UNIVERSITY OF TURKU

Monitoring Integration Traffic in Microservices Architecture

Computer Science
Faculty of Technology
Master's thesis

Author:
Elias Kinnunen

20.5.2025
Turku

Master's thesis

Subject: Computer Science
Author: Elias Kinnunen
Title: Monitoring Integration Traffic in Microservices Architecture
Supervisor: Dr. Jouni Smed
Number of pages: 56 pages
Date: 20.5.2025

Microservice architecture has become a widely adopted approach for building modern software systems, offering advantages such as improved scalability, maintainability, and flexibility over monolithic architectures. However, monitoring microservices remains a complex task due to their distributed nature and the intricate interactions between independent services.

This thesis investigates how integration traffic in Java-based microservices can be monitored more effectively. It identifies core principles for monitoring such traffic, examines the limitations of current solutions, and defines the key requirements for tools that support observability in microservice environments.

A mixed-methods approach was applied, combining literature review with semi-structured interviews of industry professionals. The study's main contribution is a set of design criteria for integration traffic monitoring tools that address both technical and organizational needs.

The results show that existing systems often lack the detail and context required to trace data across services. Effective monitoring requires human-readable logs, standardized formatting, alerting mechanisms, intuitive visualization, and message-level traceability.

Key words: monitoring, microservices, integration, integration traffic

Table of contents

1 Introduction

Over the past decade, the complexity of software systems has increased significantly due to rapid technological advancements and the growing need to integrate both legacy and modern systems. Architectural trends such as microservices have further amplified this complexity by promoting distributed, loosely coupled services that must still work together seamlessly to support complete business processes [1].

In this context, integration refers to the process of enabling different software components, applications, or systems to exchange data and functionality in a cohesive manner. Integrations can vary in scope, ranging from straightforward point-to-point connections to sophisticated orchestrations that involve multiple services and real-time data flows [2]. As the number and diversity of integrated services grow, the need for robust monitoring becomes increasingly important.

Monitoring, as defined by the IEEE, involves the supervision, recording, analysis, or verification of the operation of a system or its components [3]. In practice, monitoring provides visibility into how systems perform, whether they meet operational expectations, and when they require attention [1, 4]. It is a core mechanism for ensuring system reliability, maintaining performance, and supporting timely diagnostics.

In microservices architectures monitoring poses unique challenges. Services are decentralized, independently deployable, and subject to frequent change which make it difficult to gain a complete view of system behaviour [5, 6]. Monitoring individual services in isolation is often insufficient. Instead, there is a growing emphasis on achieving a context-aware view of system interactions, where the focus shifts to how integrated services collectively contribute to the functionality and experience of the end user or business application [7].

Adding to the complexity, distributed systems generate large volumes of constantly changing data, which can be hard to manage. Integration traffic refers to the flow of data and messages exchanged between microservices or systems as they coordinate to execute business processes.

This thesis specifically focuses on monitoring integration traffic within Java-based microservice architectures, utilizing the widely adopted Spring Boot and Apache Camel frameworks. While the empirical context is provided by Netum Oy, the challenges and solutions explored are relevant to a broader range of organizations employing similar

technology stacks. The goal is to enhance monitoring practices in a way that supports proactive system management, reduces manual effort, and improves the overall reliability of integrated services.

1.1 Research Context and Objectives

This thesis is conducted in collaboration with Netum Oy (hereafter referred to as either *Netum* or *Netum Oy*), an IT services company offering a wide range of solutions in areas such as cybersecurity, digital transformation, information systems, and system integration. The primary objective of this thesis is to examine the principles of integration traffic data monitoring and to utilize this information to identify the requirements for enhancing existing monitoring solutions.

The research aims to address existing limitations by clearly defining the needs of integration traffic monitoring. Although the research is grounded in microservices developed using technologies utilized at Netum's integration projects, the findings are designed to be applicable to a broader range of organizations facing similar challenges.

The thesis is guided by the following research questions:

RQ1: What are the key principles for designing effective monitoring solutions in microservices architectures?

RQ2: How can existing monitoring systems be improved to provide greater visibility into integration traffic flows?

RQ3: What are the key requirements for a monitoring tool to effectively support integration traffic monitoring in microservices architectures?

These questions define the scope and focus of the thesis, guiding both the theoretical exploration and practical analysis of integration traffic monitoring. They also help ensure the research remains aligned with real-world challenges faced in Netum's projects.

To answer these questions, a mixed-methods approach is employed. First, a literature review is conducted, covering academic publications, technical books, and industry reports related to monitoring in Java-based microservice systems. Based on this foundation, a set of design criteria for integration traffic monitoring is formulated. Second, semi-structured interviews are conducted with Netum professionals who have practical experience in monitoring and

managing integration traffic in microservice environments. These interviews are expected to provide a practical perspective on the real-world requirements and limitations of existing monitoring tools.

By combining theoretical research with hands-on expertise, this thesis aims to develop a comprehensive understanding of effective integration traffic monitoring. The findings are intended to support the development of more proactive, scalable, and business-aware monitoring practices, ultimately improving system reliability and maintainability in complex integration environments.

This research is carried out for the benefit of Netum and its clients, with the goal of enhancing service delivery through improved observability and monitoring capabilities. Netum Oy has had the opportunity to review the contents of this thesis prior to its publication to ensure that the outcomes are relevant, accurate, and aligned with its operational context.

1.2 Document Structure

This thesis is divided into six chapters, each building toward a comprehensive understanding of how integration traffic can be effectively monitored in Java-based microservice architectures.

Chapter 2 establishes the theoretical foundation by reviewing existing literature and concepts relevant to monitoring in microservices environments. It covers monitoring tools, practices, and perspectives alongside discussions on data collection, visualization, compliance, and the unique challenges posed by microservice architectures.

Chapter 3 presents the practical context by examining the integration and monitoring ecosystem at Netum Oy. It introduces the technologies currently in use, such as Apache Camel, Spring Boot, ActiveMQ, and various monitoring tools. The chapter also identifies key challenges in the current setup, which inform the design improvements proposed later in the thesis.

Chapter 4 focuses on identifying the key design principles of improved monitoring solutions. It applies decision guidance models to identify stakeholder needs, explores alternative design options, and defines a set of criteria for integration traffic monitoring tools. The chapter concludes with an overview of the existing monitoring tool landscape considering the defined criteria.

Chapter 5 presents the results of the empirical research. It outlines the research method, summarizes findings from interviews with Netum professionals, and synthesizes key themes related to design principles, monitoring challenges, and tool requirements in real-world scenarios. The discussion connects these findings to the research questions and broader implications.

Chapter 6 concludes the thesis by summarizing the main findings, discussing their contributions and limitations, and proposing directions for future work. It reflects on how the outcomes can support more effective, proactive, and business-aware monitoring practices in microservices-based integration environments.

2 Background

This chapter explores monitoring from a more abstract perspective, providing a broader understanding of its role. Monitoring encompasses a complex set of challenges that require careful consideration. Taking a higher-level view helps in designing more effective monitoring solutions by identifying key requirements, anticipating potential issues, and ensuring a scalable approach. Monitoring is inherently interconnected, with various approaches working together to provide a comprehensive view of system health. As a result, the perspectives in this chapter are closely related, with some topics appearing in different contexts.

A fundamental aspect of this discussion is understanding the primary purpose of monitoring. Monitoring serves to observe and report on the state of a system's components, enabling the detection of anomalies, collection of relevant data, and generation of useful information to support issue diagnosis and resolution. It is important to note, however, that monitoring systems do not inherently resolve problems; their value lies in supporting subsequent processes such as incident response and system recovery. [8, 9]

This chapter provides the theoretical foundation for the thesis by exploring the principles, tools, and challenges related to monitoring in microservice architectures. It begins by reviewing commonly used monitoring tools and emphasizing the importance of monitoring as a shared and continuous practice.

Different monitoring perspectives are discussed to highlight the multifaceted nature of observability. Attention is also given to compliance considerations, followed by a detailed look at data collection strategies, including metrics, logs, events, and alerting mechanisms.

Further sections examine data collection models (push, pull, hybrid), layered monitoring across system components, and the role of monitoring data visualization. The chapter concludes by focusing on the specific challenges and technologies associated with monitoring microservice-based systems, including distributed tracing, fault tolerance, and cloud-native environments.

2.1 Monitoring Tools

Monitoring tool can be defined as "a process or set of possibly distributed processes whose function is the dynamic gathering, interpreting, and acting on information concerning an

application as that application executes.” [1] Monitoring tools can be classified into two categories: libraries and platforms. Monitoring libraries are used during the development of microservices and permit collecting the application data. In contrast, monitoring platforms allow gathering and analysing data from different sources, such as the hosts, infrastructure, and microservices [5]. This means that the source code of the services being monitored must be extended or annotated with probes that suitably collect the metrics, logs, and traces of interest. Most available monitoring tools require instrumentation [1]. In contrast, the non-intrusive methods do not require instrumentation or monitoring agents for monitoring but are more uncommon [10].

There are tens of monitoring tools available for monitoring microservices [1]. However, the rapid increase in the number of tools has led to closer examination of how they are implemented. A common anti-pattern, known as "tool obsession," involves prioritizing the adoption of tools without a clear understanding of the problems they are intended to solve. While patterns offer proven solutions to recurring challenges by providing structured, effective approaches, anti-patterns reflect seemingly helpful practices that ultimately result in inefficiency or unintended consequences. In this context, tools should be viewed as enablers within a thoughtfully designed monitoring framework and not as standalone solutions. [9]

Legacy systems often impose additional constraints on tool selection, introducing a layer of complexity to the monitoring process. Knoche and Hasselbring surveyed the drivers, barriers, and goals of using microservices to modernize legacy systems. They observed that practitioner’s in Germany view monitoring distributed systems a significant barrier in modernizing legacy systems [11]. Decisions made in the past regarding infrastructure and software can create technical debt, limiting the organization’s ability to adopt modern, more versatile monitoring solutions. These constraints necessitate creative approaches to integrate new tools with existing systems, often requiring customized solutions or hybrid strategies that balance legacy dependencies with contemporary monitoring needs.

2.2 Monitoring as a Shared Practice

Effective monitoring is inherently a collaborative and continuous effort, demanding active participation from development, operations, security, and business teams. Integrator is the mediator position which emphasizes the need for effective communication between all parties involved.

When monitoring is treated as the sole responsibility of a single team, it easily creates knowledge gaps: developers who lack visibility into runtime behaviour may overlook essential instrumentation, while operators without architectural context are more likely to misinterpret operational signals. This is referred as “siloing” anti-pattern. [9]

Research on microservices monitoring observers that when knowledge is compartmentalized in this way, fault detection and diagnosis often depend heavily on the deep, specialized expertise of individual team members. This reliance can create bottlenecks in resolving incidents efficiently. Furthermore, the development and deployment approaches common to microservices, DevOps, and cloud environments promote a high degree of independence and specialization. As a result, monitoring and observability solutions often become fragmented, making it challenging to maintain a cohesive view of the service from a customer or business perspective. This highlights the importance of organizations actively addressing these issues to ensure effective system oversight. [7]

2.3 Monitoring perspectives

Monitoring can be categorized into three primary areas of focus:

1. Functional monitoring
2. Technical monitoring
3. Content monitoring

Each of these categories serves a distinct purpose, addressing specific aspects of system performance and reliability. Table 2.1 summarizes the three primary monitoring perspectives, highlighting their distinct focus areas and associated metrics.

Table 2.1: Monitoring perspectives and their focus areas.

Perspective	**Primary Focus**	**Key Metric examples**
Functional	Cross-service workflows	End-to-end latency
Technical	Infrastructure health	CPU usage, API error rates
Content	Data and contract integrity	Schema mismatches

2.3.1 Functional Monitoring

Functional monitoring prioritizes the end-user experience by evaluating workflows that span multiple services. In decentralized systems, user journeys often involve interactions with numerous independent components, such as payment gateways, inventory management, and recommendation engines. [12] Key metrics include transaction success rates across service boundaries, end-to-end latency for cross-service API calls, and error propagation paths that may cascade through interdependent services.

Research highlights the importance of simulating real-world failures to validate fault tolerance, a practice exemplified by chaos engineering methodologies that intentionally disrupt services to test recovery mechanisms [7]. For instance, frameworks like Google's Site Reliability Engineering (SRE) advocate for Service Level Objectives (SLOs) to quantify user experience, ensuring alignment between technical performance and business priorities. This approach prioritizes granular visibility into service dependencies, as incomplete or fragmented monitoring can obscure critical bottlenecks in user-facing workflows. [8]

2.3.2 Technical Monitoring

Technical monitoring focuses on the underlying infrastructure and components that sustain a system, such as server performance, network latency, database operations, and resource utilization. By tracking these metrics, organizations gain visibility into the health of their technical environment, enabling them to detect hardware or software issues that threaten system stability or efficiency [7, 8]. This proactive approach ensures potential bottlenecks or failures such as server overloads or database slowdowns are addressed before escalating into critical outages or performance degradation.

To align these infrastructure-centric metrics with user experience, frameworks like Google's Site Reliability Engineering (SRE) emphasize Service Level Indicators (SLIs), such as request latency and error rates. These SLIs connect technical metrics to user experience outcomes, like application responsiveness. By doing so, they help align operational priorities with real-world service quality [9].

Modern practices further enhance technical monitoring through methods like nonintrusive monitoring, which relies on passive network tracing or log analysis to infer system behaviour without disrupting workflows [10, 13]. However, infrastructure-level monitoring remains the

most widely adopted method due to its practicality: metrics such as CPU usage or disk I/O are straightforward to collect, requiring minimal development and operational effort compared to higher-level application monitoring [5].

