly to OpenSearch. A single-node Kafka broker (with Zookeeper for coordination) was set up for the prototype environment, while retaining the ability to scale out to multiple brokers if needed. On the consumption side, an instance of Logstash was configured as a Kafka consumer to pull incoming log messages from the topic, perform any necessary parsing or formatting, and forward the data into OpenSearch for indexing.

This Kafka-backed pipeline (illustrated in Figure 3.2) provides a buffer that smooths out bursts of log events and increases fault tolerance. If the central node or OpenSearch becomes temporarily overloaded or unavailable, Kafka will retain the incoming data, preventing loss and allowing the system to catch up when the consumer is back online. Likewise, the probes are no longer tightly coupled to the indexing backend; they simply "fire-and-forget" messages to the queue, improving overall throughput and reducing backpressure on the probes. Logstash, running on the central node, ensures that logs are uniformly transformed into JSON and enriched if needed (leveraging its Grok and filter plugins) before inserting them into the OpenSearch cluster. This extra processing stage also opens the possibility for more complex analytics or filtering in transit, beyond what was feasible with the lightweight Fluent Bit forwarder.

In terms of deployment, the Kafka and Zookeeper services were containerized and orchestrated on the central node (alongside OpenSearch and Grafana). Logstash was also containerized as part of the central node's Docker Compose stack, configured with pipelines to consume from the Kafka topic and output to OpenSearch over HTTPS (using authentication credentials consistent with OpenSearch's security setup). Care was taken

to allocate sufficient resources to these new components: for instance, Logstash's JVM heap size was tuned to avoid memory contention with OpenSearch, and Kafka's retention settings were adjusted to ensure that no data would be dropped during the test runs.

The introduction of Kafka and Logstash did add complexity to the system, but it closely aligns the prototype with real-world distributed logging architectures, where such message brokers are standard for high-volume data. With this integration, the system moves from a simple pipeline to a more robust streaming architecture capable of handling higher throughputs and providing better guarantees of delivery.

3.5 Centralized Processing and Visualization

The central node comprises the core integration, indexing, and visualization components for all events triggered within the network by the distributed probes. Within this layer, the OpenSearch cluster stands out as the most crucial part. It is deployed as a two-node cluster for high availability and performance. Each node is placed in a separate container, and these containers are managed under the same Docker Compose stack that contains all the central services. To maintain high stability, memory and file descriptor limits are set. Moreover, TLS at client and inter-node level provides encryption, authentication, and data integrity for all communications, safeguarding the integrity of the ingested data.

The ingested log data is split into daily time-based indices (for example, suricata-kafka-YYYY.MM.dd). This helps in shard reallocation, incident response, and targeted querying. OpenSearch uses a distributed architecture and it balances these indices across the two nodes. It also creates replica shards for redundancy. We fine-tune the JVM heap sizes to the container resource limits to reduce the risk of incurring garbage collection pauses which would slow the system down on data-heavy periods.

The stack contains both layers providing complementary visualization services as they pertain to data exploration. Users can carry out ad hoc searches, field analysis, and filtering at very granular levels due to the direct access provided by OpenSearch Dashboards to the indexed documents and their structures. This user interface is helpful in schema

and parser configuration troubleshooting as well as in detailed query construction during forensic analysis.

Grafana, on the other hand, is used as the main real time visualization interface. It is pre integrated with OpenSearch via HTTPS, and utilizes specialized network monitoring dashboards that are already configured. These dashboards highlight the KPIs and other pertinent security metrics like as alert severity distribution, traffic volume and flow, breakdowns of the used protocols and many more with namespace, probe ID, timestamps and even event type filtering. The panels update in real time at short intervals allowing traffic operators to monitor dynamic trends and anomalies. The reason operational monitoring is done in Grafana and not Kibana is due to operational flexibility with multiple data sources, reduced rendering overhead, and faster simplified panel construction workflow which benefited iterative dashboard refinement.

All visualization endpoints are made available through the Traefik reverse proxy, which handles HTTPS termination and domain-based routing. This setup provides controlled and secure access regardless of whether the analysts are remote or in the same network segment, and it also protects sensitive data. Traefik integrates with Docker Compose, granting the ability to declare service-level routing rules in conjunction with the container definitions, which streamlines both deployment and service maintenance.

Combining OpenSearch's indexing functions with Grafana's visualization capabilities while securing the entire layer with Traefik provides a centralized processing stack capable of high-volume log ingestion and flexible interactive, near real-time network security monitoring. This setup also ensures that insights obtained from the probes are not only preserved with high fidelity but also enriched and made actionable through rich, query-driven visual interfaces with instant access.

Chapter 4

Implementation

This chapter translates the architecture presented in Chapter 3 into a concrete, reproducible deployment. The goal is to demonstrate how the conceptual model was instantiated as an operational environment capable of scalable and fault-tolerant network monitoring across multiple simulated domains. Although the setup remains experimental, it already incorporates all mechanisms necessary to measure performance and detection metrics such as latency, resource utilization, accuracy, and load tolerance.

4.1 Implementation Overview

Building upon the layered architecture defined in Chapter 3, the implementation follows the same modular organization, probe layer, messaging and processing layer, storage and indexing layer, and visualization layer, but now expressed through containerized and orchestrated components. Each layer corresponds directly to the conceptual counterpart introduced earlier, preserving the same logical data path while adapting it to a practical execution model.

Each probe is composed of Suricata for deep packet inspection and Fluent Bit for structured log forwarding. Events are generated in EVE JSON format and transmitted asynchronously to the central pipeline through Kafka topics, which decouple data producers from consumers. The downstream stages, implemented with Logstash, OpenSearch,

and Grafana, perform parsing, indexing, and visualization of these records.

Two complementary deployment topologies are realized to test the system under distinct conditions. The first, consolidated topology (Architecture 1), reproduces the single-pod design described in Chapter 3, where hosts, services, and the probe share the same network namespace. The second, segmented topology (Architecture 2), separates those elements into independent pods connected via a virtual bridge configured with Multus. Both topologies rely on the same central pipeline, enabling controlled comparisons of scalability and realism. The next sections describe the infrastructure and implementation details that materialize this design.

4.2 Infrastructure and Experimental Environment

All experiments were conducted in a virtualized laboratory designed to emulate the life cycle of a distributed monitoring infrastructure. The environment combines local virtual machines that host the probe clusters with a remote VPS running the central analytics stack. This separation allows realistic evaluation of communication overhead and resource distribution between edge and core components.

The central node runs on a dedicated VPS with reserved CPU and memory. Its services are containerized and orchestrated with Docker Compose, ensuring reproducibility and isolation. The stack includes OpenSearch for indexed storage, Grafana and OpenSearch Dashboards for visualization, Apache Kafka as the messaging backbone, Zookeeper for coordination, Logstash for stream processing, and Traefik for secure remote access through HTTPS endpoints.

On the probe side, deployments are managed by Kubernetes to provide automation and scalability. Two lightweight distributions were used according to architectural needs: K3s for Architecture 1, chosen for its minimal footprint suitable for single-pod tests, and MicroK8s for Architecture 2, which supports advanced networking through Multus and CNI chaining, necessary for proxy-bridge scenarios. Both maintain API compatibility with upstream Kubernetes, so manifests and operational procedures remain portable.

Each probe is deployed as a StatefulSet to ensure persistent identity and stable networking. Suricata and Fluent Bit containers share a persistent volume for EVE JSON logs, ensuring reliable hand-off between detection and forwarding even under restarts. This infrastructure forms a reproducible foundation for experimentation across both topologies.

4.3 Probe Layer Implementation

4.3.1 Probe Orchestration and Lifecycle

Probes are orchestrated via Kubernetes StatefulSets, which provide predictable hostnames and stable interfaces required for consistent probe identifiers. Each pod includes both Suricata and Fluent Bit containers sharing a local volume. Suricata writes structured events to this volume, and Fluent Bit continuously tails, enriches, and forwards them to Kafka. This design maintains separation of detection and transport concerns while minimizing I/O overhead. Startup routines include automatic rule updates and configuration staging, ensuring that each sensor loads the latest rule set at initialization without introducing mutable state into the base image.

4.3.2 Single-Pod Consolidated Topology (Architecture 1)

This implementation directly materializes the baseline model defined in Chapter 3. Each logical department (e.g., HR, Finance, Development) maps to a dedicated Kubernetes namespace, and its workload is instantiated via a StatefulSet that guarantees stable identities and predictable naming. Because hosts, services, and the probe co-exist within a single pod and therefore share the same network namespace and the pod's eth0, inter-container flows never traverse the node bridge or the CNI switch and are visible to the sensor without any mirroring mechanism.

Inside each pod, the runtime composition follows a consistent pattern. A Suricata container performs deep packet inspection and writes EVE JSON locally; a Fluent Bit container tails the same files and forwards structured records to the central pipeline;

auxiliary containers emulate client and server roles to generate realistic east–west and client–service exchanges. Service diversity is deliberate to stimulate multiple protocol families and log paths. Typical services include a web front-end, a relational database, a file transfer daemon, and a directory service. Lightweight host containers periodically issue benign requests (HTTP fetches, TCP port checks, FTP sessions, ICMP probes), sustaining a continuous stream of application-layer events for parser validation, field mapping checks, and dashboard filtering.

Sensor bootstrapping occurs in two phases to keep the base image immutable while allowing rule management. At initialization, a small utility container copies the sensor configuration from a read-only mount into a writable directory within the pod and runs the rule updater (suricata-update). This approach avoids symlink edge cases and ensures the sensor starts with an up-to-date rule set without embedding mutable state into the image. Once prepared, Suricata runs in privileged mode bound to the pod interface, writes structured events to a shared /logs volume, and exposes no network service of its own. The logging agent tails this shared path, parses JSON at the source, injects lightweight context, and assigns a stable probe identifier derived from pod metadata. Timestamp normalization is performed early to preserve event-time ordering downstream.

Transport is asynchronous through the message broker. Fluent Bit publishes JSON records with explicit timestamps to a designated Kafka topic exposed by the reverse proxy. The details of parsing, enrichment, idempotent indexing, and daily time-based indices are described in Section 4.4 (Central Pipeline Implementation) and conceptually in Chapter 3. This decoupling minimizes probe-side back pressure and isolates capture from transient indexing hiccups.

Operationally, the consolidated pod uses *emptyDir* volumes to separate concerns: one shared path for sensor–forwarder log exchange and small ephemeral state for the forwarder's internal bookkeeping, keeping I/O local and avoiding container-to-container networking on the hot path. StatefulSet semantics provide stable names that are reused as probe identifiers in the analytics stack, simplifying dashboard variables and filters. Because the entire workload resides in one pod, scaling this topology increases contention for

CPU and memory among endpoints, services, and the sensor. This constraint is intrinsic to the model and serves as a reproducible baseline for comparison with the segmented topology that enables independent scaling of endpoints and probes.

4.3.3 Multi-Pod Segmented Topology (Architecture 2)

The second implementation extends the baseline introduced in Section 4.3.2 by distributing hosts, services, and probes into distinct pods, each bound to its own IP address and network namespace. Unlike the consolidated topology, inter-pod communication now traverses the cluster's virtual switch, and the probe no longer shares its interface with traffic producers. To ensure visibility of departmental flows, a dedicated proxy-bridge container is introduced within each namespace. This component mirrors packets from the departmental subnet toward the probe, analogous to a SPAN port or TAP device in physical networks, providing the sensor with an authentic copy of the traffic without interfering with the production path.

The deployment relies on MicroK8s with Multus to enable multiple CNI attachments per pod and to create isolated departmental subnets. Each namespace corresponds to a logical department (e.g., Human Resources, Development, Finance), and within each one, the proxy bridge guarantees that the corresponding Suricata instance receives an uninterrupted copy of both east–west and client–service communications. This structure reproduces more faithfully the operational conditions of enterprise monitoring infrastructures, where sensors are typically positioned on mirrored links rather than co-located with the endpoints.

Within the probe pod, Suricata and Fluent Bit operate following the same model described for the single-pod implementation: Suricata generates structured EVE JSON events, and Fluent Bit performs local normalization and forwarding to the central pipeline (Kafka $\rightarrow$ Logstash $\rightarrow$ OpenSearch). Kafka provides buffering and decoupling, ensuring resilience to indexing back pressure. However, in this topology, the independence of hosts, services, and probes allows each component to be scaled independently, eliminating

resource contention inherent to the consolidated model.

Traffic generation in this setup differs from that of Architecture 1. Instead of relying solely on synthetic host activity, the evaluation employs public packet-capture traces replayed with tcpreplay[1]. These traces, ranging from 14,261 to 791,615 packets, yield a substantially higher and more heterogeneous traffic load, creating realistic stress conditions to validate probe performance, forwarding stability, and indexing throughput under workloads approaching production scale.

Operationally, this segmented architecture represents a significant step toward realism while retaining the reproducibility and controllability of a containerized environment. The use of proxy bridges introduces capture points independent of the endpoints, enabling controlled experiments on monitoring coverage, timing fidelity, and probe scalability across multiple subnets. This segmentation directly reflects common enterprise deployment practices and establishes a stronger foundation for evaluating the trade-offs of distributed network intrusion detection in Kubernetes-based environments.

