National College of Ireland

Exploring Lightweight Monitoring Tools for Microservices in Cloud-Based Architectures

MSc Research Project
MSc in Cloud Computing

Allen Joy
Student ID: 23202351

School of Computing
National College of Ireland

Supervisor: Shaguna Gupta

National College of Ireland
Project Submission Sheet
School of Computing

National College of Ireland

Student Name:	Allen Joy
Student ID:	23202351
Programme:	MSc in Cloud Computing
Year:	2024
Module:	MSc Research Project
Supervisor:	Shaguna Gupta
Submission Due Date:	12/12/2024
Project Title:	Exploring Lightweight Monitoring Tools for Microservices in Cloud-Based Architectures
Word Count:	7393
Page Count:	27

I hereby certify that the information contained in this (my submission) is information pertaining to research I conducted for this project. All information other than my own contribution will be fully referenced and listed in the relevant bibliography section at the rear of the project.

ALL internet material must be referenced in the bibliography section. Students are required to use the Referencing Standard specified in the report template. To use other author's written or electronic work is illegal (plagiarism) and may result in disciplinary action.

Signature:	
Date:	28th January 2025

PLEASE READ THE FOLLOWING INSTRUCTIONS AND CHECKLIST:

Attach a completed copy of this sheet to each project (including multiple copies).	☐
Attach a Moodle submission receipt of the online project submission, to each project (including multiple copies).	☐
You must ensure that you retain a HARD COPY of the project, both for your own reference and in case a project is lost or mislaid. It is not sufficient to keep a copy on computer.	☐

Assignments that are submitted to the Programme Coordinator office must be placed into the assignment box located outside the office.

Office Use Only	
Signature:	
Date:	
Penalty Applied (if applicable):	

Exploring Lightweight Monitoring Tools for Microservices in Cloud-Based Architectures

Allen Joy
23202351

Abstract

This research aims to analyze and evaluate lightweight monitoring tools for microservices in cloud architectures. Microservices bring unique challenges like service discovery, data consistency, and effective communication that traditional monitoring tools fail to address efficiently. The study adopts a mixed-methods approach, integrating quantitative experiments and qualitative analysis to assess tools like Prometheus, Grafana, Telegraf, and Jaeger. Key evaluation metrics include performance overhead, real-time data processing capabilities, and scalability. The proposed cloud-based architecture leverages AWS services for containerization, networking, and data management, while incorporating the lightweight monitoring tools for deep visibility and real-time analytics. The findings highlight the advantages of this approach in terms of scalability, flexibility, and low-overhead monitoring. However, challenges remain around system complexity, data collection overhead, and vendor lock-in. Future research could explore distributed monitoring architectures, further optimization of resource utilization, and tighter integration with DevOps processes.

1 Introduction

1.1 Research Background

The existence of cloud computing and a microservices architecture has dramatically shifted modern application development and deployment, especially with the current dominance of hyperscale providers including Amazon, Microsoft, and Google. Microservices, as opposed to traditionally constructed monolithic applications built together in one unit, instead break up the functionalities into smaller, independently deployable services. This modularity enables easy scalability of each component, support during maintenance, and fast updates that significantly improve the response time to technological shifts. Containerization has become crucial in deploying microservices since it is relatively more flexible and resource-efficient than a traditional virtual machine. Second, Kubernetes becomes the normal for orchestrating these containers by automating deployment, scaling, and management. Still, the complexity managed by a distributed microservices environment brings unique challenges such as Service discovery, data consistency, and effective communication channels across services catchpoint (2024). These architectures become the focus of monitoring primarily because of fault detection, health checks, and failure recovery as a contributor to sustaining high availability and resilience in cloud-native designs. All

the classical monitoring tools, however, are resource-intensive and lose the efficiency benefits of microservices Calderón-Gómez et al. (2021). Using lightweight monitoring tools is an effective strategy for cloud-based architectures. The identification of performance overhead-lightweight tools is used to support real-time data processing, efficient scaling, and dependable operations within dynamic cloud environments. Lightweight monitoring is important in making complex operational issues become manageable for microservices, without sacrificing robust performance and reliability but not at the expense of scalability and flexibility that makes microservices valuable in cloud-based systems.