2.3.3 Content Monitoring

Content monitoring safeguards data validity and structural compliance across interconnected systems, particularly in environments where data flows between independently managed components. A core challenge arises from the principle of decentralized data ownership, where subsystems maintain autonomous control over their data formats and storage mechanisms. This independence avoids tight coupling but introduces risks of inconsistency, such as mismatched schemas or unsynchronized updates across workflows. Data inconsistency is a common pain point especially in distributed services [6].

A key concern from the content perspective is schema drift, which refers to uncontrolled deviations from predefined data structures. In distributed architectures, this often occurs when independent teams modify data schemas without coordinated oversight. For instance, undocumented changes to timestamp formats, such as switching from UTC to localized time, in one subsystem can introduce errors into downstream analytics pipelines and ultimately lead to inaccurate insights. [13, 14]

Automated validation frameworks help mitigate schema drift by enforcing structural consistency. They compare incoming data against predefined schemas and flag deviations such as missing fields, type mismatches, or invalid enumerations [13]. These tools help to maintain contract integrity and ensure reliable data exchange.

2.4 Policy and Compliance Considerations

Service Level Agreements (SLAs) are formalized commitments between service providers and users, specifying expected levels of service performance and availability. These agreements establish a baseline for evaluating service quality and define key metrics that must be monitored and reported. The issue of compliance and regulations may be underrated in context of migrating from a monolithic architecture to microservices [11]. SLAs are often either undefined or inadequately enforced [7].

SLAs typically encompass a range of performance indicators, such as system availability, response latency, and incident resolution timeframes. Monitoring systems are required to

capture and analyse these metrics in real-time or near real-time to enable the timely detection and reporting of deviations. This clarity ensures both providers and users understand whether agreed standards are met. However, tracking SLAs can be challenging due to the interdependence of multiple services within a system. A failure or degradation in any component can impact the overall SLA compliance. When designing a monitoring system, SLAs should be carefully considered to ensure that all necessary metrics are tracked and reported effectively. [9]

2.5 Data collection

Data collection is a critical component of system monitoring, as it provides the foundational information required to analyse, diagnose, and optimize system performance. Observability refers to the ability to understand the internal state of a system based on the data it produces, such as metrics, logs, and traces. It is the basis of all data collection. Effective data collection strategies ensure the availability of accurate and timely data from various system components, enabling informed decision-making. The process involves capturing metrics, logs, events, and other telemetry data from monitored entities and transmitting it to a central location for storage, analysis, and visualization. [15]

2.5.1 Metrics, Logs, Events, and Alerts

To address the scope of data collection, it is necessary to define and analyse the various types of data being collected in detail. Metrics are numerical measurements that capture how a system performs and behaves across different time intervals. Various types of metrics exist, including gauges, counters, and timers. Gauges track properties that are expected to fluctuate often, like CPU utilization. Counters quantify the amount of a specific property, such as memory usage. Timers measure the duration of an action, for instance, the latency of requests. [4]

Metrics are typically collected at regular intervals to monitor trends, identify anomalies, and support performance analysis. The temporal granularity of collected data intervals is referred to as resolution (or time window). For instance, depending on requirements, metrics might be gathered at resolutions such as one-second, 30-second, 1-minute, or 5-minute intervals. [4]

Selecting an appropriate resolution granularity is critical: overly coarse resolutions risk omitting meaningful patterns, while overly fine resolutions impose increased storage demands

[7]. Figure 2.1 illustrates this trade-off: a coarser view (represented by the average line) can obscure critical spikes, while a finer view (seen in the higher percentiles) reveals them. For example, tracking CPU utilization with a 1-minute resolution could fail to capture sudden, critical spikes. Conversely, excessively detailed resolutions incur unnecessary costs -such as monitoring a highly available system's uptime at ultra-fine granularities, which offers negligible practical benefit. [9]

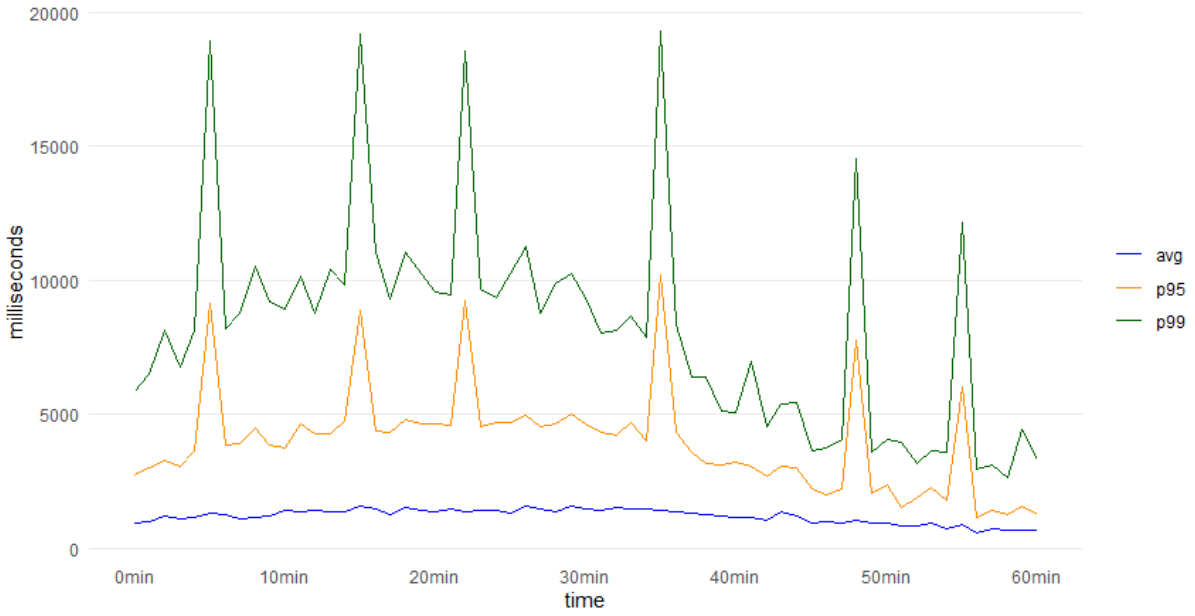

Figure 2.1: Impact of Monitoring Granularity on Capturing Performance Spikes.

These measurements are often aggregated and presented as time-series data, making them particularly useful for real-time monitoring and visualizing trends. Examples of metrics include CPU usage, memory consumption, request latency, transaction throughput, and error rates. By tracking these indicators, organizations can monitor system health, establish performance baselines, and detect deviations that may signal potential issues. Metrics are also critical for ensuring compliance with Service Level Agreements (SLAs) and triggering alerts when predefined thresholds are breached. [4]

Logs, on the other hand, are detailed, timestamped records of specific events or activities that occur within a system. They provide contextual information about system operations, errors, and user activities. Unlike metrics, which aggregate data, logs capture granular, often unstructured or semi-structured, textual information. Examples include application error messages, system startup or shutdown logs, database queries, and API request details. Logs are essential for debugging, root cause analysis, and auditing, as they allow engineers to

reconstruct the sequence of events leading to a failure or disruption. By examining logs, teams can troubleshoot issues effectively and gain a deeper understanding of system behaviour. [4]

Events represent significant occurrences or state changes within a system. While logs provide detailed records of activities, events focus on highlighting key milestones or activities that may require attention. Events are typically discrete notifications that indicate state changes such as service failures, successful user authentications, deployment completions, or resource scaling activities. They serve as a higher-level summary of system activities and are often used to trigger automated workflows or notifications. Events are particularly valuable in event-driven architectures, where specific actions, such as auto-scaling or alert generation, are executed in response to predefined triggers. [4]

Alerts serve as notifications generated when certain predefined conditions or thresholds are met. They are derived from metrics, logs, or events and are designed to inform system operators about issues that require immediate attention. Alerts are commonly configured with severity levels to denote the urgency and potential impact of specific issues. These notifications are triggered by predefined conditions, such as CPU utilization exceeding critical thresholds, unplanned application downtime, or sudden increases in error rates. Their primary function is to support incident management by providing timely notifications to engineers or designated stakeholders who can then investigate the issue. [4]

Table 2.2 Key Monitoring Concepts and Definitions

Concept	Description
Metric	A numerical measurement that represents the health or performance of a system, such as request latency or CPU usage, collected over time for analysis.
Log	A detailed, timestamped record of system activities or errors, providing context for debugging and tracing issues in microservices.
Alert	A notification triggered when a predefined metric threshold or error condition is met, signalling potential issues that require attention.
Event	A discrete occurrence within a system, such as a service start, user action, or failure, which may be logged or used to trigger alerts.

Table 2.2 summarizes the core concepts and their definitions. Metrics provide continuous performance data, which can trigger alerts when thresholds are breached. Logs complement metrics by offering detailed context for diagnosing issues that led to an alert. Events highlight significant changes or milestones, often prompting further analysis through logs or metrics. Alerts give stakeholders real-time updates on issues that need fixing.

2.5.2 Push, Pull, and Hybrid Models

Data collection in system monitoring can be divided into three different models: push (white-box), pull (black-box), and hybrid. Each model defines a specific mechanism for how data is transmitted from monitored components to the monitoring system.

1. *Push Model:* Relies on hosts, services, and applications actively transmitting data to a central collector, typically requiring instrumentation for data generation.
2. *Pull Model:* Observes system behaviour externally without accessing internal components, typically by querying the monitored components.
3. *Hybrid Model:* The hybrid model combines the push and pull methods to balance their respective strengths.

Monitoring strategies can also be categorized into reactive and proactive approaches. Reactive monitoring identifies issues as they occur, typically employing a black-box approach that observes system outputs without detailed insight into internal workings. For example, tracking response times or error rates allows teams to respond to visible performance degradations.

Proactive monitoring, often associated with a white-box approach, examines internal system metrics to anticipate potential issues. This might involve analysing resource utilization patterns or other indicators that suggest emerging risks. Most of the monitoring systems rely on reactive monitoring style while more mature systems seem to prefer more proactive approaches. [4, 8]

2.5.3 Levels of Monitoring

Effective monitoring requires a structured approach that obtains data from multiple levels of the system being monitored. Each level represents a layer of abstraction, from the foundational hardware to the high-level applications and services. The levels of monitoring are depicted in Figure 2.2.

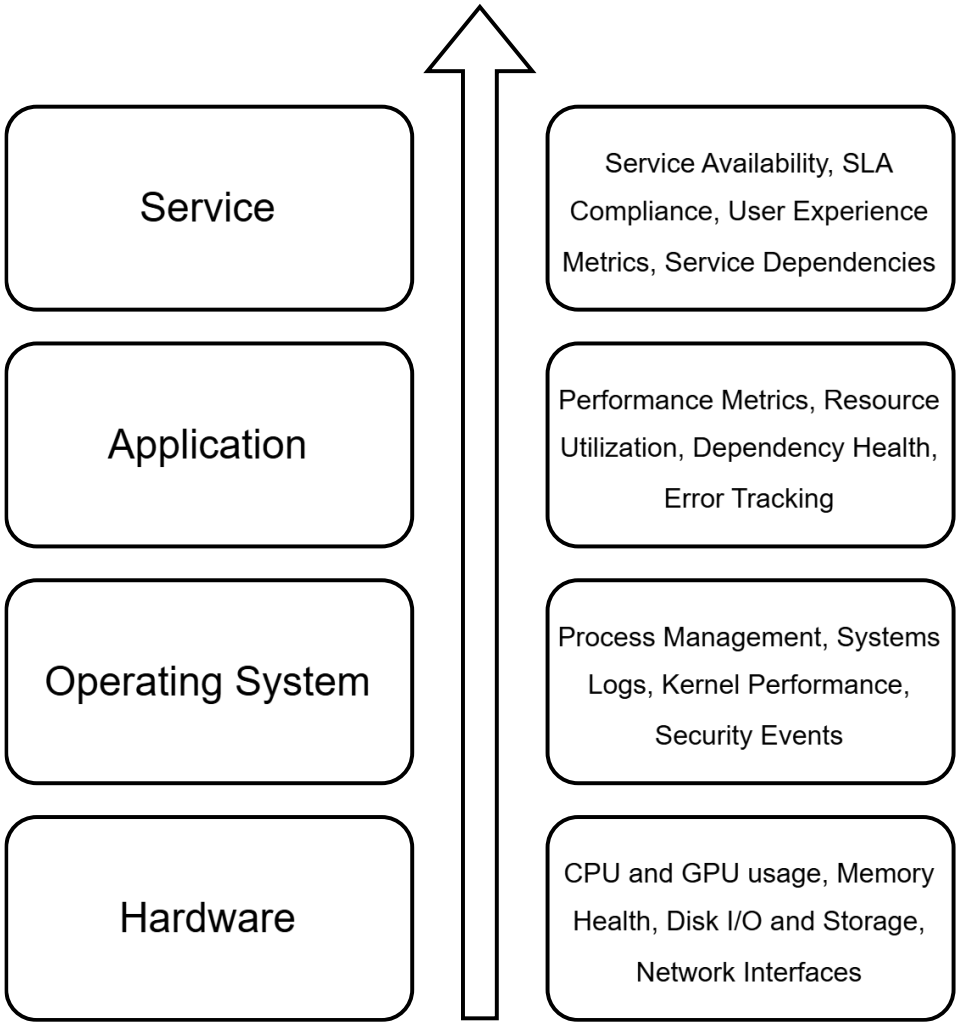

Figure 2.2. Monitoring Metrics Across System Abstraction Layers.