4.4 Central Pipeline Implementation

The central node implements the complete data plane for ingestion, transformation, indexing, and visualization. It consolidates all analytical services of the architecture defined in Chapter 3 into two coordinated Docker Compose stacks deployed on the same VPS: a messaging and processing stack (Kafka, ZooKeeper, Logstash) and an analytics and visualization stack (OpenSearch nodes, OpenSearch Dashboards, Grafana). Both operate on isolated bridge networks, with only selected entry points published through Traefik for controlled external access. This design ensures modularity, reproducibility, and TLS protected communication between all components.

[1]Tcpreplay is a suite of free Open Source utilities for editing and replaying previously captured network traffic. `https://tcpreplay.appneta.com/`

4.4.1 Messaging and Stream Processing Layer

Kafka and ZooKeeper are deployed on a dedicated network kafka-net. Two listener profiles are configured for Kafka: an internal listener for intra-stack traffic (port 9092) and an external listener (port 9094) with an advertised address for client access. External connections are routed by Traefik using a TCP router in passthrough mode with HostSNI(*), preserving end-to-end TLS encryption. Internal broker traffic is likewise encrypted, and ZooKeeper client sessions are also secured with TLS. Logstash is attached to the same network and consumes from the Kafka topic *nids-logs* using the Kafka input plugin.[2] The processing pipeline performs a minimal yet consistent transformation sequence before indexing:

1. **Ingestion.** Events produced by the probes are consumed from Kafka and decoded as structured JSON records.

2. **Event time normalization.** A canonical timestamp is derived from each record and standardized, ensuring consistent temporal ordering across event types.

3. **Schema harmonization.** Core networking attributes (endpoints, ports, protocol, category) are mapped to a common schema and redundant payloads pruned.

4. **Identity and deduplication.** Each record receives a deterministic UUID and a probe label, allowing idempotent indexing and reliable filtering by probe.

5. **Output.** Records are written to OpenSearch via HTTPS using the logstash-output-opensearch plugin,[3] targeting daily indices to facilitate shard rotation and retention.

This configuration ensures event-time integrity, idempotent writes, and uniform enrichment across all probes. HTTPS and basic authentication are enabled, while certificate verification is disabled for this experimental setup.

[2] `https://www.elastic.co/docs/reference/logstash/plugins/plugins-inputs-kafka`
[3] `https://github.com/opensearch-project/logstash-output-opensearch`

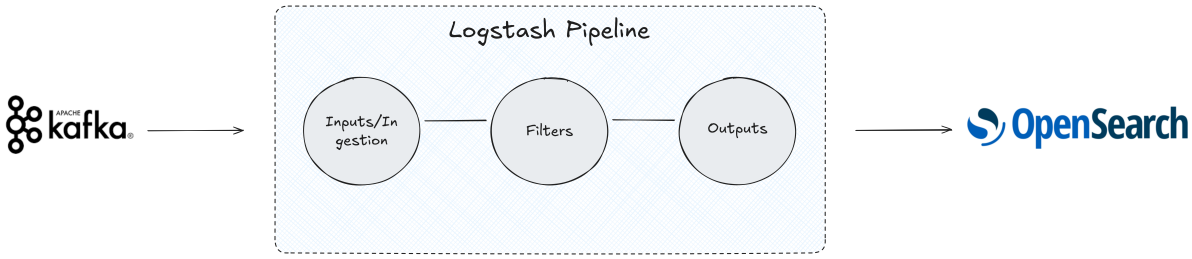

Figure 4.1: Simplified overview of the Logstash processing pipeline linking Kafka ingestion to OpenSearch indexing.

The diagram in Figure 4.1 summarizes this flow, showing how events are consumed from Kafka, processed sequentially through input, filter, and output stages, and finally indexed into OpenSearch with the applied normalization and enrichment logic.

4.4.2 Storage and Indexing Layer

OpenSearch operates as a secure two-node cluster with TLS enabled for both REST and inter-node transport, ensuring confidentiality and integrity of indexed events. Indices are rolled daily following the pattern suricata-kafka-{YYYY.MM.dd}, with replica shards providing redundancy and improved query throughput. JVM heap sizes are tuned to container resource limits to minimize garbage-collection pauses under bursty loads.

In practice, the index pattern suricata-kafka-* serves as the unified query base for both Grafana and OpenSearch Dashboards, enabling consistent cross-tool exploration. Figure 4.2 shows an exploratory view in OpenSearch Dashboards (Discover) over this index pattern. The upper panel displays a time histogram of @timestamp values, revealing event burstiness during replay periods. The lower document table exposes the normalized schema generated by the Logstash pipeline: canonical network fields and categories (e.g., event.kind=event, event.category=network); helper fields for analytics such as kubernetes.pod_name, event_type_flat, src_ip_flat, and dest_ip_flat; and flow-level attributes (e.g., app_proto, [flow].pkts_toclient). The per-record UUID (logstash_uuid)

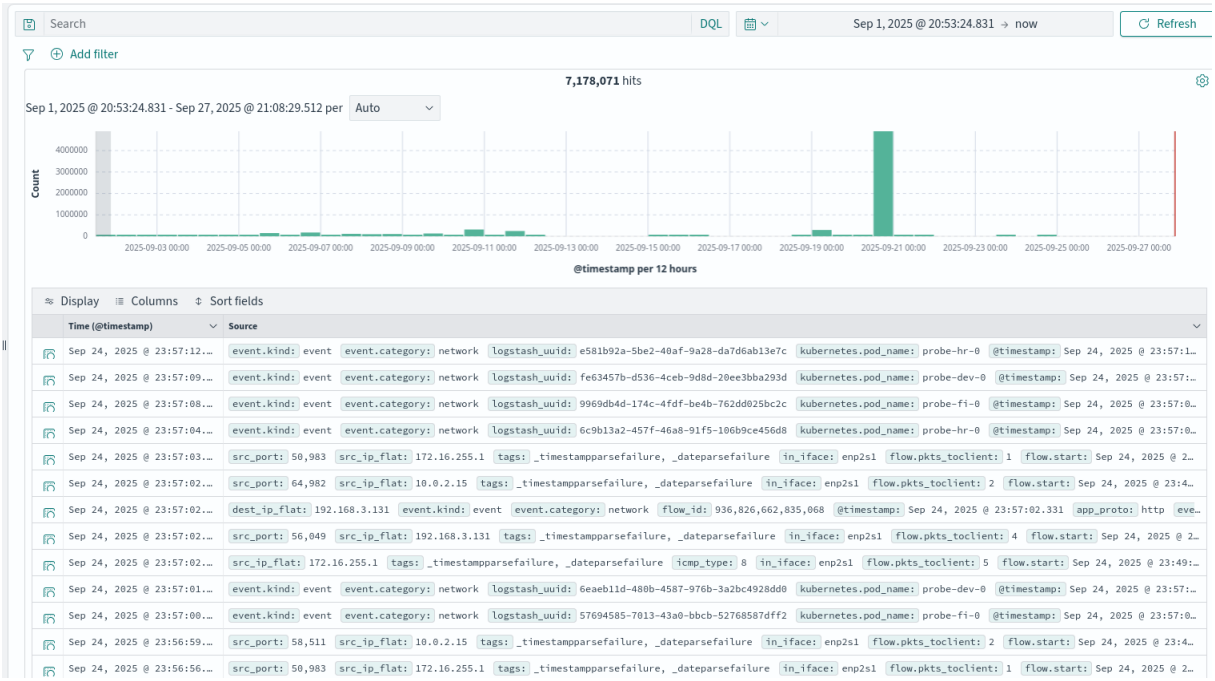

Figure 4.2: OpenSearch Dashboards view of the indexed Suricata events (suricata-kafka-*).

facilitates traceability and confirms idempotent indexing. Together, these features demonstrate that the indexing layer preserves event-time semantics and delivers an analysis-ready schema.

4.4.3 Visualization and Access Layer

Grafana connects directly to OpenSearch via HTTPS and serves as the main visualization interface for flow, alert, and protocol metrics. Query variables bind to normalized fields such as kubernetes.pod_name and event_type_flat, enabling per-probe comparisons and cross-namespace analysis. Figure 4.3 illustrates the "Probe Overview" dashboard. The upper panel plots time series of flow events across probes (probe-hr-0, probe-dev-0, probe-fi-0), exposing activity spikes aligned with traffic replays. The lower-left table lists the top alert signatures for the selected probe, while the lower-right chart summarizes alert categories by severity, providing an immediate assessment of the active risk profile. These

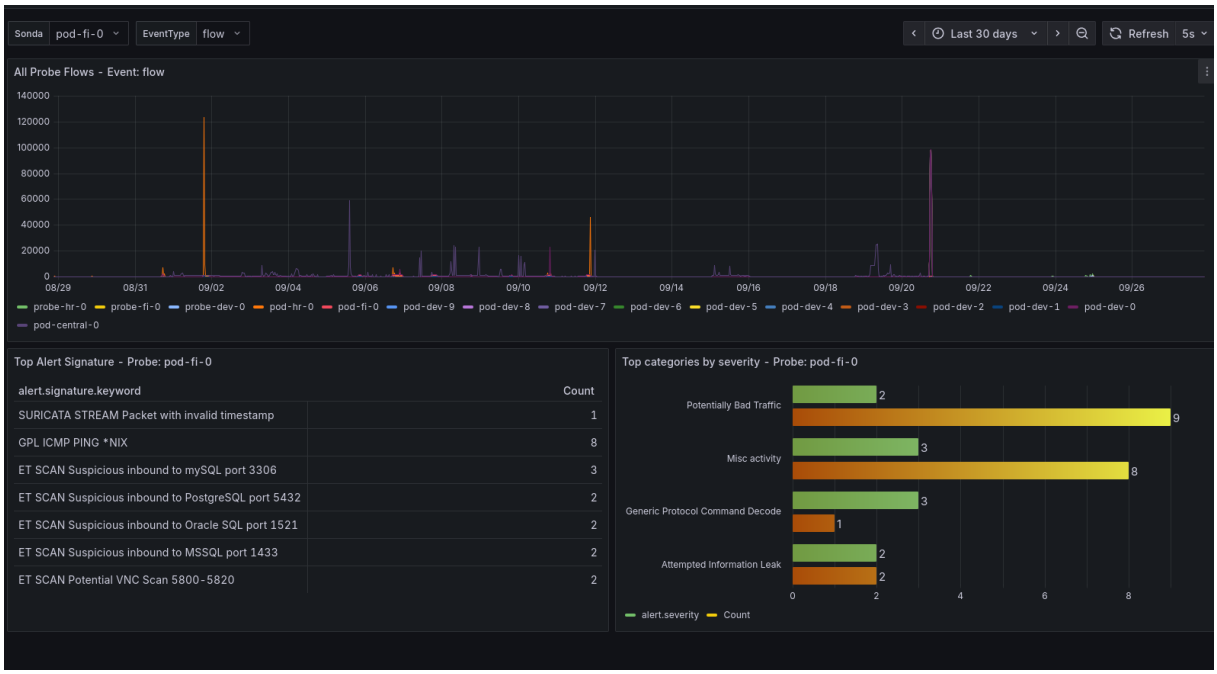

Figure 4.3: Grafana “Probe Overview” dashboard visualizing cross-probe flow and alert metrics.

dashboards confirm that the ingestion pipeline’s normalized schema supports interactive triage and comparative analytics across probes.

Service exposure and access control are centralized in Traefik. Kafka’s external listener is published through a TCP router with a HostSNI matcher, while HTTP services (Grafana and OpenSearch Dashboards) are routed behind HTTPS with domain-based termination. This configuration maintains a strict separation between private east–west traffic on internal bridge networks and controlled north–south access, preserving both security and operational clarity.[4]

The complete implementation integrates the distributed probes, messaging middleware, and central analytics services into a unified and reproducible environment. This setup establishes the operational foundation required for the experimental evaluation, where the performance, scalability, and detection effectiveness of both topologies are systematically assessed in Chapter 5.

[4] `https://doc.traefik.io/traefik/routing/routers/`

Chapter 5

Evaluation and Discussion

This chapter reports the empirical evaluation of the proposed multi–probe monitoring platform. The experiments were designed to validate the end–to–end pipeline, assess operational limits under increasing probe counts, and verify that detection outputs are actionable through the dashboards. The tests were carried out over the two simulation architectures presented earlier, keeping the central analytics stack unchanged in order to isolate the influence of topology on capture, transport, and indexing behavior. The pipeline components and orchestration choices referenced below follow the design in Chapter 3.

5.1 Goals and Method

The evaluation pursued four goals: to validate end-to-end pipeline correctness from probe to dashboards; to assess the scalability of the probe layer under increasing sensor counts; to examine the robustness of the central data plane under bursty workloads; and to confirm that detection outputs are actionable and navigable in dashboards.