At the foundation of any system lies the hardware. Monitoring at this level focuses on the physical components that support the entire infrastructure. Key metrics include:

- *CPU and GPU Usage*: Monitoring the utilization of processing units ensures they are not overburdened, which could lead to system slowdowns or failures.
- *Memory Health:* Keeping track of RAM usage, errors, and swap activity helps identify potential bottlenecks or defective memory modules.
- *Disk I/O and Storage*: Monitoring read/write speeds, available storage space, and disk health is critical to prevent data loss and performance degradation.
- *Network Interfaces*: Observing bandwidth utilization, packet loss, and latency provides insights into network performance and potential connectivity issues.

Above the hardware layer lies the operating system (OS), which acts as the intermediary between hardware and software. Monitoring the OS involves:

- *Process Management*: Keeping track of active processes, their resource consumption, and potential deadlocks or hangs.
- *System Logs*: Parsing logs for warnings or errors helps identify patterns or issues that could escalate.
- *Kernel Performance*: Monitoring kernel activities, including interrupt handling and task scheduling, to ensure system stability.
- *Security Events*: Tracking unauthorized access attempts, malware activity, or misconfigurations that could compromise the OS.

Application monitoring is focused on the software components and processes that provide specific functionalities. Key considerations include:

- *Performance Metrics*: Monitoring request rates, response times, error rates, and throughput.
- *Resource Utilization*: Tracking memory usage, CPU cycles, and performance at the application level.
- *Dependency Health*: Ensuring that databases, APIs, and other external dependencies are responsive and meet performance expectations.
- *Error Tracking*: Capturing and analysing application logs for exceptions and stack traces.

At the highest level, monitoring focuses on the overall services provided by applications. This layer emphasizes:

- *Service Availability*: Ensuring uptime and accessibility of services to end-users.
- *SLA Compliance*: Verifying that services meet agreed-upon service-level agreements (SLAs) in terms of uptime and performance.
- *User Experience Metrics*: Monitoring latency, downtime, and transaction errors from the end-user perspective.

- *Service Dependencies*: Observing interactions and dependencies between services to detect cascading failures.

Resource usage, load balancing, availability, and database connections are among the most frequently monitored metrics [5], with over 70% of available microservice monitoring tools supporting their tracking [1]. In comparison, approximately 40% of microservice monitoring tools support user-oriented metrics, such as response times, latency, SLA violations, and network requests [1].

While each level has distinct monitoring requirements, integrating and correlating data across these layers provides a comprehensive view of the entire system. Monitoring all the different levels is necessary because it enables organizations to pinpoint issues accurately. For instance, if you only monitor the application layer and neglect the infrastructure layer, you might notice slow response times but miss the root cause, such as a failing server or network bottleneck. This lack of visibility can lead to prolonged troubleshooting and cascading issues, ultimately impacting system performance and user experience. [4, 9]

2.6 Data Visualization

Monitoring data visualization transforms raw metrics, logs, and traces into intuitive graphical representations. Visualization enables stakeholders to detect anomalies, diagnose faults, and make informed decisions by presenting data in human-readable formats [4]. For instance, proper tracing combined with sophisticated visualization analysis can help locate various kinds of faults, enabling quicker diagnosis and resolution of issues [16].

Tracing, which involves tracking the flow of requests and transactions through a system, is a foundational element in monitoring. When integrated with visualization tools, tracing data can be represented in formats such as flow diagrams or heat maps that clearly illustrate system behaviour over time. Such visual representations simplify the process of pinpointing performance bottlenecks or unusual patterns. In many monitoring frameworks various metrics are aggregated and displayed on dashboards that offer real-time insights into system health [6].

However, despite these benefits, dashboard visualization is less frequently used as a standalone method compared to other monitoring techniques. A key factor is likely the difficulty of setting up and managing these dashboards, along with the struggle to present complex data clearly so teams can act on it. [16]. An example of such a visualization is shown

in Figure 2.3, which illustrates how system metrics can be visually represented to reveal trends over time.

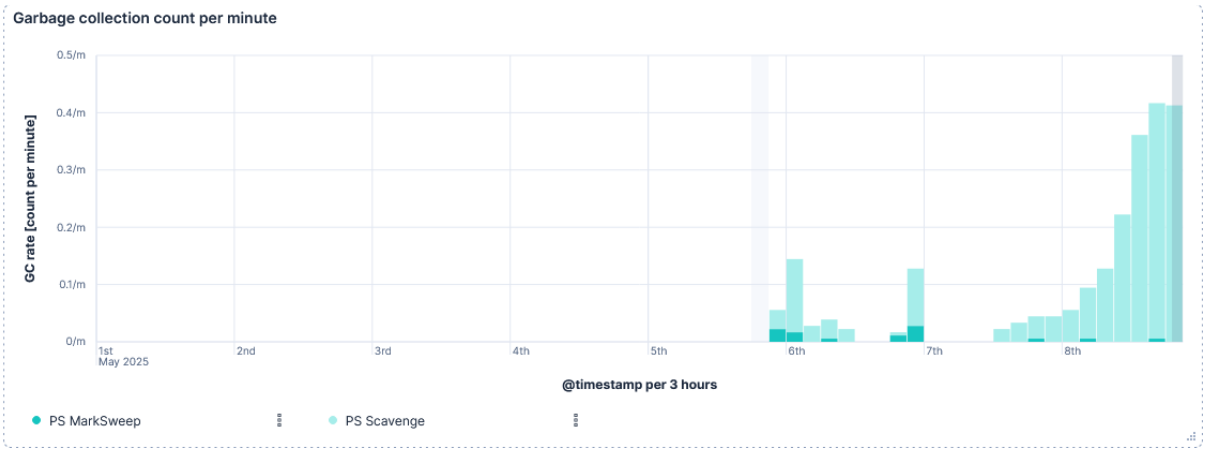

Figure 2.3: Screen capture from the visualization tool Kibana showing a monitoring dashboard of garbage collection count per minute, illustrating how system activity trends can be analysed and interpreted.

2.7 Monitoring Microservice Architectures

Modern applications increasingly adopt microservice architectures (MSA), which decompose software systems into distributed, independently deployable services. These services are often cloud-hosted and operate as technology-agnostic processes, each serving a specific function and communicating via lightweight protocols like HTTP. [16, 17]

This architectural style builds on service-oriented architecture (SOA), which structures systems as business-aligned services connected through a centralized middleware layer called an Enterprise Service Bus (ESB). The ESB acts as a software infrastructure that enables service integration, communication, and message routing between components. Microservices extend SOA principles by emphasizing finer granularity, decentralized communication, and independence in deployment and technology stacks. [17]

The transition from monolithic systems to SOA and microservices reshapes monitoring requirements. A study examining industry monitoring metrics, practices, tools, challenges, and solutions found insufficient evidence for a full comparison between microservice-based and non-microservice-based systems but identified log management as a common practice in both [5]. The table 2.3 summarizes key distinctions.

Table 2.3: Monitoring requirements across architectural styles.

Aspect	Monolithic	SOA	Microservices
Structure	Single process	Business-capability services	Decentralized, fine-grained services
Communication	Intra-process calls	Centralized service bus	Direct, protocol-based (e.g., HTTP)
Deployment	Unified deployment	Partially modularized	Independent, cloud-native units
Data Management	Centralized database	Shared databases	Service-specific storage
Monitoring Focus	Centralized metrics	Service bus performance	Distributed tracing, cloud metrics

2.7.1 Advantages and Challenges of MSA Monitoring

The microservices design pattern, which decomposes software applications into independent, self-contained services, underpins its key advantages. By distributing functionality across discrete components, this architecture enhances flexibility in design, development, and operations. Developers can work on individual microservices without deep knowledge of the entire system, accelerating development cycles. Each service operates in isolation, enabling independent scaling to meet demand and confining faults to specific components, thereby improving system resilience. [1]

However, the distributed nature of microservices also introduces inherent complexity, particularly in operational management. The most prominent challenges of monitoring microservices systems are collection of monitoring metrics data and logs from containers, and distributed tracing [16]. To help the localization of microservice faults four common monitoring strategies are implemented: log analysis, automatic alarming, call chain tracking, and dashboard [6]. Monitoring and troubleshooting become challenging due to the need to aggregate and analyse logs across multiple services [1]. For instance, tracing errors in interconnected microservices may require inspecting distributed invocation chains, complicating root-cause analysis [2]. Similarly, ensuring data consistency across independently managed databases adds overhead, while inter-service communication increases network resource costs [1]. Tools like distributed tracing, integrated into logging frameworks, mitigate these issues by mapping request flows across services [6, 15].

Aggregation is the process of collecting, consolidating, and analysing data from multiple sources to provide insights into the overall system. This approach is critical in microservice

architectures, where individual services generate isolated logs and metrics. By consolidating data, operators can identify patterns and correlations across services, detect anomalies and performance bottlenecks, and gain a high-level understanding of system behaviour. [12, 17]

In MSA-oriented systems, microservices collaborate with each other when handling requests. Monitoring these systems introduces challenges less prevalent in monolithic architectures, particularly in tracing user requests and mitigating cascading errors [15, 16]. Cascading errors arise when a failure in one service propagates to dependent services, often due to retries, timeouts, or resource exhaustion. For example, if Service A fails to handle errors from Service B, it may degrade Service A's performance, subsequently impacting Service C and creating a domino effect. These issues are exacerbated by service interdependencies and asynchronous communication in distributed systems [16]. Additionally, resource exhaustion can occur when a service consumer waits indefinitely for a provider's response, further destabilizing the system [3].

To address these challenges, three fault tolerance patterns are widely adopted in industrial practices: Circuit Breaker, Rate Limiting, and Degradation [3].

- *Circuit Breaker*: Used by 30% of interviewees, the Circuit Breaker pattern prevents cascading failures by adding an avoidance mechanism to the caller. When consecutive failures exceed a predefined threshold, the circuit breaker "trips", halting downstream requests during a timeout period. Afterward, it allows partial test calls and resumes normal operation once successful.

- *Rate Limiting*: Adopted by 35% of interviewees, this pattern sets a maximum QPS (Queries Per Second) threshold for each request type. Exceeding this threshold results in immediate request rejection, preventing resource exhaustion. Interviewees distinguish between service-level thresholds (e.g., API call limits) and infrastructure-level thresholds (e.g., CPU and memory constraints). Visualization tools like Kibana help monitor these metrics, with abrupt changes signalling potential faults. However, configuring thresholds remains challenging, as it requires scenario-specific analysis and often manual intervention [3].

- *Degradation*: Similar to the Circuit Breaker pattern and used by 30% of interviewees, Degradation prioritizes core services during high system load. If the overall load exceeds a preset threshold, non-core services are temporarily to preserve core

functionality. This ensures system availability under stress but requires careful balancing of service priorities [3].

In a distributed system, tracking the journey of a user request becomes complex because requests often traverse multiple services, each contributing to the overall response. Without proper monitoring, identifying delays or failures in the request path is difficult. Cascading errors further complicate this, as symptoms (e.g., latency spikes) may manifest in services far removed from the root cause [8].

To address this issue each incoming request is tagged with a unique identifier, or request ID. This identifier remains with the request throughout its lifecycle, including any subsequent requests it triggers. This process is called distributed tracing, and it can be utilized to obtain insights into the sequence of services a request interacts with, the time spent in each service, and any bottlenecks or failures in the request path. [15]

2.7.2 Key Technologies

Health endpoint pattern is commonly used in microservice monitoring [17]. Health endpoints are HTTP-based interfaces that return operational status metrics for a service, including availability, resource utilization, and dependencies (e.g., databases or external APIs). These endpoints typically respond with standardized HTTP status codes (e.g., 200 for "healthy", 503 for "unavailable") and may include detailed payloads describing subcomponent health. For instance, a health check might verify database connectivity, cache latency, or certificate validity, providing granular insights into system performance. [9, 12, 18]

To operationalize health data, monitoring tools aggregate and visualize endpoint responses. Frameworks like Spring Boot Actuator simplify implementation by offering preconfigured health checks and metrics endpoints [12].

In enterprise environments, Prometheus and Grafana are frequently paired with microservices to collect time-series data and generate dashboards. Research highlights additional tools widely adopted for monitoring, including the ELK stack (Elasticsearch, Logstash, Kibana), Jaeger, Dynatrace, Amazon CloudWatch, and Zipkin [1, 2, 5, 19]. These tools collectively support critical monitoring patterns such as application metrics, health check APIs, log aggregation, and distributed tracing [1].

Practitioners emphasize the importance of combining log management, exception tracking, and health check APIs to trace issues and identify root causes in microservice systems. For example, Spring Boot Actuator is used by 22.6% of practitioners (24 out of 106) for health monitoring, while Grafana (17.9%) and Zipkin (16.9%) are leveraged for visualization and distributed tracing, respectively. Notably, Jira (39.6%) and Datadog Kafka Dashboard (38.6%) also rank highly as monitoring aids, reflecting the integration of issue-tracking systems with observability workflows. [16] Figure 2.4 illustrates these adoption trends, based on data from *Design, Monitoring, and Testing of Microservices Systems: The Practitioners' Perspective* [16].

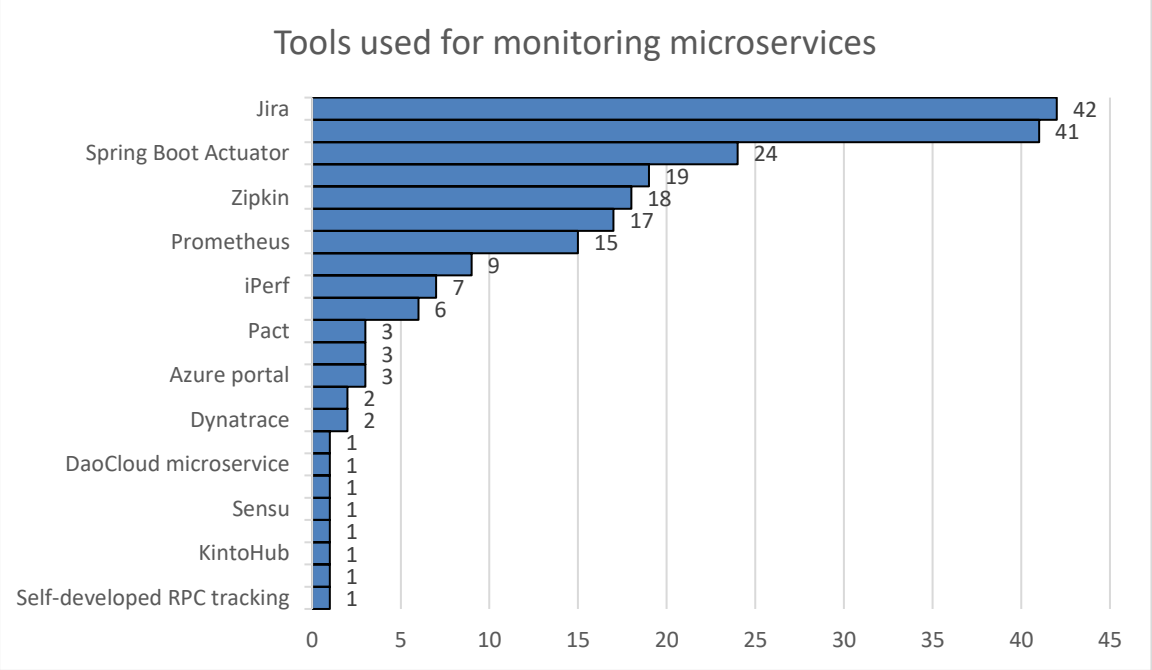

Figure 2.4 Adoption of monitoring tools in microservice systems (Recreated from *Design, Monitoring, and Testing of Microservices Systems: The Practitioners' Perspective* [16]).

2.7.3 Cloud-Native Considerations

The shift towards microservices architecture (MSA) in cloud environments has been significant. International Data Corporation (IDC) predicted that by the end of 2021, 80% of cloud software would be developed using the MSA style [5]. Cloud environments inherently support the deployment and scaling of microservices by providing elastic resources, orchestration platforms, and managed services [8]. However, this elasticity introduces monitoring challenges distinct from static architectures. Unlike monolithic systems, cloud-based microservices require monitoring classes of resources, such as containers, pods, or serverless functions, rather than individual static components [8]. While research continues to

bridge the complexity gap between cloud environments and their monitoring, practical adoption of these solutions remains limited [7].

Leading cloud platforms such as AWS, Google Cloud, and Microsoft Azure provide built-in monitoring solutions (e.g., Amazon CloudWatch, Google Cloud Operations Suite, Azure Monitor) optimized for their environments. These tools support microservices architectures by enabling real-time metrics gathering, log analysis, and distributed tracing. While effective within their ecosystems, they pose vendor lock-in risks due to limited interoperability with other platforms. Research highlights this challenge, noting that vendor lock-in occurs when organizations become dependent on a single provider's infrastructure. This dependency often results in high migration costs or technical obstacles and is largely caused by the lack of standardization in cloud technologies. [20] Survey reveals that half of the companies rely solely on their cloud providers' monitoring dashboard. Participants expressed a need for quality of service monitoring integrated with their monitoring tool. [7]

Platform-specific monitoring tools reinforce this dependency. For example, AWS Lambda's integration with CloudWatch or Azure's Application Insights ties observability workflows to proprietary systems, complicating cross-cloud portability. Such lock-in restricts flexibility, as shifting providers may require redeveloping monitoring strategies or adopting new tools entirely. Cross-cloud incompatibilities in APIs, data formats, and service integrations further amplify operational and financial risks. [20]

Dynamically assigned network locations, automatic upgrades, and scalable resource provisioning in cloud-based microservices architectures exemplify the elasticity that modern cloud platforms offer. Automated scaling mechanisms continuously adjust computational resources in response to fluctuating workloads, thereby ensuring both optimal performance and cost efficiency. However, this inherent dynamism introduces challenges to conventional monitoring practices. Since resource instances like containers or serverless functions are temporary and distributed across multiple nodes or zones, traditional monitoring methods often fall short. These conventional approaches are designed for static components and struggle to track dynamic resources that may appear or disappear within seconds. [18] This necessitates adaptive monitoring strategies capable of dynamically discovering and aggregating metrics from short-lived instances without disrupting performance. [21]

3 Integration and Monitoring Ecosystem at Netum Oy

Building on the general monitoring principles introduced in the background chapter, this chapter focuses on Netum Oy's architectural framework for microservice-based integrations, translating broad observability concepts into a specialized ecosystem. The integration landscape includes roughly 50 individual microservices, connecting multiple data sources and outputs across various systems. Due to the nature of Netum's clients' business, these services are deployed on-premises rather than in the cloud. However, the underlying architectural principles remain equally applicable to cloud-based deployments.

A combination of open-source solutions is presented, including Apache Camel for message routing, Spring Boot for streamlined microservice development, and ActiveMQ for asynchronous communication, along with monitoring platforms such as Zabbix and Nagios. Additionally, lightweight utilities such as Hawtio, Jolokia, and essential Linux/GNU tools like grep, awk, and bash are integrated to address the challenges of distributed systems.

A detailed examination of each tool's role, capabilities, and limitations within the microservices architecture is provided. The analysis reveals gaps in the current monitoring setup, such as an over-reliance on reactive log analysis and the absence of business-centric metrics—a challenge addressed in subsequent chapters.

3.1 Integration Tools at Netum Oy

Netum Oy's microservices architecture is built upon a suite of integration tools. This chapter introduces five key technologies at the core of this architecture: Apache Camel, which handles message routing and transformation; Spring Boot, the foundational framework for microservice development; ActiveMQ, enabling asynchronous messaging; and Hawtio and Jolokia, which provide tools for management and diagnostics. Each tool is examined in terms of its core functionality and its role within the broader integration landscape. Together, these technologies form a backbone for supporting distributed system operations and establish the foundation for the monitoring challenges and enhancements explored in later sections.

3.1.1 Apache Camel: Routing and Mediation Framework

Apache Camel is a flexible open-source integration framework designed to connect heterogeneous systems through declarative routing workflows. Camel implements Enterprise

Integration Patterns (EIPs), established design patterns which abstract complex communication logic into reusable components [2]. Camel's key features include:

- *Domain-Specific Language (DSL):* Camel provides Java and XML-based DSLs to define routes that specify message sources (e.g., HTTP endpoints, message queues), processing steps (e.g., transformation, filtering), and destinations.
- *Protocol Agnosticism*: It supports over 300 connectors (e.g., HTTP, JMS, FTP) to interface with databases, APIs, and legacy systems, enabling data exchange across various formats such as JSON, XML and CSV.
- *Spring Boot Integration*: Camel's first-class support for Spring Boot simplifies embedding routing logic into microservices. For example, a Camel route in a Spring Boot service might ingest data from an ActiveMQ queue, transform it using XSLT, and forward it to a REST API.

Camel accelerates the development of scalable integration pipelines by abstracting low-level communication details. [12]

3.1.2 Spring Boot: Microservices Development Framework

Spring Boot is a Java-based framework optimized for building production-ready microservices. Its "convention over configuration" philosophy reduces boilerplate code and accelerates deployment:

- *Autoconfiguration*: Automatically configures Spring applications based on the added jar dependencies, eliminating manual setup.
- *Standalone Packaging*: Applications are compiled into self-contained "fat JARs" with embedded servers which enables deployment without external application servers.
- *Camel Integration*: Spring Boot's dependency management simplifies embedding Apache Camel routes into microservices. For instance, a *spring-boot-starter-camel* dependency automatically configures Camel contexts and route scanners.
- *Actuator*: Spring Boot Actuator provides production-ready endpoints (e.g., /*health*, */metrics*, */info*, /*env*) that expose application health, metrics, thread dumps, and environment info (discussed in chapter 2.7.2). These endpoints can be secured,

extended with custom probes, and easily hooked into external monitoring systems to provide deep visibility into microservice performance and behaviour at runtime.

The lightweight runtime and modular architecture of Spring Boot make it well-suited for developing integration-oriented microservices that utilize Apache Camel's advanced routing capabilities. [12, 17, 18]

3.1.3 Hawtio and Jolokia: Management and JMX Tools

Hawtio and Jolokia are tools used to manage Java-based integration components. Jolokia acts as a bridge between Java Management Extensions (JMX) and HTTP, making it possible to access JMX metrics and operations through RESTful APIs. It translates JMX operations into standard HTTP and JSON requests, which simplifies the remote management of Java applications and avoids the complexity associated with using Java's Remote Method Invocation (RMI) protocol. [22]

Hawtio is a web-based console that leverages Jolokia to interact with JMX-managed resources. Key functionalities include:

- *Camel Route Management*: Viewing active routes, starting and stopping routes, and inspecting message processing steps.
- *ActiveMQ Administration*: Creating queues and topics, browsing messages, and managing brokers.
- *System Diagnostics*: Monitoring JVM metrics (such as memory and thread count) and Spring Bean configurations.

While not strictly monitoring tools, Hawtio and Jolokia aid in managing and observing the runtime behaviour of integration components by exposing real-time metrics, operational controls, and configuration data. This enables developers and operators to monitor system performance, troubleshoot issues, and manage message flows through both programmatic access and a graphical user interface. [12, 23]

3.1.4 ActiveMQ: Messaging Backbone

ActiveMQ is an open-source message broker that facilitates the exchange of messages between systems. It supports protocols such as the Java Message Service (JMS), which is a

Java API for sending messages between two or more clients, and the Advanced Message Queuing Protocol (AMQP), which is a platform-neutral protocol designed for message-oriented middleware. By using these protocols, ActiveMQ enables asynchronous communication between microservices, allowing them to interact without waiting for immediate responses. Its role in Netum's architecture includes:

- *Decoupled Communication*: Producers and consumers exchange messages via queues and topics without direct dependencies, improving fault tolerance.
- *Camel Integration*: Apache Camel provides an *activemq* component to seamlessly send and receive messages. For example, a Camel route can consume messages from an *orders.queue*, process them, and forward results to a *fulfillment.topic*.
- *Persistence and Reliability*: Messages are persisted to disk to prevent loss during outages, with configurable redelivery policies for failed transactions.

ActiveMQ's integration with Spring Boot enables streamlined setup and configuration, while its scalability makes it well-suited for high-throughput scenarios such as event-driven order processing. [12, 24]

3.2 Monitoring Tools at Netum Oy

This chapter presents the key monitoring tools used at Netum Oy to oversee its on-premises microservice-based integration systems. The monitoring strategy combines infrastructure-level platforms, service-specific diagnostics, and custom log analysis scripts to provide multi-layered visibility into system operations.

The first section introduces Zabbix and Nagios, which are used for infrastructure and service availability monitoring. Zabbix serves as the primary tool, offering centralized metric collection and alerting, while Nagios remains in use for legacy system checks.

Next, the chapter describes self-monitoring mechanisms built into selected microservices. These services can detect failures in critical operations and send alerts directly, complementing infrastructure-level monitoring with real-time feedback from within the application layer.

Finally, the chapter details Netum's custom log monitoring scripts, which collect application logs, scan for known errors using pattern matching, and store results in a MongoDB-based

knowledge base. Alerts are sent when issues are detected, supporting incident response and tracking.

Together, these tools form a functional but primarily reactive monitoring setup. The lack of centralized insight into integration traffic or business-level metrics highlights areas for improvement, addressed in the following chapters.

3.2.1 Zabbix and Nagios: Infrastructure Monitoring and Alerting

Zabbix serves as Netum's primary monitoring platform for tracking infrastructure health, resource utilization, and service availability. It operates on a pull-based model, where the server actively retrieves data from monitored devices (see Section 2.5.2). It was originally introduced to replace Nagios as a primary monitoring tool at Netum. Core components include:

- *Zabbix Server*: Centralizes data collection, processes triggers (e.g., CPU > 90%), and manages alerts.
- *Zabbix Proxy*: Deployed to reduce server load by preprocessing data from monitored devices.
- *Zabbix Agent*: Installed on monitored systems to collect metrics like disk usage, memory consumption, and application statuses.

Current monitoring setup monitors services and infrastructure on many different layers (discussed in Section 2.5.3). Zabbix's implementation at Netum leverages template-driven configurations, managed through the Zabbix server UI, to monitor standardized infrastructure metrics like disk space, RAM usage, and network traffic across servers. It also monitors specific files and directories, including password changes and processing directories, to detect potential issues. Predefined thresholds trigger email notifications for critical events such as host downtime or resource exhaustion.

Though superseded by Zabbix as the primary tool, Nagios retains a supplementary role at Netum. It performs basic host checks using protocols like ICMP (ping), TCP port verification, and SSH-based service monitoring. It also monitors older Tomcat-based ActiveMQ routes, ensuring continuity for legacy systems while allowing gradual migration to Zabbix's more advanced capabilities.

3.2.2 Microservices Self-Monitoring

In addition to infrastructure-level tools, some of Netum's microservices applications incorporate built-in self-monitoring capabilities. These services autonomously verify the success of critical operations, such as data transmissions. If a failure is detected, the microservice triggers automated email alerts to notify appropriate personnel. This push-based approach complements the broader infrastructure monitoring provided by Zabbix and Nagios.