The measurements collected per run included: packet emission time; capture time at the Suricata probe; indexing time in OpenSearch; and CPU/memory usage of probes (Suricata + Fluent Bit), the messaging layer (Kafka, Logstash), and the central node (OpenSearch nodes). Two traffic replays were used (smallFlows.pcap & bigFlows.pcap)

and, for adversarial validation, on-demand scan traffic was generated from a Kali Linux host. These procedures and datasets mirror the test summaries reported during iterative trials.

5.2 Test Environment

All experiments were conducted in a virtualized laboratory. The central services ran on a VPS and were deployed as two independent Docker Compose projects: a messaging stack (Apache Kafka, ZooKeeper, Logstash) and an analytics stack (two OpenSearch nodes, OpenSearch Dashboards, Grafana, Traefik). Each stack used its own bridge network; only selected entry points were exposed by Traefik over TLS, with HTTPS for Grafana/OpenSearch Dashboards and TCP passthrough for Kafka. The OpenSearch cluster stored daily time-based indices following the pattern suricata-kafka-YYYY.MM.dd.

Probes executed on Ubuntu Server VMs. Architecture 1 used K3s; Architecture 2 used MicroK8s with Multus and a proxy-bridge to mirror departmental traffic toward the probe pod. Each probe combined Suricata (EVE JSON) and Fluent Bit (forwarder), sharing a local volume for log exchange. Fluent Bit enriched events with pod/namespace metadata and produced to Kafka over TLS. Traffic was generated by a mix of synthetic activity (HTTP/ICMP/TCP checks), adversarial scans (Kali), and PCAP replays via tcpreplay , providing both steady baselines and bursty workloads.

Resource sizing evolved with scale. Probe-side VMs were increased when control-plane stability required it (e.g., from 4 GB/2 vCPU to 6 GB/4 vCPU and later to 12 GB/6 vCPU). The central VPS initially had 2,vCPU/8,GB RAM and hosted a single-broker Kafka topic (nids-logs) consumed by Logstash and indexed in OpenSearch. This single-broker, single-consumer design is adequate for functional tests but, as discussed later, becomes the limiting factor under high fan-out. Security was enforced end-to-end (TLS on Kafka, Logstash→OpenSearch, and HTTPS on dashboards), with basic authentication for user access.

5.3 Results and Analysis

5.3.1 Architecture 1 - Baseline Validation

Architecture 1 consolidates hosts, application services, and the probe within a single pod that shares one network namespace. The aim is to validate the correctness of the end-to-end data path without introducing variability from multi-hop forwarding or mirroring. In this arrangement Suricata inspects traffic directly from the pod interface, writes structured events in EVE JSON to a shared volume, and Fluent Bit forwards these records to the central pipeline using Kafka as a buffering layer. Logstash consumes from Kafka, normalizes timestamps, maps core fields to a common schema, and inserts documents into time-partitioned indices in OpenSearch. Visualization queries issued by Grafana and OpenSearch Dashboards operate on the same corpus over HTTPS with authentication and TLS.

Under this configuration the pipeline behaved deterministically. Events produced by Suricata propagated to Kafka without back pressure and were consumed by Logstash with stable end-to-end latency. Timestamp normalization preserved event time ordering at index, and idempotent document identifiers avoided duplication during component restarts. Field mappings remained consistent across runs, enabling dashboard variables such as [kubernetes][pod_name] and event_type_flat to drive per-probe and per-family drill-down. Access endpoints for Grafana, OpenSearch, and the reverse proxy were validated with TLS and user authentication. Short replay bursts confirmed that producer and consumer components recovered cleanly after restarts, that the indexing layer did not generate duplicate documents, and that dashboard panels refreshed as expected.

This baseline removes confounding factors related to container networking, CNI behavior, proxy-bridge mirroring, and inter-pod routing, thereby establishing invariants for subsequent experiments. Event-time normalization, stable schema, idempotent writes, and reproducible dashboard bindings are verified here so that deviations observed later can be attributed to network placement and scale rather than parsing, transport, or

indexing defects. In practice this validation step is necessary for interpreting Architecture 2, where increased probe counts and segmented topologies stress the central data plane composed of Kafka, Logstash, and OpenSearch.

5.3.2 Architecture 2 - Segmented Multi-Pod Evaluation

Single–Probe Characterization

This experiment isolates the behavior of a single probe (Suricata + Fluent Bit) across twenty controlled replays alternating smallFlows.pcap and bigFlows.pcap. For each run, three timestamps were recorded: packet emission at the traffic generator (T_0), capture at the probe (T_1), and indexing availability in OpenSearch (T_2), from which the end-to-end latency $L_{\text{total}} = T_2 - T_0$ was derived. The dataset comprises $n = 20$ runs, with 13 using smallFlows.pcap and 7 using bigFlows.pcap, while probe CPU values ranged between 68 and 177 millicores and memory remained stable between 546 and 558 MiB. The median end-to-end latency across all runs was 0.959063 s and the 95th percentile 1.340178 s, with only one observation exceeding this threshold, namely run n°19 with bigFlows.pcap, where $L_{\text{total}} = 1.369938$ s with CPU = 68. A scatter analysis of L_{total} against CPU, shown in Figure 5.1, reveals that no harmful monotonic trend exists; higher instantaneous CPU values occasionally coincide with slightly lower end-to-end latency, but the effect is not significant and overall L_{total} remained bounded below 1.35 s in all runs. These results indicate that, under a single probe, the capture stage is not a bottleneck, since the dominant contribution to latency lies downstream in forwarding, message broker ingestion, and indexing. The figure further illustrates the dispersion of L_{total}, the CPU Probe values, and the identification of the single outlier in the dataset, thereby confirming that a single probe instance sustains the tested workloads for both smallFlows and bigFlows replays.

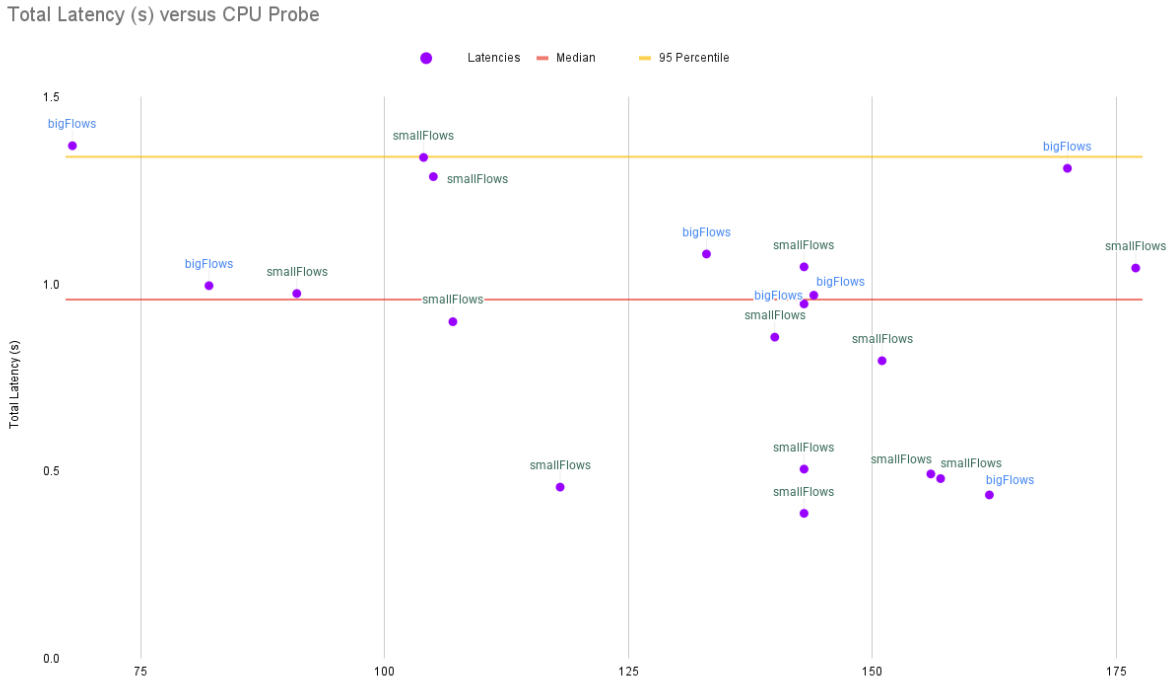

Figure 5.1: End-to-end latency (L_{total}, in seconds) versus probe CPU for the single-probe runs. Horizontal lines mark the median (0.959063 s) and the 95th percentile (1.340178 s).

Five–Probe Scaling Behavior

This experiment executes five probes concurrently (S0–S4) under alternating replays of smallFlows.pcap and bigFlows.pcap. For each run (id), the generator timestamp T_0, the capture timestamps at each probe $T_1^{(S_i)}$, and the indexing timestamps $T_2^{(S_i)}$ were recorded, from which the capture latency $L_{\text{probe}}^{(S_i)} = T_1^{(S_i)} - T_0$, the indexing latency $L_{\text{idx}}^{(S_i)} = T_2^{(S_i)} - T_1^{(S_i)}$, and the end-to-end latency $L_{\text{total}}^{(S_i)} = T_2^{(S_i)} - T_0$ were derived. Figure 5.2 illustrates the probe latency for S0 in comparison with the run-wise mean. The values remain short and stable, with only isolated excursions at specific id, a pattern confirmed by the deviation bars, indicating that capture is not the dominant source of delay. By contrast, Figure 5.3 shows the indexing latencies of all probes superimposed, together with the mean series. The five curves are nearly coincident, revealing that the downstream ingestion

and indexing stage is shared and evolves coherently across all probes, with systematic increases visible along the run sequence. Consequently, the total latency inherits the same shape: as shown in Figure 5.4, L_{total} across S0–S4 follows the indexing curves closely, with minimal per–probe divergence. Overall, these results demonstrate that under the five–probe setting the capture component remains bounded and homogeneous, while the system-level behavior is driven predominantly by the indexing path, which imposes the end-to-end delay structure in a synchronized manner across all sensors.

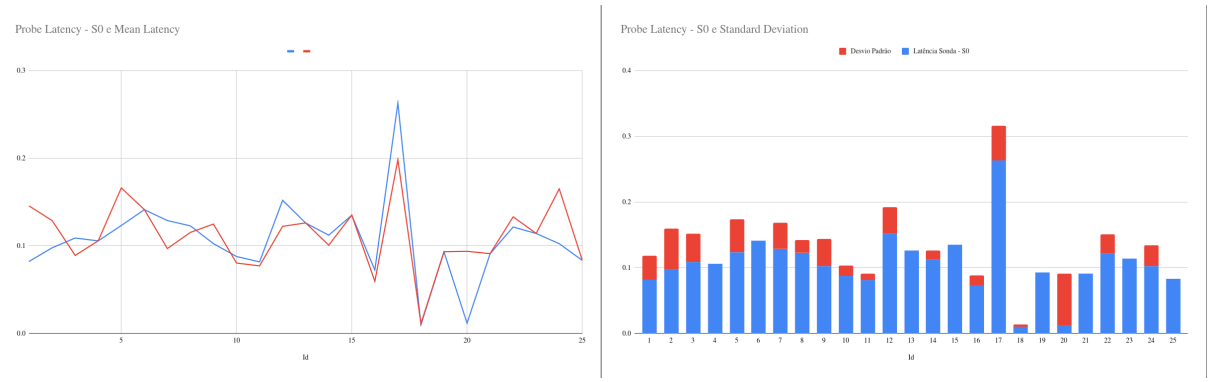

Figure 5.2: Probe latency L_{probe} for S0 compared with the run-wise mean, highlighting localized deviations but overall stability at the capture stage.

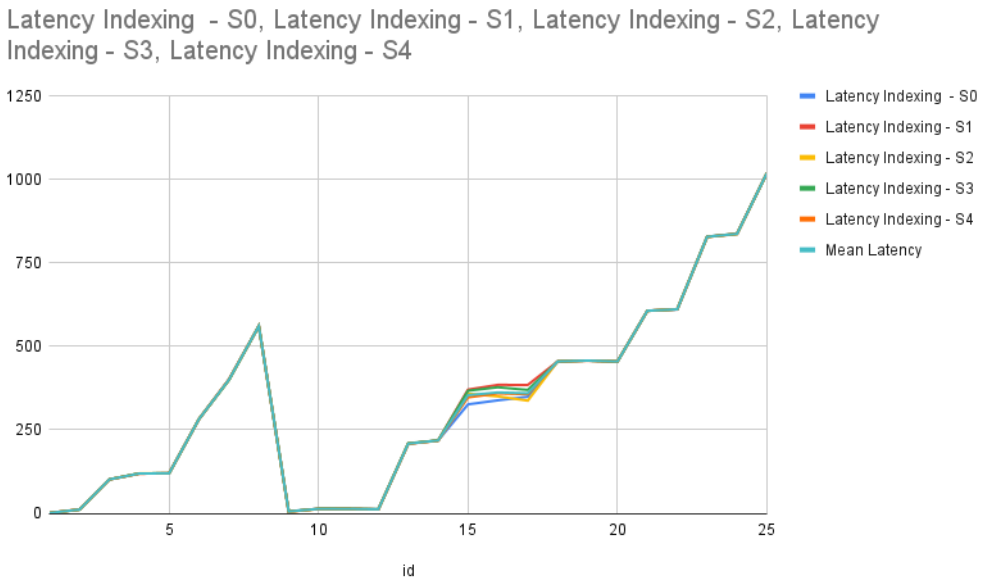

Figure 5.3: Indexing latency L_{idx} for probes S0–S4, overlaid with the mean. The curves overlap almost entirely, showing that indexing dominates and evolves coherently across all sensors.

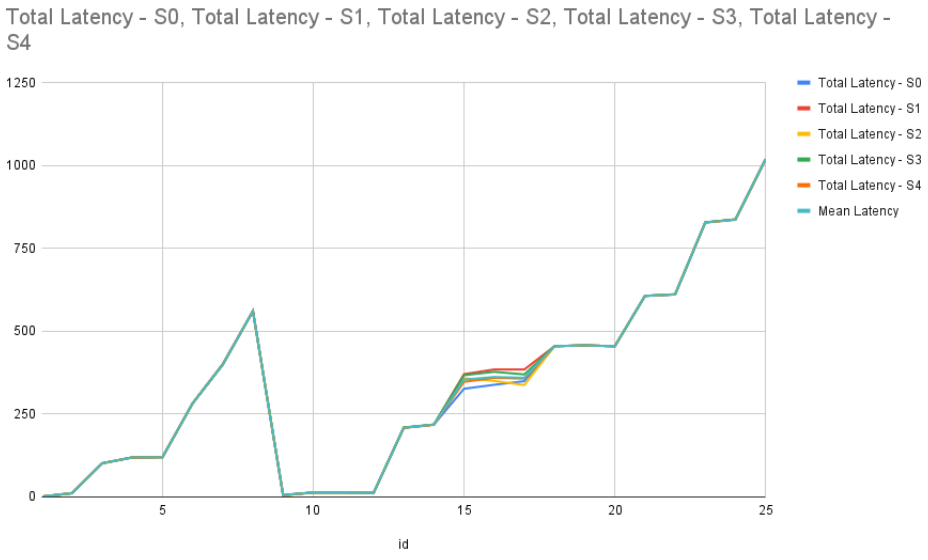

Figure 5.4: Total latency L_{total} for probes S0–S4 compared with the mean. The similarity with the indexing curves indicates that the end-to-end behavior is determined by the indexing stage.

Under five concurrent probes, the messaging layer (Kafka and Logstash) and the central store (two OpenSearch nodes) were monitored per run (id) for CPU and memory. The broker side remained lightweight on Kafka while Logstash and the store absorbed the bulk of the work. Kafka CPU stayed modest overall with a median of 3.69% (min 2.19%, max 7.52%), and Kafka memory presented a median of 217.8 MiB (min 153.9 MiB, max 453.5 MiB), indicating limited buffering pressure at the broker. Logstash showed sustained processing activity with a CPU median of 57.13% (min 24.35%, max 81.58%) and memory around the GiB range, median 965.8 MiB (min 846.1 MiB, max 1374.2 MiB), consistent with pipeline throughput and queue handling at the consumer stage. Figure 5.5 illustrates this behavior, where instantaneous oscillations in CPU are visible but remain consistently centered near the mean.

On the central store, OpenSearch Node 1 periodically peaked up to 97.79% CPU with a median of 46.30% (min 17.96%), and memory centered near the GiB scale at a median of 1129.47 MiB (min 964.2 MiB, max 1311.74 MiB). OpenSearch Node 2 complemented this pattern with a CPU median of 20.66% (min 6.56%, max 80.21%) and memory median of 1059.84 MiB (min 976 MiB, max 1261.57 MiB). Figures 5.6 and 5.7 show the temporal behavior of each node, confirming that Node 1 is consistently more loaded and occasionally saturates a core, while Node 2 runs at a lighter but more variable profile. The comparative view in Figure 5.8 further emphasizes that the two nodes share complementary utilization, but peaks in Node 1 dominate the overall profile of the cluster.

Taken together, these measurements confirm that, under the five–probe workload, Kafka acts as a lightweight ingress buffer while Logstash and the OpenSearch nodes bear the dominant processing and memory footprint; the store's CPU excursions explain the coherent rises observed in the indexing and end-to-end latencies, whereas the broker's footprint remains comparatively bounded.

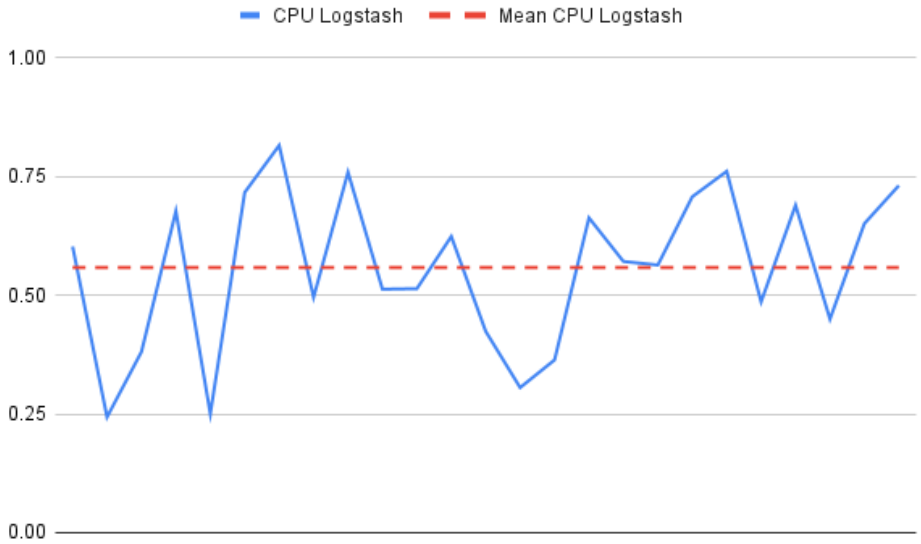

Figure 5.5: CPU usage of Logstash across runs with mean reference line.

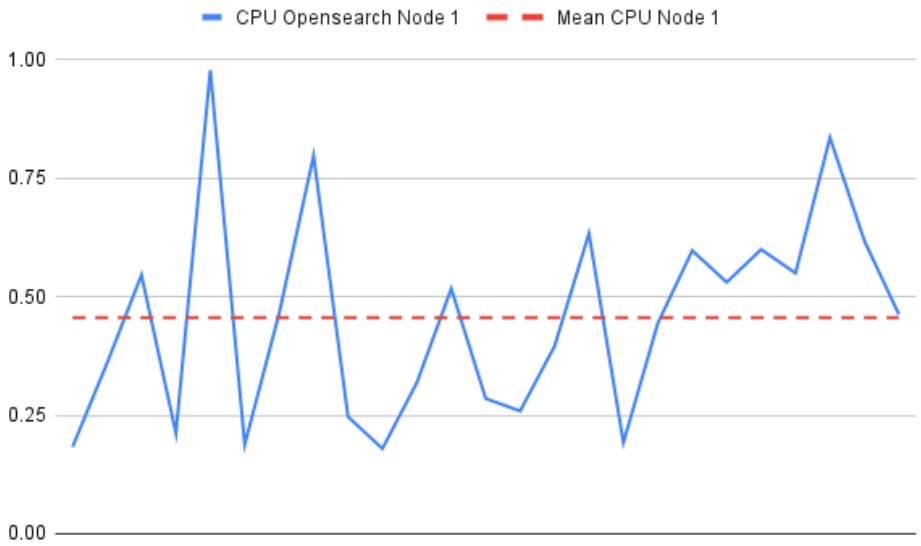

Figure 5.6: CPU usage of OpenSearch Node 1 with mean reference line.

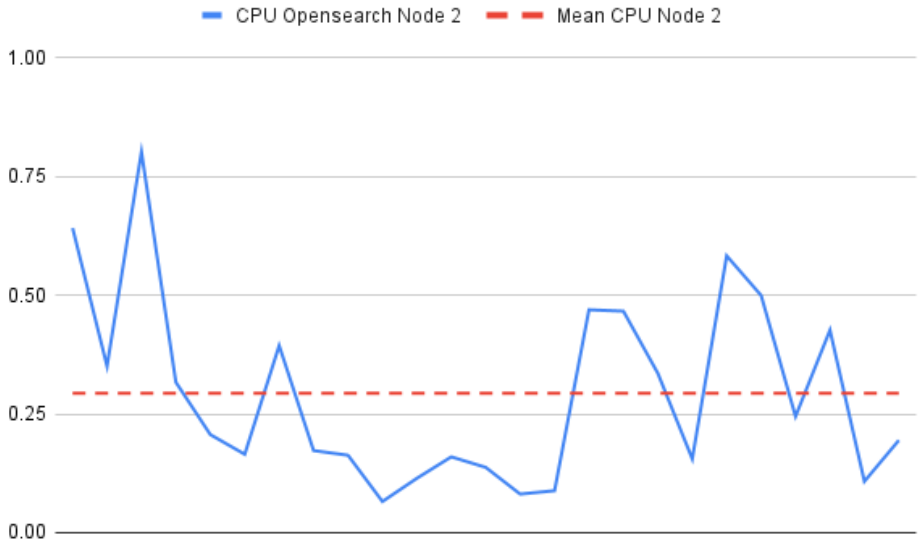

Figure 5.7: CPU usage of OpenSearch Node 2 with mean reference line.

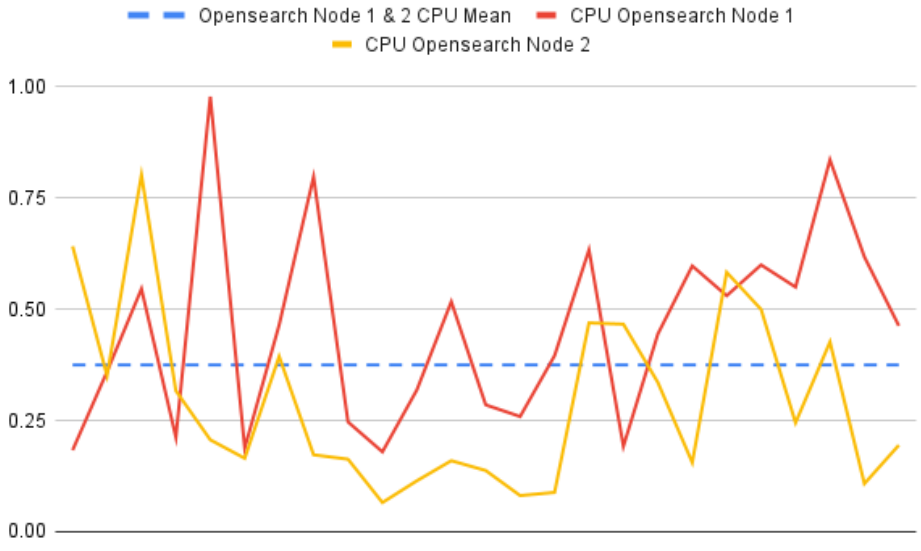

Figure 5.8: Comparative CPU usage of OpenSearch Nodes 1 and 2 with overall mean.

Ten-Probe Stress Test (Post-Ingestion Bottleneck)

While the single and five-probe trials confirmed the correctness and stability of the end-to-end pipeline, the execution with ten concurrent probes exposed a critical limitation. During this run, the probes successfully captured and forwarded traffic traces, alternating between smallFlows.pcap and bigFlows.pcap replays. The Kafka broker was able to absorb the high-frequency bursts without immediate data loss, demonstrating robust buffering capacity. However, the downstream stages revealed a persistent mismatch between ingestion and processing throughput.

According to the replay logs, the traffic source sustained an approximately constant rate of 10 Mbps, with a mean of 2,335.19 packets per second (pps) and a peak of 2,784.10 pps per cycle. Each long bigFlows.pcap segment lasted around 284 s, and the total active replay time covered roughly 8,190.9 s (≈2.27 h), with alternating bursts from smallFlows.pcap filling the gaps. During this period, the probes maintained stable performance, successfully exporting EVE JSON logs to the broker without significant delay.

Resource monitoring through docker stats over a 2.48 h window identified Logstash as the principal bottleneck of the pipeline. CPU utilization remained high and frequently saturated one full vCPU on the KVM-2 instance, as detailed in Table 5.1. Logstash sustained a median CPU load of 46.1%, with a 95th percentile of 70.1% and a peak of 146.27% (i.e., ∼1.46 vCPU on a 2 vCPU host). OpenSearch node 1 presented a median of 39.0%, P95 of 67.1%, and a peak of 96.93%, while node 2 reached 34.2%, P95 of 62.5%, and a peak of 115.96%. Kafka remained comparatively light, with a median of 7.22% and transient peaks near 64.42%, confirming headroom at the broker layer. In contrast, per-probe resource usage (Suricata + Fluent Bit) remained comparatively stable: across the ten probe pods, the mean CPU hovered around 166–171 m (95th percentile ∼209–213 m), with occasional spikes below 0.92 vCPU and mean memory usage near 520–555 MiB per pod.