Relying exclusively on self-monitoring introduces limitations. For instance, the decentralized nature of self-monitoring can result in a lack of a centralized view of system-wide health, obscuring interdependencies between services. Additionally, potential inconsistencies may arise if individual microservices implement varying thresholds or criteria for failures, leading to fragmented visibility or redundant alerts. This is why Netum utilizes self-monitoring as a complementary method rather than the sole monitoring strategy.

3.2.3 Custom Scripts: Log Monitoring System

Netum Oy employs custom scripts to perform application-specific log monitoring. The system retrieves recent log files from applications running on monitored machines using application-specific parameters stored in a JSON-based knowledge base. Identified issues trigger email notifications, with one email generated per analysed log file.

The monitoring process runs on a dedicated server where log files are copied from remote machines to local storage for analysis, ensuring that the original files remain unmodified. Predefined search criteria are applied to extract relevant log entries, with all processing handled locally on the monitoring server. The script accepts variables as parameters to both limit the scope of the search and define specific search terms drawn from the knowledge base.

Error information is tracked in MongoDB, a widely used document-oriented NoSQL database that stores data in flexible, JSON-like documents for efficient management, as shown in Table 3.1.

Table 3.1: MongoDB record format.

Key	Type	Explanation
_id	string	Unique MongoDB identifier
posttype	string	Error severity: • *FATAL*: critical error requiring immediate action • *ERROR*: error that requires customer notification or corrective action • *UNKNOWN ERROR*: error not yet defined (requires review and potential addition to known errors) • *WARN*: error to be reported, usually non-critical unless repeated • *INFO*: informational message or significant log entry
searchtype	string	Specifies the search type: • *Exact*: includes all errors defined in the database • *Generic*: includes general error terms (e.g., "ERROR") to capture new error types
searchword	string	The specific search term used to identify errors in the log (e.g., "OutOfMemoryException")
postinfo	string	A description of the error, its nature, and possible actions to be sent to the control mailbox
helpurl	string	URL directing to Netum Oy's Confluence page with detailed information
alternativemail	array	List of alternative email addresses to use instead of the default mailing list
application	array	List of integration applications affected by the error. If empty, the error is searched across all application logs

The *searchword* variable defines the terms scanned in logs, while *searchtype* determines whether the search is precise or broad. Generic search terms such as *ERROR* are intentionally limited to avoid excessive noise. While MongoDB can store nearly unlimited records, maintainability is deemed optimal with fewer than 100 entries.

The original integration application logs are copied to the monitoring server using the *scp* command. These logs are obtained by running a script that compiles information about currently running integration applications and their configuration files. Consequently, any changes to the running applications directly affect the listing results. Once copied as temporary files, the logs are processed by an awk script, which produces a new file containing the extracted error information. The log monitoring process is depicted in Figure 3.1.

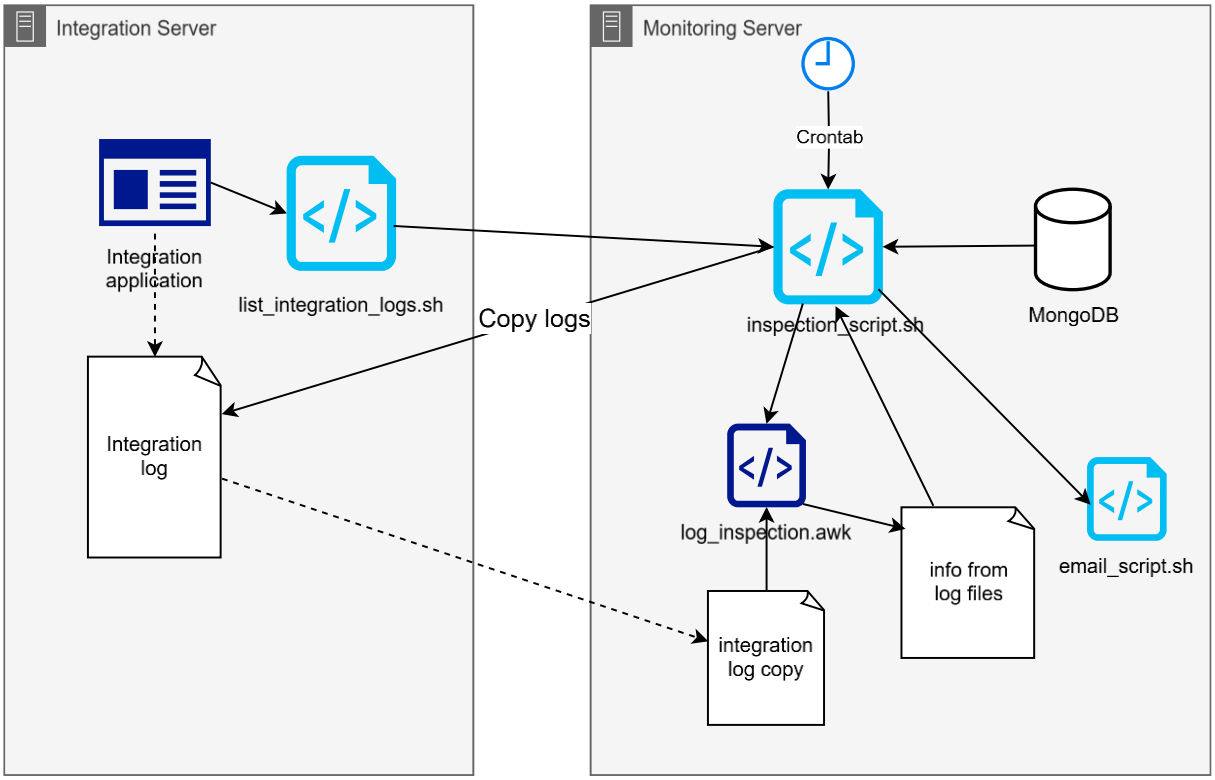

Figure 3.1: Netum's log-monitoring system. Monitoring consists of three scripts and a more general script that sends mail.

3.3 Apparent Challenges in the Current Monitoring Setup

Netum Oy's integration and monitoring ecosystem summarized in Table 3.2 combines modern microservices tools, legacy systems, and custom scripts.

Table 3.2: Tools and frameworks for integration, monitoring, and management.

Tool or framework	Description
ActiveMQ	Facilitates asynchronous messaging between services.
Apache Camel	Orchestrates data flows via EIP-driven routes.
Hawtio	Manages Camel routes, ActiveMQ brokers, and Spring Beans via GUI.
Jolokia	Exposes JMX metrics for programmatic management.
Monitoring Scripts	Custom scripts that inspect logs (e.g., grep, awk) for error detection and alerts.
Nagios	Legacy system that monitors infrastructure outages via ping, TCP, and SSH checks.
Spring Boot	Simplifies microservice development and deployment.
Zabbix	Newer monitoring system that tracks infrastructure performance, network metrics, and service availability.

The integration environment spans multiple architectural eras, resulting in inconsistent technologies and workflows. As observed in the research and previous chapters, systems often contain legacy and modern service technologies in parallel, requiring additional tooling to integrate legacy components. This technological heterogeneity complicates the consistent application of monitoring tools across services. [7]

A key limitation is the emphasis on infrastructure health rather than in the integration traffic itself. Zabbix and Nagios monitor system metrics such as CPU usage, disk space, and service uptime but do not provide visibility into the transmitted data. The system lacks a central view of the situation, creating blind spots (“siloing” anti-pattern described in Section 2.2).

Custom scripts parse logs for predefined error patterns and trigger alerts only after issues occur. This approach makes it difficult to identify emerging trends because message counts are not tracked or visualized. Additionally, logs must be checked manually by developers whenever managers, or other non-technical personnel have questions, as they lack a clear view of the messages being transmitted—apart from occasional alerts triggered by custom scripts. This primarily represents a reactive monitoring model, as discussed in Section 2.5.2.

This manual, reactive approach delays root-cause identification, forcing developers to cross-reference Zabbix alerts, Nagios downtime reports, and log snippets to diagnose failures. Furthermore, collected metrics lack business context. For instance, Zabbix and Nagios focus on technical monitoring but cannot answer critical questions like “How many messages in total were delivered last week?” or “What data was forwarded to a target system?” As noted in the research and previous chapters, teams often operate without awareness of the broader business context, resulting in siloed perspectives and weak collaboration across services. [7]

The practical solutions to these problems are explored in the next chapter. Information derived from integration traffic data enable proactive decision-making, reduce reliance on manual efforts, and foster shared situational awareness across teams by aligning infrastructure health with customer outcomes.

In summary, Netum Oy’s current integration and monitoring ecosystem, while functional, faces several challenges. The reliance on a mix of modern and legacy systems creates inconsistencies and the emphasis on infrastructure health overlooks the critical need for visibility into integration traffic itself. The lack of a centralized view and the reactive nature of log-based monitoring impede proactive issue identification and timely root cause analysis.

To address these limitations and enhance the monitoring of integration traffic within microservices architectures, Chapter 4 will explore the key requirements for the design of enhanced monitoring system.

4 Monitoring Design and Criteria

This chapter transitions from the foundational concepts and existing ecosystem analysis towards the practical design and specification of an improved monitoring solution for Netum Oy's integration environment. Leveraging decision guidance models, it first identifies key stakeholder requirements, focusing particularly on the need for business-contextual insights currently lacking.

The chapter then systematically explores various design options for generating, managing, and processing monitoring data, evaluating them against the identified needs and Netum's constraints. Based on this analysis, specific selection criteria for a monitoring toolset are established.

To further inform the solution design, the chapter presents a comprehensive overview of the current monitoring tool landscape, assessing how well existing solutions align with the defined selection criteria. The analysis highlights significant challenges in achieving effective integration traffic monitoring, particularly due to gaps in support for message-level visibility and data integrity validation. These observations suggest a lack of research and targeted tools to address the demands of integration traffic monitoring.

4.1 Decision Guidance Models

The apparent challenges identified in Section 3.3 are based on initial observations drawn from the monitoring foundations discussed in the background chapter. To support and validate these observations, this chapter also incorporates decision guidance models for microservices monitoring as introduced by Haselböck et al. [27], offering a structured lens through which the current setup can be evaluated and improved. These models provide a foundational framework that is revisited throughout the chapter.

Decision guidance models are a well-known approach for design space exploration and decision-making in software architecture. They serve as a means for both design space exploration and documentation.

The introduced decision guidance models are intended for several purposes, including:

- Identifying key requirements for a microservice infrastructure.
- Selecting the best design options to address specific requirements.

- Comparing design options based on their consequences (implications).
- Identifying concerns such as requirements, goals, and stakeholder needs addressed by a particular design option.
- Determining essential system components (e.g., an aggregation service) required to implement a specific design option.
- Identifying the design options implemented by a given monitoring technology.
- Comparing existing monitoring technologies in terms of their support for specific design options and/or requirements.

The presented decision guidance models were developed based on a literature review, previous work on monitoring, and a comparison of existing monitoring technologies. [25]

It is noted that the models focus primarily on qualitative design decisions and lack empirical data or performance benchmarks for comparing tools. Cross-impact analysis between model elements is also limited. Changes in one area (e.g., instrumentation strategy) may not dynamically reflect consequences in others (e.g., data processing or storage). Furthermore, security and compliance considerations are only lightly treated.

The introduced decision guidance models serve as inspiration for improving the monitoring system and are used alongside insights from the background chapter and other research sources. They function as a structured checklist applied in three key areas:

1. Identifying stakeholder needs
2. Exploring design options to address those needs
3. Establishing criteria for evaluating potential monitoring tools

These steps help ensure that the presented data is relevant to stakeholders, that suitable design solutions are thoroughly explored, and that choices of monitoring tools are guided by well-defined criteria. Together, they also help identify any challenges that may otherwise have been overlooked, including challenges discussed in Section 3.3.

4.2 Stakeholders and Their Requirements

The model identifies developers, operators, quality assurance personnel, and managers as key stakeholders in a microservices monitoring ecosystem. Based on the analysis in Section 3.3, the current system relies heavily on manual, reactive monitoring practices that hinder timely root-cause identification.

These limitations disproportionately impact managers and operators. Managers require business-contextual data to assess the health of integrations in terms of their impact on operations and decision-making. Operators are tasked with interpreting system-level data and conveying information to both technical teams and business stakeholders. Without appropriate tools and visibility, both groups struggle to perform their roles effectively, resulting in siloed perspectives and reduced cross-team collaboration as highlighted in previous chapters.

Therefore, the primary stakeholders targeted by the proposed monitoring improvements are managers and operators. They require intuitive dashboards and reports that not only display real-time runtime data but also contextualize this information in terms of business impact. This aligns with the need, outlined in Section 2.3, for monitoring tools that bridge the gap between technical metrics and business insights.

Haselböck et al. [25] categorize monitored data into static and runtime information to better understand the types of data that are relevant to different stakeholders:

- *Static information:* Refers to relatively unchanging details such as service types, interfaces, associated developers, and general metadata. However, this type of information is not relevant to the current scope.
- *Runtime information:* More directly relevant to the current monitoring challenges and divided into three key subcategories:
    - *Service runtime metrics*: Includes indicators such as CPU load, throughput, failure rates, and response times. While the existing setup captures host-level metrics, improving visibility into service-specific metrics remains a primary focus.
    - *Service interactions:* Encompasses communication paths, microservice dependencies, and root cause analysis.

- *Service runtime infrastructure*: Covers aspects like running service instances, host availability, and data center distribution. This area is acknowledged but not targeted for improvement within the current scope.

Integration traffic monitoring operates at the level of service interactions and service runtime metrics. Service interactions include communication paths, microservice design, and root cause analysis—with communication paths and microservice design being especially relevant for understanding integration flows.

Service runtime metrics, such as failure rate, throughput, and latency, provide insight into traffic behaviour and system performance. To effectively track integration traffic, microservices should be designed with observability in mind (see Section 2.5).

4.3 Exploring Design Options

With stakeholders' needs and monitoring focus areas established, various design options for microservice monitoring can be explored. According to Haselböck et al. [25], microservice monitoring can be divided into four major areas:

1. *Generation of Monitoring Data:* Instrumentation of hosts, runtime environments, and services to produce monitoring data.
2. *Data Management:* Storage, hosting, and distribution of collected monitoring data.
3. *Processing of Monitoring Data:* Analysis of data to derive service interactions, service runtime metrics, infrastructure metrics, and perform data aggregation (see Section 2.7.1).
4. *Dissemination and Presentation:* Visualization and presentation of monitoring data through user interfaces for stakeholders. (see Section 2.6)

Each of these areas is supported by one or more decision guidance models. In the following sections, each category is examined in detail to evaluate its relevance and implementation options within Netum Oy's integration monitoring framework.

4.3.1 Generation of Monitoring Data

Focusing on the first area, generation of monitoring data, Haselböck et al. [25] define three different layers of the system:

- *Host level*: An agent (such as Zabbix or Nagios agent) is installed on each host in the infrastructure. These agents collect core system metrics like CPU load, memory usage, and disk I/O. This provides broad visibility across the infrastructure but lacks context for specific applications or runtime environments.

- *Platform level*: Each runtime environment (e.g., JVM or database server) is instrumented using specialized tools or native plugins in systems like Zabbix or Nagios. This enables more granular, technology-specific monitoring, capturing metrics such as garbage collection times, memory usage, or thread activity regardless of the underlying host operating system.

- *Service level*: Instrumentation is embedded directly within each microservice using self-monitoring services, custom service logging scripts, or metrics frameworks. This allows for detailed, service-specific observability, including business-level metrics and internal interactions such as API call timings and error rates. Developers integrate instrumentation code or agents during development and maintain them through deployment pipelines, making monitoring an integral part of the service lifecycle.

Netum's monitoring setup employs a combination of instrumentation techniques applied across various architectural layers. The different levels of monitoring are discussed in detail in Section 2.5.3, while their implementation within Netum's monitoring architecture is outlined in Section 3.2.

To summarize, the current setup includes tools and methods that cover all three instrumentation levels: host, platform, and service. However, the most relevant information for managers and operators is typically generated at the service level. To better support their needs, the next step is to enhance the existing foundation in a way that provides clearer visibility into integration traffic and service interactions that impact business operations.

4.3.2 Storing of Monitoring Data

Generated monitoring data must also be efficiently stored to support analysis and decision-making. Haselböck et al. [25] present a decision guidance model for storing monitoring data, which distinguishes between two mutually exclusive architectural design options:

- *Centralized storage*: Monitoring data is collected and stored in a single, central storage component.

- *Decentralized storage*: Monitoring data is retained locally at each host, platform, or service, without aggregation into a central repository.

At Netum, the current monitoring infrastructure corresponds to the centralized storage model. Monitoring data is collected from multiple servers and aggregated into a single, central monitoring server. However, from the perspective of generating useful information about integration traffic data for stakeholders, the choice between centralized and decentralized storage is of lesser significance. The primary concern lies in the ability to extract, process, and present relevant monitoring data in a meaningful and accessible way, regardless of the underlying storage architecture.

4.3.3 Hosting of Monitoring Data

Another architectural consideration in monitoring systems is the hosting location of the monitoring data. Haselböck et al. [25] identify two principal options:

- *Cloud-based hosting*: Involves storing monitoring data in a public or third-party cloud environment.
- *On-premises (self-hosted) deployment*: Monitoring infrastructure components are deployed within the organization's internal network.

The choice between these alternatives is often influenced by organizational policies, regulatory requirements, and the sensitivity of the monitored data.

At Netum, monitoring data is hosted on-premises. This decision is driven primarily by the sensitive nature of the integration traffic and log data, which may include confidential information subject to legal or contractual confidentiality obligations. Storing such data in a public or third-party cloud environment would introduce risks related to data exposure, compliance, and governance. As a result, monitoring infrastructure components are deployed within the organization's secure internal network.

From a functional standpoint, the hosting location does not significantly impact the capabilities of monitoring tools. Whether hosted in the cloud or on-premises, the critical factor remains the capability of the monitoring system to collect, aggregate, and visualize relevant data in a timely and reliable manner.

4.3.4 Distribution of Monitoring Data

The distribution of monitoring data concerns the mechanism and direction through which data is transmitted between monitored components and the monitoring system. Haselböck et al. [25] identify two principal distribution patterns:

- *Pull-based monitoring*: The monitoring system actively queries the monitored components at regular intervals.
- *Push-based monitoring*: The components themselves send monitoring data or alerts when specific conditions are met.

These patterns are discussed in detail in Section 2.5.2, with their implementation at Netum outlined in Section 3.2. Netum's monitoring system adopts a hybrid approach that combines both patterns, allowing them to complement one another and enhance the overall visibility and effectiveness of integration monitoring. Both patterns can independently provide valuable information about integration traffic data, but their combined use results in a more comprehensive and reliable monitoring solution.

4.3.5 Processing of Monitoring Data

The decision guidance model for processing monitoring data outlines design options related to capturing service interactions, determining suitable levels of data aggregation, and performing threshold-based analysis. As discussed earlier, the primary concerns of key stakeholders at Netum include root cause analysis, the examination of service interactions, and the generation of long-term performance and usage reports. To address these concerns, the decision model proposes several processing techniques:

- *Tracing*: Captures the complete execution path of individual requests across services, which helps in identifying bottlenecks and diagnosing issues at a granular level.
- *Call monitoring:* Observes communication patterns and frequencies between services, offering insights into system behaviour and interdependencies.
- *Storing data in its native form*: Retains raw, detailed monitoring data, enabling flexible querying and deep diagnostic capabilities.

- *Storing data in aggregated form*: Compiles summarized metrics (e.g., averages, counts, thresholds), which supports efficient long-term storage and trend analysis.

These techniques are discussed in more detail in Chapter 2. The relationships between concerns and design options are depicted in Figure 4.1.

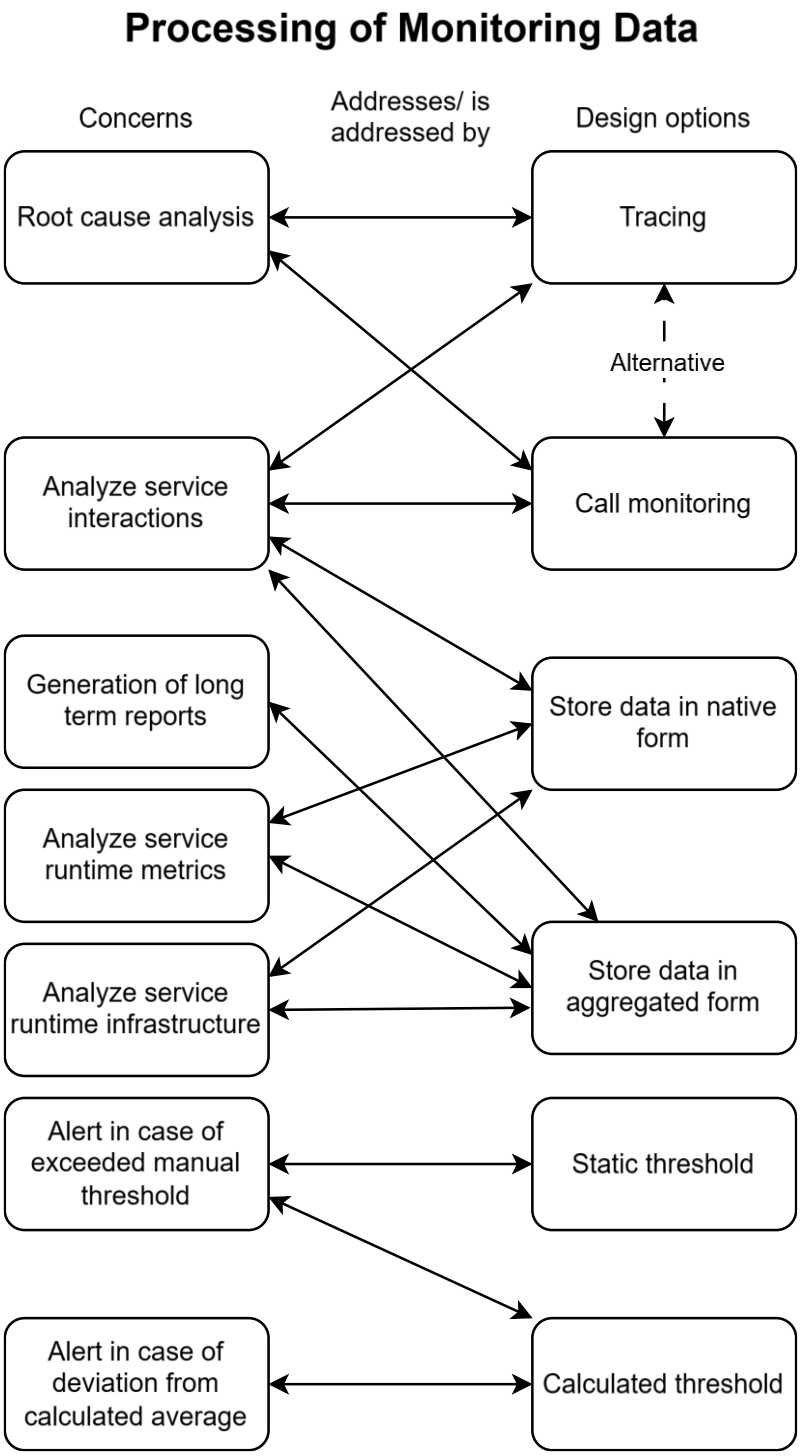

Figure 4.1: Mapping of stakeholder concerns (Recreated from *Decision Guidance Models for Microservice Monitoring* [25])

At Netum, the current monitoring setup only partially supports these approaches. Tracing and call monitoring capabilities are minimal or entirely absent, which significantly limits visibility into integration traffic between services. This limitation poses a challenge for effective root cause analysis and proactive problem resolution.

Most monitoring data is stored in its native form, which is useful for diagnostics when specific problems arise. Certain monitoring data, such as log files, is also aggregated and centralized to support alerting and high-level overviews. While this approach offers basic observability and historical tracking, it does not fully support the proactive or real-time insights that stakeholders increasingly require in complex integration scenarios.

Closing these gaps requires enhanced instrumentation of service interactions and the adoption of more sophisticated processing techniques as outlined in the decision guidance model. The limitations identified here are consistent with the common monitoring challenges in microservice architectures discussed in Section 2.7.1.

4.4 Criteria for Integration Traffic Monitoring Tools

Based on the discussion in previous chapters and the decision guidance models introduced by Haselböck et al. [25], the criteria for selecting a monitoring tool for Netum have been established. These criteria reflect both technical capabilities and organizational constraints. The monitoring tool should support a specific combination of design options that align with stakeholder needs, existing infrastructure, and long-term maintainability.

The primary selection criteria are as follows:

- *Service Level Monitoring*: The tool must facilitate the generation of monitoring data mainly at the application service level. This should ideally occur with minimal changes to the existing microservices. Compatibility with adopted frameworks, such as Spring Boot Actuator, is considered advantageous.
- *On-Premises Deployment Capability*: Due to internal policies governing data sensitivity external SaaS-based monitoring solutions are not permissible. The selected tool must be fully deployable within the organization’s infrastructure.

- *Support for Tracing and Call Monitoring*: The ability to capture and visualize inter-service communications is critical for identifying performance bottlenecks and supporting root cause analysis.
- *Visual Dashboards and Stakeholder Usability*: A web-based user interface that provides intuitive, customizable dashboards is required. This ensures that both technical and non-technical stakeholders can effectively interpret monitoring data.

These criteria form a foundation for evaluating potential tools. The following section examines how the current landscape of monitoring solutions measures up against these requirements.

4.5 Overview of Monitoring Tool Landscape

A broad spectrum of monitoring tools is available today. This examines how well these tools align in general with the previously defined selection criteria. A comprehensive study by Giamattei et al. [1] in 2024 reviewed 71 monitoring tools commonly used in DevOps-oriented microservice environments. Notably, the study excludes SaaS-based tools, focusing exclusively on on-premises solutions.

Key findings relevant to the defined selection criteria include:

- Approximately 20 % of the tools are specifically designed for microservice architectures, a trend increasingly evident in newer solutions.
- 37 % support service-level (user-focused) metrics, while the majority provide system-level metrics such as CPU, memory, and network usage.
- 43 % offer comprehensive, multi-layer monitoring that spans application, microservice, container, and infrastructure levels.
- 42 % include dedicated support for distributed tracing.
- Instrumentation is required in nearly all tools (68 out of 71).
- Visualization and dashboards are widely supported: 66.2 % offer built-in visualization features.

It remains unclear whether existing monitoring tools provide sufficient capabilities for inspecting individual events and messages within integration flows, particularly in Netum's context. This topic is explored in the next chapter through semi-structured interviews with professionals at Netum.

The adoption of various monitoring tools identified in research is discussed in Section 2.7.2. A closer examination reveals specific gaps when addressing the nuanced requirements of deep integration traffic monitoring. It becomes apparent that there is no single, comprehensive "integration traffic monitoring" tool readily available. Instead, the landscape offers a collection of tools, each supporting different combination of design aspects, such as infrastructure health, application performance, or log management.