The measurements confirmed a clear post-ingestion bottleneck. Kafka effectively decoupled producers from consumers, buffering bursts without loss, yet the single Logstash

consumer could not sustain the arrival rate produced by ten concurrent probes. Consequently, a backlog of unprocessed events accumulated in Kafka and persisted long after traffic generation stopped, extending the time-to-index beyond practical limits operationally surpassing twenty-four hours. OpenSearch exhibited elevated and occasionally saturated CPU usage during shard allocation and indexing spikes, but the root cause was upstream: the parsing and transformation throughput of Logstash remained below the sustained ingestion rate.

The ten-probe trial therefore demonstrated that scaling the capture edge without proportional scaling in the parsing and indexing layers leads to unbounded indexing latency under bursty workloads. To preserve near–real-time guarantees, the consumer tier must be parallelized (e.g., multiple Logstash consumers in a shared consumer group or increased pipeline worker threads), probe-side filtering refined to reduce per-event transformation cost, and parsing workloads distributed or simplified (e.g., schema-on-write strategies or direct ingestion of normalized EVE JSON fields). Absent these adjustments, the addition of probes merely amplifies broker backlog and deferred processing time, delaying analytic visibility across the stack.

Table 5.1 summarizes the CPU utilization metrics observed during this experiment, consolidating the median, 95th percentile, and peak usage across the core components of the central pipeline. The results clearly highlight Logstash as the dominant processing bottleneck, with both OpenSearch nodes exhibiting secondary stress under indexing load, while Kafka maintained substantial headroom. Together, these measurements provide a quantitative baseline for understanding the imbalance between ingestion and consumption capacity that later motivated the redesign and scaling of the consumer layer. The subsequent analysis therefore evaluates the impact of Kafka repartitioning and the deployment of multiple coordinated Logstash consumers to overcome the limitations identified in this trial.

Table 5.1: Central pipeline CPU utilization during the ten-probe run (docker stats).

Component	Median (%)	P95 (%)	**Peak (%)**	Samples
Logstash	46.08	70.08	**146.27**	1,168
OpenSearch node 1	39.03	67.12	**96.93**	1,168
OpenSearch node 2	34.23	62.52	**115.96**	1,168
Kafka	7.22	12.62	**64.42**	1,168

Broker and Consumer Scaling (Kafka + Logstash Repartitioning)

Following the post-ingestion bottleneck observed during the ten-probe trial, the central message broker layer was restructured to support horizontal scalability. In the original setup, Kafka, Zookeeper, and Logstash were deployed together in a single composition, with only one Logstash instance subscribing to the topic nids-logs. This arrangement confined message consumption to a single processing thread, which became saturated as concurrent probes increased. To overcome this limitation, Logstash was detached from the original stack and redeployed in an independent composition sharing the same virtual bridge network as Kafka and Zookeeper. This separation enabled multiple Logstash containers to operate concurrently as members of a shared consumer group.

The Kafka topic nids-logs, originally defined with one partition, was recreated with six partitions to allow concurrent reads across multiple consumers. Three Logstash instances were launched in parallel under the same consumer group, distributing partition ownership dynamically. This configuration aimed to improve throughput and shorten the backlog window that had previously extended well beyond one day under high probe concurrency.

Before the stress run, all six partitions were synchronized with negligible lag, confirming balanced assignment among the three consumers. However, during the subsequent 2h 30m replay, while the probes continuously emitted traffic traces alternating between smallFlows.pcap and bigFlows.pcap, the lag increased sharply. By the time all producers stopped, each partition exhibited between 5.2×10^5 and 6.2×10^5 pending records, totaling approximately 1.31×10^7 unprocessed messages across the topic. Even after traffic emission ceased, the backlog persisted for more than three hours until the system was

manually halted, indicating that consumption throughput remained below the cumulative arrival rate. Lag was measured using the kafka-consumer-groups –describe command on the target consumer group[1].

The sustained lag coincided with continuous resource saturation on the central node. Throughout the test, CPU utilization remained at or near 100% across both virtual cores, with no observable relief once the probes stopped. This pattern suggests that Logstash and OpenSearch continued parsing and indexing the residual backlog long after ingestion ended. Storage usage followed a similar trend, growing from roughly 60 GiB at the beginning of the test to nearly 84 GiB at its peak. This increase reflected the accumulation of Kafka log segments and temporary index files rather than new traffic production. Disk consumption fluctuated marginally as OpenSearch performed background merges, but the overall storage footprint did not decrease significantly during the test window.

While the reconfiguration enabled Kafka to parallelize message delivery effectively, the experiment demonstrated that logical scalability alone was insufficient to achieve real-time processing under the available resources. The combination of six partitions and three Logstash consumers provided functional redundancy and balanced partition offsets, but CPU and I/O limitations on the 2 vCPU, 8 GiB VPS constrained throughput. The backlog was eventually reduced when the load stopped, confirming pipeline integrity, yet the time required to drain the topic, over three hours post-replay, underscored the system's sensitivity to sustained burst traffic.

Overall, this experiment highlights that horizontal scaling through partitioning and consumer replication improves resilience but cannot compensate for limited computational capacity. In this configuration, the message broker architecture remained functionally correct but resource-bound. Achieving real-time performance would require additional hardware allocation, multi-broker Kafka replication for parallel disk access, or the offloading of parsing operations to a more distributed Logstash tier. The main operational parameters and observed performance indicators of this configuration are summarized in

[1]Kafka's consumer group tool for offset and lag inspection. `https://kafka.apache.org/documentation/#basic_ops_consumer_group`

Table 5.2, which consolidates the most relevant metrics, including CPU load, disk usage, and lag evolution, providing a concise overview of the system's post-scaling behavior.

Table 5.2: Summary of the reconfigured message broker and consumer setup.

Parameter	Configuration
Kafka brokers	1 (`broker.id=1`)
Kafka partitions	6 (`nids-logs`)
Logstash consumers	3 (`logstash-consumer-group`)
VPS resources	2 vCPU, 8 GiB RAM
Test duration (active probes)	≈2.5 h
Post-ingestion drain time	≈3.5 h
Initial disk usage	60 GiB
Peak disk usage	84 GiB
CPU load (average)	≈100% sustained
Lag evolution	≈ 13 million → partial drain (post-replay)

Active scan detection with Suricata

To validate signature based detection in the segmented topology, we executed a short adversarial exercise that targeted services in two namespaces while the proxy bridge mirrored traffic to the probe pods. The generator issued three families of reconnaissance and probing actions. First, ICMP echo requests to verify host liveness. Second, targeted TCP connects to the MySQL service on port 3306 to elicit banner and access response patterns. Third, a web reconnaissance phase using Nmap with version and script scans, whose HTTP requests carry the characteristic user agent strings produced by the Nmap Scripting Engine. All traffic was produced against the segmented endpoints, traversed the CNI fabric, and reached the probe via the mirror path, preserving the original flows between endpoints.

The Grafana panel filtered to event type alert for probe pod-dev-0 is shown in Figure 5.9. Two signatures aligned with the actions above. ET SCAN Suspicious inbound to MySQL port 3306 appeared with one hit, confirming that the database probe reached the service and was classified as a scan attempt. GPL ICMP PING NIX appeared with one hit, confirming liveness discovery. The category histogram attributed the alerts to Potentially Bad Traffic and Misc activity with counts consistent with the two signatures, and the time series displayed thin vertical spikes that matched the brief scan windows.

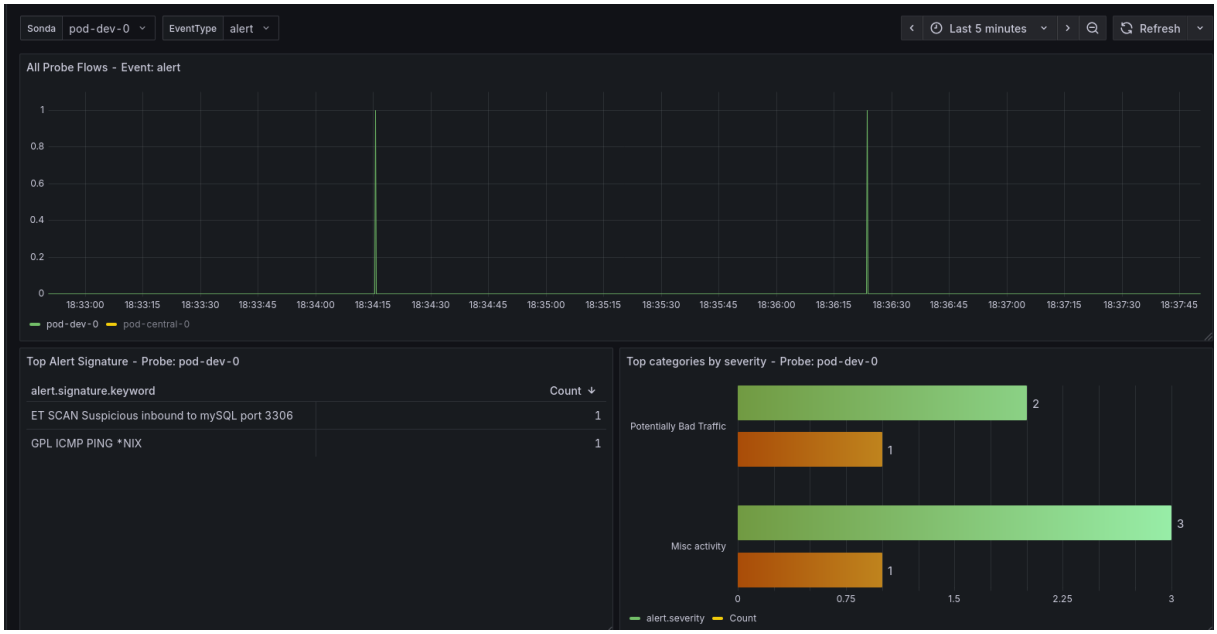

Figure 5.9: Alerts on pod-dev-0 during the scan. One MySQL 3306 hit and one ICMP hit. Categories map to Potentially Bad Traffic and Misc activity.

The flow view for pod-dev-0 during the same window is shown in Figure 5.10. The sparse spikes reflect short lived connections created by the probes. The Top Alert Signature table again lists ET SCAN Suspicious inbound to MySQL port 3306 and GPL ICMP PING NIX with one count each, while the severity chart places them in Potentially Bad Traffic and Misc activity.

Figure 5.10: Flows on pod-dev-0 in the same window, showing short spikes from ICMP and MySQL probes.

For the pod-hr-0 namespace, Figure 5.11 shows the richer pattern produced by HTTP based reconnaissance. ET SCAN Nmap Scripting Engine User-Agent Detected accumulated four hits and ET SCAN Possible Nmap User-Agent Observed accumulated four hits, which together accounted for eight events under the Web Application Attack category. The MySQL probe triggered ET SCAN Suspicious inbound to MySQL port 3306 with two hits, while liveness checks mapped to Potentially Bad Traffic. The separation of counts by probe shows that alerts were confined to the targeted segment and that the mirror path did not inject cross segment artifacts.

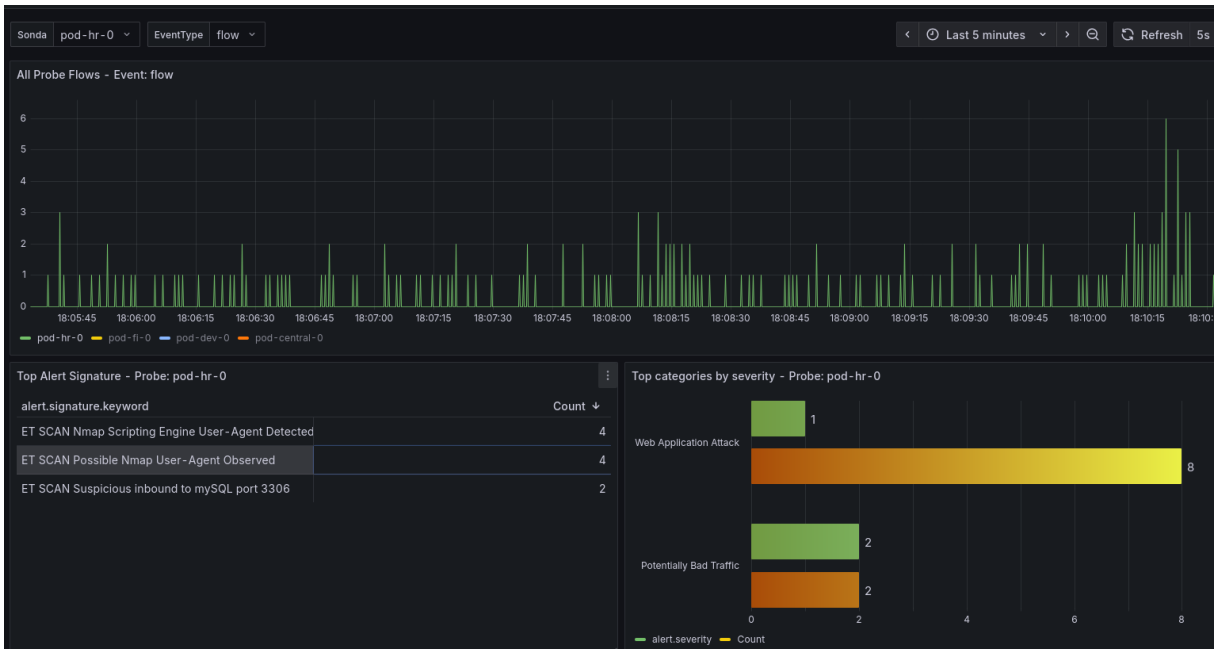

Figure 5.11: Flows on pod-hr-0 during HTTP reconnaissance. Nmap user agent alerts four plus four and MySQL 3306 two hits. Web Application Attack dominates.