This situation brings attention to the anti-pattern of "tool obsession," as described in Section 2.1. In the absence of a clearly defined monitoring solution, there is a considerable risk of assembling a fragmented set of tools without fully understanding the primary objective, which is to achieve comprehensive visibility into the flow and integrity of integration traffic. To prevent this outcome, it is essential to establish a precise definition of the monitoring requirements before proceeding with tool selection or solution design.

Most monitoring tools primarily focus on system-level metrics, with fewer than half providing support for user-centric metrics. Similarly, under half of the tools analysed offer dedicated support for distributed tracing. This may be due to the fact that distributed tracing is often less critical in smaller microservice architectures, where the number of services is limited and service invocation chains tend to be short [26]. In contrast, visualization solutions are widely supported in different monitoring solutions.

There is a noticeable lack of in-depth discussion and readily available tooling in the broader market that specifically addresses granular monitoring of application events with respect to data integrity and content validation. Most current solutions continue to emphasize infrastructure and application health, availability, and performance, rather than the validity and content of the data exchanged between components.

This challenge is evident at Netum, where the diverse technology landscape and presence of legacy services mean that a unified monitoring or tracing framework may not be feasible, necessitating the use of complementary approaches such as custom solutions [26].

5 Results

This chapter presents the results of five semi-structured interviews conducted with professionals at Netum Oy, each representing different roles within the organization's integration and monitoring landscape. The aim is to explore practical perspectives on monitoring microservices, with a specific focus on integration traffic. Thematic analysis of the interview data revealed recurring ideas across three areas: design principles, current monitoring practices and challenges, and tool requirements. These themes are discussed in alignment with the research questions and supported by direct quotes from participants.

5.1 Research Method

Semi-structured interviews were the selected research method for gathering qualitative data for this study. This approach was chosen to balance structure and flexibility. Each participant was asked the same core set of seven questions (see Appendices A [English] and B [Finnish]), designed to address the main research questions. At the same time, the format allowed conversations to flow naturally, encouraging deeper exploration of individual perspectives and emerging themes.

The interviews were conducted entirely in Finnish. They were audio-recorded with consent and subsequently transcribed. The transcribed qualitative data were systematically coded using thematic analysis to identify recurring categories and patterns directly emergent from the interview content. To maintain the conversational flow, the order of questions sometimes varied slightly between interviews. The results presented in this chapter have been manually translated into English by the author, with limited use of translation tools to support the process. This may introduce subtle nuances or shifts in meaning compared to the original Finnish dialogue.

Every interviewee was fully informed about the purpose of the interview and the use of recording and provided explicit consent prior to participation. To ensure confidentiality, all results are anonymized; participants are referred to using codes (Interviewee A, B, C, D, and E). All interview recordings will be securely destroyed upon the completion of this thesis.

A total of five senior-level professionals from Netum Oy were interviewed, each selected to represent key roles within the integration and monitoring ecosystem: a developer, an operations team member, an integration architect, a project manager, and a team leader. All

participants have been involved to varying degrees in the development of Netum’s current monitoring solution and are familiar with monitoring practices from their respective perspectives.

This diverse selection aimed to capture a comprehensive range of technical backgrounds, operational perspectives, and monitoring requirements within the organization. All interviews were conducted remotely. It is important to acknowledge that the participants' shared affiliation with Netum may introduce a potential bias into the perspectives shared. However, while all interviewees were familiar with the existing microservices monitoring setup, their distinct roles and varying levels of direct experience were expected to contribute a broad range of perspectives, enriching the overall findings of this research.

5.2 Interviews

Identifying common themes across the interviews presented challenges, largely due to the varied roles within the team and the wide-ranging scope of microservices monitoring. However, several recurring ideas and concerns emerged, reflecting different perspectives on design principles, practical challenges, and tool requirements. All quotations have been translated from Finnish to English. Table 5.1 provides a thematic overview of the interview responses.

Table 5.1: Summary of Common Themes in Microservices Monitoring Interviews

Question	Key Themes Identified
Q1. Key aspects in designing monitoring	Depends on service complexity; focus on infrastructure (interfaces, ports); success/failure tracking; logging and data movement; stakeholder needs and SLA alignment
Q2. Monitoring practices and principles	Black-box thinking; reduce false alerts; relevance of thresholds and alert channels; templates and identifiers help traceability; monitoring should match technical and business layers
Q3. Improvements to current monitoring	Better visibility into traffic and statistics; reduce manual checking; avoid developer-only perspective; anticipate issues; monitoring development is underfunded
Q4. Common undetected issues	Poor logging structure and legacy tools; missing validation layers; errors often caught by users; hard to detect missing or partial data
Q5. Needed data on integration traffic	Track volumes, timing, and anomalies; sensitive data limits logging; focus on what moved and when; detail errors clearly in logs
Q6. Tool requirements	Visual dashboards; automatic analysis; access control; alerts via multiple channels; drill-down capability; simplicity for non-technical users
Q7. Presentation of data	Dashboards, email alerts, webhooks; clarity over completeness; identifier visibility; role-specific views; emphasis on actionable summaries

The key themes identified from the interviews have been consolidated into three categories, each of which corresponds to one of the research questions. These categories are explored in the following sections, with representative quotes from participants used to illustrate common viewpoints and recurring concerns.

5.2.1 Design Principles for Monitoring Microservices

The first theme explores the foundational principles guiding the design of monitoring solutions in microservices environments. This contributes to understanding Research Question 1, “What are the key principles for designing effective monitoring solutions in microservices architectures?”, by exploring how practitioners approach monitoring design in real-world contexts.

Interviewees emphasized that monitoring must be tailored to the specific characteristics of each integration, considering system complexity, customer expectations, and Service Level Agreements (SLAs). Additional considerations included the need for end-to-end visibility, the role of logging in error tracking, and the importance of balancing transparency with data protection.

All interviewees emphasized that monitoring design is typically approached on a case-by-case basis, shaped by the specific needs of the customer and the nature of the service. One recurring observation was that customers often treat integrations as a "black box", showing concern only for what goes in and comes out, rather than the internal workings of the system.

> *What goes in and what comes out must be taken into account.* (Interviewee A)

> *In an error situation, all incoming and outgoing exception errors must be revealed.* (Interviewee B)

Speed and reliability were highlighted as essential, often tied to Service Level Agreements (SLAs). In this context, logging plays a critical role, especially for tracking traffic and behaviour over time, though it must be balanced with privacy considerations.

> *Logging is an essential part of data transfer, but when logging, it is important to consider the amount of data, as information can easily get lost in it.* (Interviewee C)

> *We should be able to tell what information has been processed and when it has been processed.* (Interviewee D)

> *The information should include key error parameters that help save developers time, rather than just providing a stack trace.* (Interviewee B)

The importance of matching monitoring design to both the client's expectations and the technical constraints of their systems was consistently mentioned.

> *The error management of the receiving system must be taken into account.* (Interviewee B)

Finally, several interviewees noted that monitoring should span across all layers of the system, from infrastructure to application logic.

> *APIs, ports used, API protocols...* (Interviewee A)

> *Effective monitoring of integration traffic requires the right level of oversight and a clear understanding of the interdependencies between technical and traffic monitoring*. (Interviewee B)

The design principles articulated by participants highlight a pragmatic, context-dependent approach, prioritizing service characteristics, customer expectations, and data integrity in monitoring microservices.

5.2.2 Monitoring Practices and Challenges

The second theme focuses on current monitoring practices and the challenges associated with gaining visibility into integration traffic. This section contributes to understanding Research Question 2, “How can existing monitoring systems be improved to provide greater visibility into integration traffic flows?”, by identifying limitations in existing systems and areas where improvements may be possible.

Interviewees raised concerns about ineffective alerting, incomplete data validation, limited traceability, and the absence of standardized practices. The need for better integration between monitoring and business processes, as well as more proactive issue detection, was also frequently emphasized.

Participants consistently highlighted practical challenges in current monitoring practices.

> *The basic principle is to monitor success and failure and key indicators that have been defined either at the business level or at the system level.* (Interviewee B)

> *Defining manual thresholds easily causes false positives.* (Interviewee D)

The configuration and management of alerts, particularly the need to minimize false positives and ensure relevance, emerged as a recurrent sub-theme.

> *It is important to reduce unnecessary alerts; the response time decreases quickly if there are unnecessary alerts. In practice, they should not be generated at all.* (Interviewee E)

> *Email alerts are a pretty functional system, but some services have web hooks, for example.* (Interviewee A)

It was noted that while customers often appreciate visibility into message traffic, this seldom translates into a formal contractual requirement.

> *Integration traffic monitoring is rarely considered, but customers view it positively if it is clearly conceptualized*. (Interviewee B)

From a development perspective, monitoring is primarily used to identify and troubleshoot issues.

> *The development of monitoring is quite developer-driven, with nothing but code to worry about*. (Interviewee B)

However, the tools should be easy to use and not add extra complexity to the workflow.

> *Sometimes it feels like we're monitoring a monitoring system.* (Interviewee C)

The role of the customer in monitoring process was also frequently emphasized.

> *Monitoring should align with the customer's business processes, and relevant points of contact must be clearly identified.* (Interviewee E)

> *What role does integration play in exploring the content of information? Data protection issues are easily encountered.* (Interviewee D)

Validation was also discussed.

> *Errors are often easy to spot, but if half of the data is missing, it is quite difficult to catch* (Interviewee D)

> *A validation layer can be built into the integration, which checks the content of the message for missing information, among other things.* (Interviewee E)

Lastly, limited financial resources were mentioned as a barrier to developing more robust or comprehensive monitoring solutions.

> *Most customers say they are interested in principle, but do not necessarily follow through in practice*. (Interviewee D)

> *Development costs should be allocated as part of the overall project budget alongside other activities*. (Interviewee E)

Overall, current monitoring practices, as described by interviewees, are characterized by a desire for greater efficiency, proactive issue detection, and better alignment with business needs, often constrained by practical limitations such as alert fatigue and resource allocation.

5.2.3 Requirements for Effective Monitoring Tools

The third theme synthesizes insights on the essential features and design requirements of monitoring tools in microservices contexts. This section contributes to understanding Research Question 3, "What are the key requirements for a monitoring tool to effectively support integration traffic monitoring in microservices architectures?", by outlining user expectations for functionality, usability, and flexibility.

Interviewees identified the need for visual dashboards, automated analysis, configurable alerting, and access control. Effective tools should also support drill-down capabilities, accommodate both technical and non-technical users, and present actionable, role-specific summaries of integration traffic.

When discussing ideal characteristics of monitoring tools, several technical requirements emerged. The general premise was that monitoring tool should be easy to use with easy-to-understand user interface.

> *A good overview that reveals if there are any problems, for example.* (Interviewee D)
>
> *Visualization is essential, some integration platforms have this built in, but they can be quite limited.* (Interviewee E)
>
> *Possible initial screening and automatic analysis.* (Interviewee A)
>
> *Message identifiers should be visible.* (Interviewee C)

It was noted that the monitoring solution can always be expanded later.

> *It's easy to come up with new monitoring features.* (Interviewee E)
>
> *It depends on how far you want to take it, but for example business activity monitoring.* (Interviewee B)

Participants acknowledged the existence of numerous integration platforms and monitoring tools. It was noted that these systems are sometimes not directly under their operational control but can still provide valuable data.

> *Even if the server is not under our control, we can still get useful information about how the integrations work.* (Interviewee E)

Developers reportedly do not often utilize these monitoring tools extensively, but it was noted that they could potentially aid in error tracking and diagnostics.

In essence, the requirements for monitoring tools centre on usability, visualization and automation, and flexible data presentation tailored to diverse user needs within the organization.

5.2.4 Summary of Interviews

The interviews were centred around three main themes concerning the monitoring of microservices: design principles, current practices and challenges, and tool requirements.

Participants described monitoring design as highly context-specific, influenced by factors such as the complexity of the integration, customer needs, and Service Level Agreements. A recurring observation was that monitoring tends to focus on what enters and exits a system rather than its internal workings. This “black box” approach often reflects the customer’s expectations. Logging was considered essential for tracking errors and understanding traffic flows, though interviewees stressed the importance of balancing transparency with data protection. Monitoring solutions were generally developed on a case-by-case basis, considering both infrastructure-level and business-level needs.

Regarding current practices, interviewees pointed to several challenges that limit the effectiveness of existing monitoring systems. These included inconsistent validation mechanisms, excessive or poorly targeted alerts, and limited traceability of issues. Monitoring was described as developer-driven, lacking sufficient integration with broader business processes. Participants also noted that while customers express interest in visibility, this rarely translates into concrete requirements or investments. Resource constraints, particularly a lack of funding, were mentioned as a barrier to improving current systems.

In terms of tool requirements, participants emphasized the need for user-friendly solutions that offer visual dashboards, automated diagnostics, and configurable alerts. Tools should support both technical and non-technical users by presenting data in a clear and accessible way. The ability to trace individual messages, drill down into traffic flows, and display role-specific summaries was considered particularly important. Participants also highlighted the importance of flexibility, noting that monitoring solutions can be adapted and scaled as needed.

Taken together, the interviews reflect a shared recognition of the importance of effective monitoring, along with a range of practical concerns and aspirations for improvement. While all participants were familiar with the existing monitoring setup at Netum, their differing roles contributed to a broad and nuanced understanding of the organization’s monitoring needs.

5.3 Discussion

The interview findings align with the themes discussed in earlier chapters and prior research. Questions and responses remain general, focusing on monitoring principles rather than deep implementation-level details.

In response to RQ1, participants highlight key principles such as service level agreements, black-box monitoring, multi-level monitoring, logging, alerting, dashboards, and visualization. These fall under technical monitoring, discussed in Section 2.3.2, and are also relevant for monitoring integration traffic. In addition to reaffirming foundational principles covered in Chapter 2, participants raise less frequently addressed concerns including budgeting constraints, customer expectations, and the role of integration in managing customer data—topics that receive limited attention in the literature.