These observations demonstrate three properties essential to the architecture. First, Suricata identified low volume reconnaissance typical of early kill chain stages, including ICMP sweeps, service specific probing of port 3306, and HTTP enumeration that reveals the Nmap script engine through its user agent string. Second, the proxy bridge supplied the probe with a faithful copy of on path traffic without altering the flows between endpoints, as only the targeted namespace accrued alerts. Third, alert volumes and categories correlated with the actions performed and provided immediate operator feedback during controlled exercises.

Binary detection accuracy of Suricata with default rules

Building on the previous subsection, Figures 5.9, 5.10 and 5.11 show that the segmented probes raised alerts for ICMP liveness checks, targeted MySQL connects and HTTP based reconnaissance. Over a broader window before the infrastructure failure, the system

recorded many repeated events with little class variety, dominated by stream anomalies and decode categories, as in Figure 5.12.

Figure 5.12: Cluster wide view showing repeated categories and volume spikes before the failure. Generic Protocol Command Decode dominates.

We measured the accuracy of Suricata with default ET Open rules in a binary setting, malicious versus benign. The attacker pod executed a suite that mixed benign HTTP and SSH requests with common offensive tools, while the Suricata probe recorded alerts filtered by the target IP. The analysis is rule based and does not train any model. For each execution the ground truth log provides start and end timestamps and the target ports. A detection is counted when Suricata emits at least one alert for the same target and port within a fixed slack window after the command finishes. During the campaign Suricata 8.0.1, updated with suricata-update 1.3.6, refreshed its rule set on 2025-10-21 17:34:29, reporting 45,873 signatures processed and 45,870 rules loaded, with most signatures inspecting application layer traffic. We used a 12 second slack and ran 396 executions in total, with 176 malicious and 220 benign runs.

Table 5.3: Binary confusion matrix for Suricata default rules (N=396).

	Predicted Positive	Predicted Negative
Actual Positive	123	53
Actual Negative	86	134

$$\text{Accuracy} = \frac{TP + TN}{TP + TN + FP + FN} = \frac{123 + 134}{396} = 0.649,$$
$$\text{Precision} = \frac{TP}{TP + FP} = \frac{123}{123 + 86} = 0.589,$$
$$\text{Recall} = \frac{TP}{TP + FN} = \frac{123}{123 + 53} = 0.699,$$
$$F_1 = \frac{2 \cdot \text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}} = 0.639.$$

With default rules, Suricata achieved precision 0.589 and recall 0.699 under a mixed malicious and benign workload, yielding an overall accuracy of 0.649. False positives were primarily associated with automated HTTP user agents and short TCP touches, while true positives came from HTTP reconnaissance, brute force attempts on SSH and service probes.

Malicious example:


```
{
  "tool": "nmap",
  "variant": "http_headers_default",
  "class": "malicious:nmap-http-80",
  "target": "10.10.0.23",
  "target_label": "binary",
  "ports": [80],
  "proto": "tcp",
  "timeout_s": 90,
  "cmd": "nmap -Pn -sT --script http-headers -p 80 10.10.0.23"
}
```

Benign example:

```
{
  "tool": "nc",
  "variant": "tcp_connect_http_nc",
  "class": "benign:http-80",
  "target": "10.10.0.23",
  "target_label": "binary",
  "ports": [80],
  "proto": "tcp",
  "timeout_s": 10,
  "cmd": "nc -z -w1 10.10.0.23 80"
}
```

The malicious example uses Nmap with the http-headers script on port 80. This command sends an HTTP request that includes the Nmap Scripting Engine user agent, which is often matched by Suricata rules for reconnaissance on web services. In the binary evaluation a positive is counted when at least one alert is emitted for the same target and port within the 12 second slack window after the command ends.

The benign example performs a short TCP connect with nc to port 80, a common availability check that does not carry an exploit payload. Under the default rule set it should not trigger alerts in normal conditions. If an alert appears in the same temporal window and port, the event is counted as a false positive for the benign class.

The ground truth fields in both snippets define the intended label, the target, the ports and the execution timestamps. The analyzer aligns alerts to these windows by destination address and port and decides detection using a simple rule, at least one alert in the window means positive for the binary setting. No statistical model is trained or applied, the procedure is deterministic and driven by the recorded timestamps and the Suricata alert stream.

5.4 Discussion and Cross-Architecture Insights

The comparison of Architectures 1 and 2 shows the complementary aspects that define the operation of the platform. Architecture 1 has the entire workload running in a single pod where the host, service, and probe share a network namespace. This design eliminates inter-pod variability, exposing the intrinsic behavior of the Suricata capture and indexing pipeline in isolation and under relatively controlled conditions. Latencies were stable across replays, the pipeline consisting of Suricata, Fluent Bit, and Kafka, exhibited deterministic timing, Logstash operated at a constant rate, and OpenSearch consistent indexing throughput, thereby indicating the single pod configuration as a baseline for validation of correctness and performance of isolated components.

In Architecture 2, the host, service, and probe are separated into different pods that are interconnected through a proxy bridge. This segmentation introduces a realistic network path between producers and the probe while maintaining the same downstream processing stack. This additional layer augments traffic diversity and reveals the extent to which the core pipeline is affected by parallel inputs. One, five, and ten probes activated in parallel during measurement showed that capture latency scaled uniformly within limits, while indexing latency scaled with near linearity as the number of probes increased. The delay in indexing and Logstash's CPU usage showed that, particularly when many concurrent probes were in play, the delay beyond the probes was dominated by the parsing and enrichment processes.

Over ten probe instances the messaging and storage layers demonstrated diverse behaviors. Kafka experienced bursts but retained headroom, while Logstash sustained CPU saturation and generated Kafka backlog. OpenSearch responded coherently to elevated input rates, temporarily increasing CPU resources during shard allocation and merges, without data loss and schema drift. These observations mark broker delivery guarantees, consumer throughput as the effective bottleneck, and indexing as the governor of long-term persistence.

In the segmented setup, we evaluated the detection accuracy of the default rules from

a performance standpoint. Suricata decisions were matched to ground truth by target and port within a fixed slack window, using a mixed suite of benign HTTP and SSH requests and standard offensive tools. Table 5.3 presents the confusion matrix summarizing 396 executions, producing a precision of 0.589, recall 0.699, and accuracy 0.649. True positives were concentrated on HTTP reconnaissance and probing service, while false positives mostly came from automated user agents and TCP touches typical of policy signatures. No statistical model was trained or tuned; detection represents the behavior of default signatures applied to the problem under realistic timing and traffic diversity.

Chapter 6

Conclusions And Future Work

This work focuses on the design, implementation, and evaluation of a distributed multi-probe architecture specifically developed for network traffic analysis, incorporating the principles of scalability, resilience, and efficient data collection and processing. This design was developed in response to the limitations of centralized monitoring systems, which experience performance and latency issues when under significant load. Through the use of a layered architecture that integrates containerized Suricata and Fluent Bit-based probes, a decoupled messaging layer using Apache Kafka, and a secure analytics stack of Logstash, OpenSearch, and Grafana, the thesis was able to successfully demonstrate a practically deployable system that normalizes and visualizes network telemetry in near-real time.

The experimental outcomes confirmed the accuracy and stability of the end-to-end pipeline. In both the consolidated single-pod and the segmented multi-pod deployments, the probes accurately maintained event-time consistency and schema uniformity while capturing and forwarding traffic to the central node. The platform scaled to multiple concurrent probes, with Kafka providing efficient temporal data stream buffering and OpenSearch rendering the data as coherent, searchable, and indexed. The performance analysis revealed the stability of the capture and forwarding layers, with the primary performance hindrance occurring at the Logstash consumer tier under scenarios of high probe concurrency. The architecture maintained fault tolerance and recoverability while

sustaining bursts of activity.

The initial goals were all accomplished successfully. The resultant system is modular, reproducible, and customizable to various network environments, accommodating horizontal scaling of the probes and the interchangeable addition of extra analytics subsystems. The probes were also validated as functioning in a distributed manner to accurately collect and transmit traffic data with a negligible impact on the network, while the central pipeline processes the data stream and outputs it within operational thresholds. Overall, the system offers a comprehensive and validated paradigm for distributed network traffic monitoring, addressing the existing disconnect between the theoretical underpinnings of the discipline and practical architectures ready for deployment. To anchor the results quantitatively, the end-to-end latency measurements confirmed near-real-time behavior in the 1 and 5 probes experiments, while the 10 probes configuration exposed a practical breakdown of real-time guarantees; the maximum Kafka consumer lag peaked at approximately 1.3×10^7 messages, and the OpenSearch CPU p95 on the central node approached 100% under the ten-probe workload.

6.1 Future Works

Building on the results obtained so far, the next steps include deploying the probes in a production network using SPAN or TAP to validate timing, packet loss, and end-to-end latency under real traffic; decomposing the central stack across dedicated hosts with a multi-broker Kafka cluster, a horizontally scaled Logstash consumer tier, and an expanded OpenSearch cluster to remove resource contention; introducing adaptive and predictive autoscaling for probes, broker partitions, consumer groups, and OpenSearch nodes driven by queue depth, consumer lag, and time to index; reducing per-event processing cost by moving part of the normalization to the edge where feasible and simplifying Logstash pipelines to lower CPU pressure; applying Index State Management (ILM) in OpenSearch with rollover, shrink or force-merge, and retention policies coupled with capacity planning

for shard sizing and merge budgets; defining observability targets and service level objectives for the ingestion path with explicit thresholds for lag and indexing latency, plus alerting and runbooks to trigger scale actions; and enriching detection by complementing Suricata with flow-centric telemetry such as Zeek and by adding a streaming analytics layer to compute features and aggregates before indexing, improving both signal quality and near real-time guarantees.

Bibliography

[1] P. Rajesh Kanna and P. Santhi, "Exploring the landscape of network security: A comparative analysis of attack detection strategies," *Journal of Ambient Intelligence and Humanized Computing*, pp. 1–18, 2024.

[2] J. Koumar, K. Hynek, T. Čejka, and P. Šiška, *Cesnet-timeseries24: Time series dataset for network traffic anomaly detection and forecasting*, 2024. arXiv: 2409.18874 [cs.LG]. [Online]. Available: https://arxiv.org/abs/2409.18874.

[3] X. Yuan, Y. Qiao, Z. Wei, *et al.*, *Diffusion models meet network management: Improving traffic matrix analysis with diffusion-based approach*, 2024. arXiv: 2411.19493 [cs.NI]. [Online]. Available: https://arxiv.org/abs/2411.19493.

[4] K. Sharma, M. Chaudhary, K. Yadav, and P. Thakur, "Anomaly detection in network traffic using deep learning," in *2023 International Conference on Recent Advances in Science and Engineering Technology (ICRASET)*, 2023, pp. 1–5. DOI: 10.1109/ICRASET59632.2023.10419951.

[5] C. Morariu and B. Stiller, "An open architecture for distributed ip traffic analysis (dita)," in *12th IFIP/IEEE International Symposium on Integrated Network Management (IM 2011) and Workshops*, 2011, pp. 952–957. DOI: 10.1109/INM.2011.5990528.

[6] C. Morariu and B. Stiller, "Distributed architecture for real-time traffic analysis," in *Mechanisms for Autonomous Management of Networks and Services*, B. Stiller and F. De Turck, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 171–174.

[7] A. Bhandari, S. Gautam, T. K. Koirala, and M. Ruhul Islam, "Packet sniffing and network traffic analysis using tcp—a new approach," in *Advances in Electronics, Communication and Computing: ETAEERE-2016*, Springer, 2018, pp. 273–280. [Online]. Available: https://link.springer.com/chapter/10.1007/978-981-10-4765-7_28.

[8] P. Asrodia and V. Sharma, "Network monitoring and analysis by packet sniffing method," *International Journal of Engineering Trends and Technology (IJETT)*, vol. 4, no. 5, pp. 2133–2135, 2013. [Online]. Available: https://www.ijettjournal.org/volume-4/issue-5/IJETT-V4I5P160.pdf.

[9] J. E. Mota Filho, *Análise de Tráfego em Redes TCP/IP: Utilize tcpdump na análise de tráfegos em qualquer sistema operacional*. Novatec Editora, 2013, ISBN: 978857 5223758.

[10] I. A. I. Diyeb, A. Saif, and N. A. Al-Shaibany, "Ethical network surveillance using packet sniffing tools: A comparative study," *International Journal of Computer Network and Information Security*, vol. 11, no. 7, p. 12, 2018. [Online]. Available: https://www.mecs-press.org/ijcnis/ijcnis-v10-n7/IJCNIS-V10-N7-2.pdf.