The interview findings emphasize application service-level considerations that match the criteria in the decision guidance model (Section 4.4).

For RQ2, participants stress the importance of structured logging formats and necessary trace parameters for improving visibility into integration traffic flows. While they see traceability and message identification as essential, they recognize these alone are not sufficient. These observations correspond with Section 2.5.3, which defines monitoring levels, and Section 2.7.1, which identifies log analysis and tracing as the most common fault localization strategies in microservices. Enhanced logging and tracking emerge as the foundation for improvements to current monitoring systems. These topics represent application-level concerns and align with the criteria outlined in the decision guidance model in Section 4.4.

Regarding RQ3, participants express no strong preferences for specific monitoring tools. Instead, they expect tools to meet basic requirements and allow for future expansion. Effective tools should generate readable, plain-language logs and present data in a clear, understandable format. Although distributed tracing is not explicitly mentioned, its role is implicitly acknowledged. Tools should support alerting, visualization, and unique message identification at minimum.

These requirements align with the visual dashboards and stakeholder usability criteria in Section 4.4. Visual presentation of integration data is necessary not only for technical teams but also for non-technical stakeholders. While the interviews do not mention tracing tools explicitly, the emphasis on logging service interactions supports the decision guidance model's criteria for inter-service communication capture and root cause analysis (Section 4.4). The ability to visualize these interactions is critical for identifying performance bottlenecks.

On-premises deployment capability does not arise in interviews, as it depends on case-specific constraints, consistent with the discussion in Section 4.4. Similarly, participants do not

discuss advanced monitoring tool capabilities, though many such tools are available and evaluated in Section 4.5. This suggests that suitable solutions can be found among existing tools, provided they support the identified requirements.

Netum's current monitoring setup, described in Chapter 3, is largely reactive and infrastructure focused. Tools like Zabbix and Nagios monitor system-level metrics but provide little visibility into integration traffic or service interactions. Some microservices include self-monitoring and alerting, and custom scripts scan logs for known errors. However, the setup lacks centralized oversight, traceability, and the ability to analyse message flows or detect emerging issues across integrations.

To improve this, Netum should adopt a lightweight monitoring tool that visualizes plain-text errors with clear identifiers and supports basic tracking of integration traffic. Suitable options are discussed in Section 2.7.2. Even a simple solution would enhance visibility into data flows and move the organization toward more proactive monitoring, addressing the reactive gaps outlined in Section 2.5.2.

6 Conclusion

This chapter outlines key findings, acknowledges study limitations, and suggests future research. Limited by a small sample and single company focus, the results are not broadly generalizable. Future work should explore diverse environments, stakeholder-friendly views, and tool comparisons to support more effective monitoring.

6.1 Summary of Findings

This study, conducted in collaboration with Netum Oy, explores how monitoring systems can better support visibility into integration traffic within microservices architectures. The findings demonstrate that while general monitoring principles remain relevant, they must be extended with service-level instrumentation and enhanced traceability to address integration-specific challenges.

The research addresses the following questions:

1. What are the key principles for designing effective monitoring solutions in microservices architectures?
2. How can existing monitoring systems be improved to provide greater visibility into integration traffic flows?
3. What are the key requirements for a monitoring tool to effectively support integration traffic monitoring in microservices architectures?

In response to the first question, effective monitoring practises are well studied subject. This study shows that integration monitoring follows these basic principles. Effective monitoring relies on structured data collection, service-level instrumentation, and support for decentralized environments. These principles align with both existing literature and practical experience. However, the interviews surface additional factors such as customer expectations, project budgets, and organizational responsibilities that remain underrepresented in academic sources but strongly influence monitoring in real-world settings.

The second question focuses on visibility into integration traffic. In the case of Netum, existing systems lack the detail and context required to trace data between services which is a common issue identified in research. The results indicate that structured logging and

consistent trace identifiers are essential for exposing message paths and enabling faster fault isolation.

The third question defines the baseline capabilities required of monitoring tools. Based on the interviews and decision guidance models used, these include human-readable logs, unified formatting, alerting, visualization, and message-level traceability. A monitoring tool must also support the ability to distinguish and correlate individual message flows across services, rather than merely reporting on infrastructure metrics.

6.2 Evaluation and Limitations

The research is limited by the small number of interviewees and its exclusive focus on a single company, which may constrain the broader applicability of the results. Conducting interviews only in Finnish and translating them into English might introduce subtle differences in meaning. The study's qualitative nature also means findings are descriptive and exploratory rather than statistically generalizable. Further empirical studies, including larger samples and quantitative validation, would strengthen the conclusions.

6.3 Future Work

While this study provides a foundation for improving integration traffic monitoring in microservices architectures, several areas remain open for further research and development.

One important area is the evaluation of monitoring tools in a variety of production environments, including cloud-native, hybrid, and containerized deployments. As this thesis focused on on-premises systems at Netum Oy, conducting comparative studies across different infrastructure models would help assess the generalizability and applicability of the proposed recommendations.

Further work could also involve the development of integration traffic data views designed for non-technical stakeholders. These views would present integration-related information in a simplified, plain-text format and enable correlation with relevant business metrics. This would improve the accessibility and usefulness of monitoring data for users without deep technical expertise.

In addition, a systematic comparison of available monitoring tools would help identify the most suitable solutions for integration traffic monitoring. Such comparisons could focus on

aspects such as traceability, usability, performance, and compatibility with common integration frameworks.

Advancing these areas would contribute to the design of more robust, context-aware, and stakeholder-inclusive monitoring systems in microservice-based environments.

References

[1] L. Giamattei et al., ‘Monitoring tools for DevOps and microservices: A systematic grey literature review’, J. Syst. Softw., vol. 208, p. 111906, Feb. 2024, doi: 10.1016/j.jss.2023.111906.

[2] G. Hohpe and B. Woolf, Enterprise integration patterns: Designing, building, and deploying messaging solutions, 17. print. in The Addison-Wesley signature series. Boston Munich: Addison-Wesley, 2013.

[3] 610.12-1990 IEEE Standard Glossary of Software Engineering Terminology. s.l: IEEE / Institute of Electrical and Electronics Engineers Incorporated.

[4] J. Turnbull, The Art of Monitoring. 2022. Accessed: Dec. 16, 2024. [Online]. Available: https://www.slideshare.net/slideshow/the-art-of-monitoring-2016pdf/252142218

[5] Q. Xiang et al., ‘No Free Lunch: Microservice Practices Reconsidered in Industry’, Jun. 14, 2021, arXiv: arXiv:2106.07321. doi: 10.48550/arXiv.2106.07321.

[6] X. Zhou et al., ‘Revisiting the practices and pains of microservice architecture in reality: An industrial inquiry’, J. Syst. Softw., vol. 195, p. 111521, Jan. 2023, doi: 10.1016/j.jss.2022.111521.

[7] S. Yangui, I. Bouassida Rodriguez, K. Drira, and Z. Tari, Eds., Service-Oriented Computing: 17th International Conference, ICSOC 2019, Toulouse, France, October 28–31, 2019, Proceedings, vol. 11895. in Lecture Notes in Computer Science, vol. 11895. Cham: Springer International Publishing, 2019. doi: 10.1007/978-3-030-33702-5.

[8] ‘Google - Site Reliability Engineering’. Accessed: Jan. 30, 2025. [Online]. Available: https://sre.google/sre-book/table-of-contents/

[9] M. Julian, Practical Monitoring. O’Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

[10] F. Pina, J. Correia, R. Filipe, F. Araujo, and J. Cardroom, ‘Nonintrusive Monitoring of Microservice-Based Systems’, in 2018 IEEE 17th International Symposium on Network Computing and Applications (NCA), Cambridge, MA: IEEE, Nov. 2018, pp. 1–8. doi: 10.1109/NCA.2018.8548311.

[11] H. Knoche and W. Hasselbring, ‘Drivers and Barriers for Microservice Adoption – A Survey among Professionals in Germany’, Enterp. Model. Inf. Syst. Archit. EMISAJ, p. 1:1-35 Pages, Jan. 2019, doi: 10.18417/EMISA.14.1.

[12] C. Ibsen and J. Anstey, Camel in Action. Greenwich, Conn: Manning, 2011.

[13] ‘Data considerations for microservices - Azure Architecture Center’. Accessed: Feb. 12, 2025. [Online]. Available: https://learn.microsoft.com/en-us/azure/architecture/microservices/design/data-considerations

[14] W. Fan, Z. Han, Y. Zhang, and R. Wang, ‘Method of Maintaining Data Consistency in Microservice Architecture’, in 2018 IEEE 4th International Conference on Big Data Security on Cloud (BigDataSecurity), IEEE International Conference on High Performance and Smart Computing, (HPSC) and IEEE International Conference on Intelligent Data and Security (IDS), Omaha, NE, USA: IEEE, May 2018, pp. 47–50. doi: 10.1109/BDS/HPSC/IDS18.2018.00023.

[15] M. C. Borges, J. Bauer, S. Werner, M. Gebauer, and S. Tai, ‘Informed and Assessable Observability Design Decisions in Cloud-native Microservice Applications’, in 2024 IEEE 21st International Conference on Software Architecture (ICSA), Jun. 2024, pp. 69–78. doi: 10.1109/ICSA59870.2024.00015.

[16] M. Waseem, P. Liang, M. Shahin, A. D. Salle, and G. Márquez, ‘Design, Monitoring, and Testing of Microservices Systems: The Practitioners’ Perspective’, J. Syst. Softw., vol. 182, p. 111061, Dec. 2021, doi: 10.1016/j.jss.2021.111061.

[17] S. Newman, Building microservices: Designing fine-grained systems, First Edition. Beijing Sebastopol, CA: O’Reilly Media, 2015.

[18] C. Richardson, Microservices patterns: with examples in Java. Shelter Island, NY: Manning Publications, 2019.

[19] Y. Jiang, N. Zhang, and Z. Ren, ‘Research on Intelligent Monitoring Scheme for Microservice Application Systems’, in 2020 International Conference on Intelligent Transportation, Big Data & Smart City (ICITBS), Vientiane, Laos: IEEE, Jan. 2020, pp. 791–794. doi: 10.1109/ICITBS49701.2020.00173.

[20] J. Opara-Martins, R. Sahandi, and F. Tian, ‘Critical analysis of vendor lock-in and its impact on cloud computing migration: a business perspective’, J. Cloud Comput., vol. 5, no. 1, p. 4, Dec. 2016, doi: 10.1186/s13677-016-0054-z.

[21] D. Moldovan, G. Copil, H.-L. Truong, and S. Dustdar, ‘MELA: Monitoring and Analyzing Elasticity of Cloud Services’, in 2013 IEEE 5th International Conference

on Cloud Computing Technology and Science, Bristol, UK: IEEE, Dec. 2013, pp. 80–87. doi: 10.1109/CloudCom.2013.18.

[22] jolokia/jolokia. (Jan. 23, 2025). Java. Jolokia - JMX on Capsaicin. Accessed: Jan. 23, 2025. [Online]. Available: https://github.com/jolokia/jolokia

[23] 'Hawtio - A modular web console for managing your Java stuff'. Accessed: Jan. 23, 2025. [Online]. Available: https://hawt.io/

[24] 'ActiveMQ'. Accessed: Jan. 23, 2025. [Online]. Available: https://activemq.apache.org/

[25] S. Haselbock and R. Weinreich, 'Decision Guidance Models for Microservice Monitoring', in 2017 IEEE International Conference on Software Architecture Workshops (ICSAW), Gothenburg, Sweden: IEEE, Apr. 2017, pp. 54–61. doi: 10.1109/ICSAW.2017.31.

[26] B. Li et al., 'Enjoy your observability: An industrial survey of microservice tracing and analysis', Empir. Softw. Eng., vol. 27, no. 1, p. 25, Jan. 2022, doi: 10.1007/s10664-021-10063-9.

Appendix A Interview Questions

Questions:

1. What are the important things to consider when designing monitoring for new or existing microservices?
2. What principles and practices are important in monitoring?
3. How would you improve our current monitoring setup for microservices?
4. What kind of integration-related problems often go unnoticed and why?
5. What kind of information, if any, is required about integration traffic?
6. What features should an integration traffic monitoring tool have?
7. How should data related to integration traffic be presented?

Appendix B Haastattelurunko

Kysymykset:

1. Mitkä asiat ovat tärkeitä, kun suunnitellaan valvontaa uusille tai olemassa oleville mikropalveluille?
2. Minkälaiset käytännöt ja periaatteet ovat tärkeitä monitoroinnissa?
3. Miten parantaisit nykyistä mikropalveluiden valvontaratkaisuamme?
4. Millaiset integraatioihin liittyvät ongelmat jäävät usein huomaamatta ja miksi?
5. Minkälaista tietoa integraation tietoliikenteestä tarvitaan (jos tarvitaan)?
6. Mitä ominaisuuksia integraatioliikenteen valvontatyökalussa tulisi olla?
7. Miten integraatioliikenteeseen liittyvä data tulisi esittää?

Appendix C Notice of AI usage in the thesis

Generative artificial intelligence (AI or Gen AI) has been utilized in the process of writing this thesis. However, it has only been used as a writing aid and not to produce new content. Its use is limited to generation of ideas, expressions, improvement of grammar and content already written by the author. The arguments, analysis, and studies presented in this thesis are written by the researcher, with AI only acting as an assistive tool for writing.

I am aware that I am fully responsible for the content of my entire thesis, including the parts generated using artificial intelligence.