[11] P. Saxena and S. K. Sharma, "Analysis of network traffic by using packet sniffing tool: Wireshark," *Int. J. Adv. Res. Ideas Innov. Technol*, vol. 3, no. 6, pp. 804–808, 2017. [Online]. Available: https://www.ijariit.com/manuscripts/v3i6/V3I6-1369.pdf.

[12] A. Varanasi and P. Swathi, "Comparative study of packet sniffing tools for http network monitoring and analyzing," *International Journal of Science, Engineering and Computer Technology*, vol. 6, no. 12, p. 406, 2016. [Online]. Available: http://ijcset.net/docs/Volumes/volume6issue12/ijcset2016061202.pdf.

[13] R. Tuli, "Packet sniffing and sniffing detection," *International Journal of Innovations in Engineering and Technology*, vol. 16, no. 1, 2020. [Online]. Available: https://ijiet.com/wp-content/uploads/2020/05/4.pdf.

[14] H. K. Ravuri, M. T. Vega, J. van der Hooft, T. Wauters, and F. De Turck, "A scalable hierarchically distributed architecture for next-generation applications," *Journal of Network and Systems Management*, vol. 30, pp. 1–32, 2022.

[15] N. Naik, "Demystifying properties of distributed systems," in *2021 IEEE International Symposium on Systems Engineering (ISSE)*, 2021, pp. 1–8. DOI: 10.1109/ISSE51541.2021.9582515.

[16] A. Ledmi, H. Bendjenna, and S. M. Hemam, "Fault tolerance in distributed systems: A survey," in *2018 3rd International Conference on Pattern Analysis and Intelligent Systems (PAIS)*, 2018, pp. 1–5. DOI: 10.1109/PAIS.2018.8598484.

[17] D. Gogri, "Advanced and scalable real-time data analysis techniques for enhancing operational efficiency, fault tolerance, and performance optimization in distributed computing systems and architectures," *International Journal of Machine Intelligence for Smart Applications*, vol. 13, no. 12, pp. 46–70, 2023. [Online]. Available: https://dljournals.com/index.php/IJMISA/article/view/37.

[18] P. Jogalekar and M. Woodside, "Evaluating the scalability of distributed systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 11, no. 6, pp. 589–603, 2000. DOI: 10.1109/71.862209.

[19] B. Assiri and A. Sheneamer, "Fault tolerance in distributed systems using deep learning approaches," *PLOS ONE*, vol. 20, no. 1, pp. 1–24, Jan. 2025. DOI: 10.1371/journal.pone.0310657. [Online]. Available: https://doi.org/10.1371/journal.pone.0310657.

[20] E. K. Gyasi, D. Asamoah, E. O. Oppong, and S. O. Oppong, "The scalability metric based on cost-effectiveness in distributed systems," *International Journal of Applied Information Systems*, vol. 12, no. 15, pp. 1–10, 2018, ISSN: 2249-0868. DOI: 10.5120/ijais2018451773. [Online]. Available: https://www.ijais.org/archives/volume12/number15/1037-2018451773/.

[21] N. Naik, "Comprehending concurrency and consistency in distributed systems," in *2021 IEEE International Symposium on Systems Engineering (ISSE)*, 2021, pp. 1–6. DOI: 10.1109/ISSE51541.2021.9582518.

[22] R. Caceres, N. Duffield, A. Feldmann, *et al.*, "Measurement and analysis of ip network usage and behavior," *IEEE Communications Magazine*, vol. 38, no. 5, pp. 144–151, 2000. DOI: 10.1109/35.841839.

[23] R. Subramanyan, J. Miguel-Alonso, and J. A. B. Fortes, "A scalable snmp-based distributed monitoring system for heterogeneous network computing," *ACM/IEEE SC 2000 Conference (SC'00)*, pp. 14–14, 2000. [Online]. Available: https://api.semanticscholar.org/CorpusID:2246666.

[24] A. Buga, "A scalable monitoring solution for large-scale distributed systems," in *Computer Aided Systems Theory – EUROCAST 2015*, R. Moreno-Díaz, F. Pichler, and A. Quesada-Arencibia, Eds., Cham: Springer International Publishing, 2015, pp. 219–227, ISBN: 978-3-319-27340-2.

[25] T. Newhall, J. Lībeks, R. Greenwood, and J. Knerr, "{Peermon}: A {peer-to-peer} network monitoring system," in *24th Large Installation System Administration Conference (LISA 10)*, 2010.

[26] Y. Bejerano and R. Rastogi, "Robust monitoring of link delays and faults in ip networks," in *IEEE INFOCOM 2003. Twenty-second Annual Joint Conference of the IEEE Computer and Communications Societies (IEEE Cat. No.03CH37428)*, vol. 1, 2003, 134–144 vol.1. DOI: 10.1109/INFCOM.2003.1208666.

[27] A. S. Shaffi and M. Al-Obaidy, "Managing network components using snmp," *International Journal*, vol. 2, no. 3, pp. 2305–1493, 2013.

[28] M. Fedor, M. L. Schoffstall, J. R. Davin, and D. J. D. Case, *Simple Network Management Protocol (SNMP)*, RFC 1157, May 1990. DOI: 10.17487/RFC1157. [Online]. Available: https://www.rfc-editor.org/info/rfc1157.

[29] M. Thottan, L. Li, B. Yao, V. Mirrokni, and S. Paul, "Distributed network monitoring for evolving ip networks," in *24th International Conference on Distributed Computing Systems, 2004. Proceedings.*, 2004, pp. 712–719. DOI: 10.1109/ICDCS.2004.1281639.

[30] I. Bermudez, S. Traverso, M. Munafò, and M. Mellia, "A distributed architecture for the monitoring of clouds and cdns: Applications to amazon aws," *IEEE Transactions on Network and Service Management*, vol. 11, no. 4, pp. 516–529, 2014. DOI: 10.1109/TNSM.2014.2362357.

[31] R. Kang, Z. Zhou, J. Liu, Z. Zhou, and S. Xu, "Distributed monitoring system for microservices-based iot middleware system," in *Cloud Computing and Security*, X. Sun, Z. Pan, and E. Bertino, Eds., Cham: Springer International Publishing, 2018, pp. 467–477.

[32] A. Depari, P. Ferrari, A. Flammini, D. Marioli, A. Taroni, *et al.*, "Multi-probe measurement instrument for real-time ethernet networks," in *Proc. of IEEE WFCS2006*, 2006, pp. 313–320.

[33] V. Mohan, Y. J. Reddy, K. Kalpana, *et al.*, "Active and passive network measurements: A survey," *International Journal of Computer Science and Information Technologies*, vol. 2, no. 4, pp. 1372–1385, 2011.

[34] A. Papadogiannakis, G. Vasiliadis, D. Antoniades, M. Polychronakis, and E. P. Markatos, "Improving the performance of passive network monitoring applications with memory locality enhancements," *Computer Communications*, vol. 35, no. 1, pp. 129–140, 2012, ISSN: 0140-3664. DOI: https://doi.org/10.1016/j.comcom.2011.08.003. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0140366411002404.

[35] C. Gandhi, G. Suri, R. P. Golyan, P. Saxena, and B. K. Saxena, "Packet sniffer–a comparative study," *International Journal of Computer Networks and Communications Security*, vol. 2, no. 5, pp. 179–187, 2014. [Online]. Available: https://www.academia.edu/download/76746987/p6_2-5.pdf.

[36] P. Goyal and A. Goyal, "Comparative study of two most popular packet sniffing tools- tcpdump and wireshark," *International Conference on Computational Intelligence and Communication Networks*, pp. 77–81, 2017. DOI: 10.1109/CICN.2017.19. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/8319360.

[37] P. Asrodia and H. Patel, "Network traffic analysis using packet sniffer," *International journal of engineering research and applications*, vol. 2, no. 3, pp. 854–856, 2012. [Online]. Available: https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=398d5c1ca4ac9943ef967722b3a57c2cfe1531fd.

[38] W. Park and S. Ahn, "Performance comparison and detection analysis in snort and suricata environment," *Wireless Personal Communications*, vol. 94, pp. 241–252, 2017.

[39] R. Fekolkin, "Intrusion detection and prevention systems: Overview of snort and suricata," *Luleå University of Technology*, Jan. 2015.

[40] O. Negoita and M. Carabas, "Enhanced security using elasticsearch and machine learning," in *Intelligent Computing: Proceedings of the 2020 Computing Conference, Volume 3*, Springer, 2020, pp. 244–254.

[41] A. Tasneem, A. Kumar, and S. Sharma, "Intrusion detection prevention system using snort," *International Journal of Computer Applications*, vol. 181, no. 32, pp. 21–24, 2018.

[42] A. Papadogiannakis, G. Vasiliadis, D. Antoniades, M. Polychronakis, and E. P. Markatos, "Improving the performance of passive network monitoring applications with memory locality enhancements," *Computer Communications*, vol. 35, no. 1, pp. 129–140, 2012.

[43] J. Vestin, A. Kassler, D. Bhamare, K.-J. Grinnemo, J.-O. Andersson, and G. Pongracz, *Programmable event detection for in-band network telemetry*, 2019. arXiv: 1909.12101 [cs.NI]. [Online]. Available: https://arxiv.org/abs/1909.12101.

[44] F. Alessi, A. Tundo, M. Mobilio, O. Riganelli, and L. Mariani, *Reprobe: An architecture for reconfigurable and adaptive probes*, 2024. arXiv: 2403.12703 [cs.SE]. [Online]. Available: https://arxiv.org/abs/2403.12703.

[45] A. DrAlconzo, I. Drago, A. Morichetta, M. Mellia, and P. Casas, "A survey on big data for network traffic monitoring and analysis," *IEEE Transactions on Network and Service Management*, vol. PP, pp. 1–1, Aug. 2019. DOI: 10.1109/TNSM.2019.2933358.

[46] A. Baer, P. Casas, A. D'Alconzo, *et al.*, "Dbstream: A holistic approach to large-scale network traffic monitoring and analysis," *Computer Networks*, vol. 107, pp. 5–19, 2016.

[47] N. G. M. Santos, "Study, implementation and configuration of an open source ids," M.S. thesis, Universidade da Maia, 2021.

[48] J. Kreps, N. Narkhede, J. Rao, *et al.*, "Kafka: A distributed messaging system for log processing," in *Proceedings of the NetDB*, Athens, Greece, vol. 11, 2011, pp. 1–7.

[49] M. Kajiura and J. Nakamura, *Practical performance of a distributed processing framework for machine-learning-based nids*, 2024. arXiv: 2405.13066 [cs.CR]. [Online]. Available: https://arxiv.org/abs/2405.13066.

[50] A. Zamini, "From gateway to dashboard: A secure microservices architecture for data provisioning to odoo erp," M.S. thesis, University of Padova, 2024/2025. [Online]. Available: https://thesis.unipd.it/handle/20.500.12608/91820.

[51] D. Gorasiya, "Comparison of open-source data stream processing engines: Spark streaming, flink and storm," National College of Ireland, Tech. Rep., Sep. 2019. DOI: 10.13140/RG.2.2.16747.49440.

[52] M. Čavojský, M. Hasin, and G. Bugár, "An analytical framework for data collection and analysis in ip network," *Acta Electrotechnica et Informatica*, vol. 23, no. 3, pp. 10–15, 2023. DOI: 10.2478/aei-2023-0012. [Online]. Available: https://doi.org/10.2478/aei-2023-0012.

[53] V. Tymoshchuk, M. Vorona, A. Dolinsky, V. Shymanska, and D. Tymoshchuk, "Security onion platform as a tool for detecting and analysing cyber threats," *Logos Science*, 2024. DOI: 10.36074/logos-13.12.2024.048.

[54] A. K. Jakkani, "Real-time network traffic analysis and anomaly detection to enhance network security and performance: Machine learning approaches," *Journal of Electronics, Computer Networking and Applied Mathematics*, vol. 4, no. 4, pp. 32–44, 2024. DOI: 10.55529/jecnam.44.32.44.

[55] Z. S. Younus and M. Alanezi, "A survey on network security monitoring: Tools and functionalities," *Mustansiriyah Journal of Pure and Applied Sciences*, vol. 3, no. 3, pp. 55–86, 2024. DOI: 10.47831/mjpia.v3i3.33.

[56] V.-A. Zamfir, M. Carabas, C. Carabas, and N. Tapus, "Systems monitoring and big data analysis using the elasticsearch system," in *22nd International Conference on Control Systems and Computer Science (CSCS)*, 2019, pp. 188–197. DOI: 10.1109/CEC5.2018.00039.

[57] W. Chen, F. Guo, and F.-Y. Wang, "A survey of traffic data visualization," *IEEE Transactions on Intelligent Transportation Systems*, vol. 16, no. 6, pp. 2970–2984, 2015.

[58] L. Shi, Q. Liao, X. Sun, Y. Chen, and C. Lin, "Scalable network traffic visualization using compressed graphs," *Journal of Network and Computer Applications*, vol. 221, p. 103 872, 2024.

[59] L. Manases and D. Zinca, "Automation of network traffic monitoring using docker images of snort3, grafana and a custom api," in *2022 21st RoEduNet Conference: Networking in Education and Research (RoEduNet)*, 2022, pp. 1–4. DOI: 10.1109/RoEduNet57163.2022.9921063.

[60] D. Zhongxing, "Network traffic monitoring algorithm based on big data analysis," *Academic Journal of Computing & Information Science*, vol. 6, no. 5, pp. 56–67, 2023. DOI: 10.25236/AJCIS.2023.060508.

[61] G. Calderon, G. del Campo, E. Saavedra, and A. Santamaría, "Monitoring framework for the performance evaluation of an iot platform with elasticsearch and apache kafka," *Information systems frontiers*, vol. 26, no. 6, pp. 2373–2389, 2024.

[62] C. Misa, "Traffic monitoring using programmable switch hardware for in-network aggregation," University of Oregon, Department of Computer and Information Science, Technical Report AREA-202303, 2023. [Online]. Available: `https://www.cs.uoregon.edu/Reports/AREA-202303-Misa.pdf`.

[63] V. Serbanescu, F. Pop, V. Cristea, and G. Antoniu, "Architecture of distributed data aggregation service," in *2014 IEEE 28th International Conference on Advanced Information Networking and Applications*, 2014, pp. 727–734. DOI: `10.1109/AINA.2014.89`.

Appendix A

Central Node - Docker Compose

```
services:
  opensearch-node1:
    image: 'opensearchproject/opensearch:2.19.1'
    environment:
      - cluster.name=${OPENSEARCH_CLUSTER_NAME}
      - node.name=${OPENSEARCH_NODE1_NAME}
      - discovery.seed_hosts=${OPENSEARCH_SEED_HOSTS}
      - cluster.initial_cluster_manager_nodes=${
          OPENSEARCH_INITIAL_CLUSTER_MANAGER_NODES}
      - bootstrap.memory_lock=${BOOTSTRAP_MEMORY_LOCK}
      - OPENSEARCH_JAVA_OPTS=${OPENSEARCH_JAVA_OPTS}
      - OPENSEARCH_INITIAL_ADMIN_PASSWORD=${
          OPENSEARCH_INITIAL_ADMIN_PASSWORD}
    ulimits:
      memlock:
        soft: -1
        hard: -1
      nofile:
        soft: 65536
        hard: 65536
```

```
19      ports:
20        - '9200:9200'
21        - '9600:9600'
22      healthcheck:
23        test:
24          - CMD-SHELL
25          - 'curl -k -s https://localhost:9200/_cluster/health |
                grep -q ''"status":"green"\|"status":"yellow"'''
26        interval: 30s
27        timeout: 10s
28        retries: 5
29      volumes:
30        - 'opensearch-data1:/usr/share/opensearch/data'
31
32    opensearch-node2:
33      image: 'opensearchproject/opensearch:2.19.1'
34      environment:
35        - cluster.name=${OPENSEARCH_CLUSTER_NAME}
36        - node.name=${OPENSEARCH_NODE2_NAME}
37        - discovery.seed_hosts=${OPENSEARCH_SEED_HOSTS}
38        - cluster.initial_cluster_manager_nodes=${
             OPENSEARCH_INITIAL_CLUSTER_MANAGER_NODES}
39        - bootstrap.memory_lock=${BOOTSTRAP_MEMORY_LOCK}
40        - OPENSEARCH_JAVA_OPTS=${OPENSEARCH_JAVA_OPTS}
41        - OPENSEARCH_INITIAL_ADMIN_PASSWORD=${
             OPENSEARCH_INITIAL_ADMIN_PASSWORD}
42      labels:
43        - traefik.enable=${TRAEFIK_ENABLE}
44        - traefik.http.routers.opensearch.rule=Host(`${
             OPENSEARCH_PUBLIC_HOST}`)
```

```
45        - traefik.http.routers.opensearch.entryPoints=${
              TRAEFIK_ENTRYPOINTS}
46        - traefik.http.routers.opensearch.tls=${TRAEFIK_TLS}
47        - traefik.http.services.opensearch.loadbalancer.server.port
              =${OPENSEARCH_HTTP_PORT}
48      ulimits:
49        memlock:
50          soft: -1
51          hard: -1
52        nofile:
53          soft: 65536
54          hard: 65536
55      healthcheck:
56        test:
57          - CMD-SHELL
58          - 'curl -k -s https://localhost:9200/_cluster/health |
                grep -q ''"status":"green"\|"status":"yellow"'''
59        interval: 30s
60        timeout: 10s
61        retries: 5
62      volumes:
63        - 'opensearch-data2:/usr/share/opensearch/data'
64
65    grafana:
66      image: 'grafana/grafana:latest'
67      ports:
68        - '3000:3000'
69      volumes:
70        - 'grafana-storage:/var/lib/grafana'
71      environment:
```

```
72        - GF_SECURITY_ADMIN_PASSWORD=${GRAFANA_ADMIN_PASSWORD}
73      labels:
74        - traefik.enable=${TRAEFIK_ENABLE}
75        - traefik.http.routers.grafana.rule=Host(`${
              GRAFANA_PUBLIC_HOST}`)
76        - traefik.http.routers.grafana.entryPoints=${
              TRAEFIK_ENTRYPOINTS}
77        - traefik.http.routers.grafana.tls=${TRAEFIK_TLS}
78        - traefik.http.services.grafana.loadbalancer.server.port=${
              GRAFANA_HTTP_PORT}
79        - traefik.http.routers.grafana.service=${
              GRAFANA_ROUTER_SERVICE}
80
81    opensearch-dashboards:
82      image: 'opensearchproject/opensearch-dashboards:2.19.1'
83      ports:
84        - '5601:5601'
85      environment:
86        - OPENSEARCH_HOSTS=https://${OPENSEARCH_PUBLIC_HOST}:${
              OPENSEARCH_HTTP_PORT}
87        - OPENSEARCH_USERNAME=${OPENSEARCH_USERNAME}
88        - OPENSEARCH_PASSWORD=${OPENSEARCH_PASSWORD}
89        - OPENSEARCH_TLS_REJECT_UNAUTHORIZED=${
              OPENSEARCH_TLS_REJECT_UNAUTHORIZED}
90        - DISABLE_SECURITY_DASHBOARDS_PLUGIN=${
              DISABLE_SECURITY_DASHBOARDS_PLUGIN}
91      labels:
92        - traefik.enable=${TRAEFIK_ENABLE}
93        - traefik.http.routers.osdash.rule=Host(`${
              OPENDASH_PUBLIC_HOST}`)
```

```
94        - traefik.http.routers.osdash.entryPoints=${
              TRAEFIK_ENTRYPOINTS}
95        - traefik.http.routers.osdash.tls=${TRAEFIK_TLS}
96        - traefik.http.services.osdash.loadbalancer.server.port=${
              OPENDASH_HTTP_PORT}
97
98  volumes:
99    grafana-storage: null
100   opensearch-data1: null
101   opensearch-data2: null
```

Listing A.1: Central Node Docker Compose)

Appendix B

Message Broker - Docker Compose

```
services:
  zookeeper:
    image: 'confluentinc/cp-zookeeper:7.5.1'
    container_name: zookeeper
    environment:
      ZOOKEEPER_CLIENT_PORT: ${ZOOKEEPER_CLIENT_PORT}
      ZOOKEEPER_TICK_TIME: ${ZOOKEEPER_TICK_TIME}
    networks:
      - kafka-net

  kafka:
    image: 'confluentinc/cp-kafka:7.5.1'
    container_name: kafka
    depends_on:
      - zookeeper
    environment:
      KAFKA_BROKER_ID: ${KAFKA_BROKER_ID}
      KAFKA_ZOOKEEPER_CONNECT: ${KAFKA_ZOOKEEPER_CONNECT}
      KAFKA_LISTENER_SECURITY_PROTOCOL_MAP: ${
         KAFKA_LISTENER_SECURITY_PROTOCOL_MAP}
```

```
20      KAFKA_LISTENERS: ${KAFKA_LISTENERS}
21      KAFKA_ADVERTISED_LISTENERS: ${KAFKA_ADVERTISED_LISTENERS}
22      KAFKA_INTER_BROKER_LISTENER_NAME: ${
           KAFKA_INTER_BROKER_LISTENER_NAME}
23      KAFKA_OFFSETS_TOPIC_REPLICATION_FACTOR: ${
           KAFKA_OFFSETS_TOPIC_REPLICATION_FACTOR}
24    networks:
25      - kafka-net
26    labels:
27      - traefik.enable=${TRAEFIK_ENABLE}
28      - traefik.tcp.routers.kafka.rule=HostSNI(`${
           TRAEFIK_KAFKA_HOSTSNI}`)
29      - traefik.tcp.routers.kafka.entrypoints=${
           TRAEFIK_KAFKA_ENTRYPOINTS}
30      - traefik.tcp.services.kafka.loadbalancer.server.port=${
           TRAEFIK_TCP_SERVICE_KAFKA_PORT}
31
32  logstash:
33    build:
34      context: .
35      dockerfile: Dockerfile
36    container_name: logstash
37    networks:
38      - kafka-net
39    volumes:
40      - './logstash/pipeline:/usr/share/logstash/pipeline'
41      - './logstash/config:/usr/share/logstash/config'
42    depends_on:
43      - kafka
44
```

```
45 networks:
46   kafka-net:
47     driver: bridge
```

Listing B.1: Message Broker Docker Compose

Appendix C

Probe Configuration Files

This appendix presents the exact Kubernetes manifests used to deploy the probe. Unless otherwise noted, Suricata uses the default configuration shipped in the container image; only the HOME_NET variable is adjusted to the chosen subnet of the department network.

```
apiVersion: v1
kind: Namespace
metadata:
  name: hr
```

Listing C.1: Probe Namespace

```
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: pod-hr
  namespace: hr
spec:
  serviceName: "pod-hr"
  replicas: 1
  selector:
    matchLabels:
```

```
11      app: pod-hr
12  template:
13    metadata:
14      labels:
15        app: pod-hr
16    spec:
17      hostNetwork: true
18      initContainers:
19        - name: suricata-rules-init
20          image: jasonish/suricata:latest
21          command: ["/bin/sh","-lc"]
22          args:
23            - |
24              set -e
25              suricata-update -D /var/lib/suricata
26          volumeMounts:
27            - name: suricata-lib
28              mountPath: /var/lib/suricata
29            - name: suricata-config
30              mountPath: /etc/suricata/suricata.yaml
31              subPath: suricata.yaml
32
33      containers:
34        - name: suricata
35          image: jasonish/suricata:latest
36          args: ["-i", "br-hr", "-S", "/var/lib/suricata/rules/
                suricata.rules"]
37          securityContext:
38            privileged: true
39          volumeMounts:
```

```
40            - name: suricata-logs
41              mountPath: /var/log/suricata
42            - name: suricata-config
43              mountPath: /etc/suricata/suricata.yaml
44              subPath: suricata.yaml
45            - name: suricata-lib
46              mountPath: /var/lib/suricata
47
48        - name: fluent-bit
49          image: cr.fluentbit.io/fluent/fluent-bit:2.1.9
50          env:
51            - name: SONDA_ID
52              valueFrom:
53                fieldRef:
54                  fieldPath: metadata.name
55          volumeMounts:
56            - name: fluent-bit-config
57              mountPath: /fluent-bit/etc/
58            - name: suricata-logs
59              mountPath: /var/log/suricata
60
61      volumes:
62        - name: suricata-logs
63          emptyDir: {}
64        - name: suricata-config
65          configMap:
66            name: suricata-config
67        - name: fluent-bit-config
68          configMap:
69            name: fluent-bit-config
```

```
70          - name: suricata-lib
71            emptyDir: {}
```

Listing C.2: Probe StatefulSet (Suricata + Fluent Bit)

```
1  apiVersion: v1
2  kind: ConfigMap
3  metadata:
4    name: fluent-bit-config
5    namespace: hr
6  data:
7    fluent-bit.conf: |
8      [SERVICE]
9          Flush          1
10         Log_Level      debug
11         Daemon         Off
12         Parsers_File   parsers.conf
13
14     [INPUT]
15         Name              tail
16         Path              /var/log/suricata/eve.json
17         Tag               suricata
18         Parser            json
19         Refresh_Interval  10
20         Read_From_Head    true
21         Skip_Long_Lines   On
22         Mem_Buf_Limit     10MB
23
24     [FILTER]
25         Name           modify
26         Match          *
27         Add            sonda_id ${SONDA_ID}
```

```
28
29      [OUTPUT]
30          Name              kafka
31          Match             *
32          Brokers           kafka.dev.amoz.space:9094
33          Topics            nids-logs
34          Format            json
35          Retry_Limit       False
36
37    parsers.conf: |
38      [PARSER]
39          Name            json
40          Format          json
41          Time_key        timestamp
42          Time_Format     %Y-%m-%dT%H:%M:%S.%f%z
43          Time_keep       On
```

Listing C.3: Fluent Bit ConfigMap for the Probe