1.2 Research Aim

The research aims to analyze and evaluate lightweight monitoring tools for microservices in cloud architectures. Microservices are dynamic and have constraints. Hence, traditional monitoring tools incur a tremendous amount of performance overhead and limit scalability and operational efficiency. This research identifies solutions that consume as little as possible yet allow real-time processing of data and scaling of services. This paper shall identify the impacts of such tools on developing practical monitoring strategies that enhance the resilience and performance of microservices through system reliability, responsiveness, and resource utilization in a cloud environment.

1.3 Research Objective

- To identify lightweight monitoring tools that are compatible with microservices in cloud-based architectures.
- To test the selected monitoring tools in terms of performance overhead, resource consumption, and scalability.
- To test whether it can process real-time data in a dynamic microservices environment.
- To analyze the effects of lightweight monitoring on system reliability and responsiveness over cloud platforms.

1.4 Research Questions

- What are the best lightweight monitoring tools for controlling microservices in cloud architectures?
- How do they compare in terms of performance overhead, real-time data processing capabilities, and scalability?

1.5 Problem statement

With the scale, flexibility, and applications for rapid deployment provided in the cloud-based architecture, it is hard to track down highly distributed systems, with traditional monitoring tools that require a lot of computations, thereby creating too much overhead and reducing potential microservice benefits for both performance and scalability, this may turn out to be extremely critical in efficiency-resource-constrained or high demand. The required state is a light monitoring solution that consumes fewer resources and

still captures real-time performance data effectively, allows scalability, and maintains the system's reliability Chamari et al. (2023). Efficient low-overhead monitoring solutions fill the gap between microservices' operational needs and the available tools, hence impacting the responsiveness and resilience of the system. This research fills this gap in exploring, evaluating, and identifying lightweight monitoring tools optimized for resource usage that promise dependable performance and it allows cloud-based microservices to run at peak efficiency with no reliability tradeoff.

1.6 Research Significance

This research fills the critical gap in cloud-based architecture microservices, especially with efficient monitoring without resource-intensive usage. Many organizations have begun adopting microservices from scalability, flexibility, and rapid deployment perspectives. Efficient monitoring solutions, therefore, become critical for the continued performance and reliability of such systems. Traditional monitoring tools always incur large performance overhead, which generally degrades the efficiency of microservices applications, mainly in resource-intensive environments Giamattei et al. (2024). This paper identifies and discusses lightweight monitoring tools that have the potential to enable data tracking in real-time as well as rapid scalability of the system without compromising performance. The findings will meet the interest of cloud service providers, developers, and enterprise organizations fostering resourceful exploitation, system reliability, and cost-effective operations. In addition to leveraging the field of microservices in the cloud, the research contributes to resilient and efficient infrastructures in the growingly digitalized world.

1.7 Motivation

The motivation behind this research is the increasing need for scalable, efficient monitoring solutions in cloud-based microservices architectures. Traditional monitoring tools are comprehensive but generate considerable resource overhead and thus nullify the benefits of inherent flexibility and scalability in microservices. There is a pressing need for lightweight tools that can monitor system health and performance without hampering efficiency as microservices quickly become a core component in modern applications. It focused on the gap identified above, and it sought to investigate the monitoring tool that can process data in real-time and with reliability into a cloud-native system's peak performance, resilience, and resource utilization.

1.8 Structure

The research design to study lightweight monitoring tools for microservices on cloud-based architectures begins with an Introduction that provides the background and scope of the study. This then follows with a Literature Review or Related Work covering the prior work on monolithic and microservices architectures and the monitoring method plus the demand for low overhead. Then the Methodology discusses the different research methodologies that were considered in assessing and evaluating the lightweight monitoring tools. This paper presents the Findings and Analyses of the results from evaluation experiments conducted to evaluate lightweight monitoring tools in the effectiveness of their performance on tools in cloud environments. Finally, the Conclusion draws key

insights and summarizes the potential influence of the optimization of performance and resilience of microservices posed by lightweight monitoring tools.

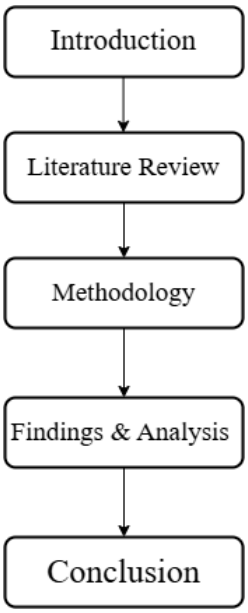


Figure 1: Research Structure

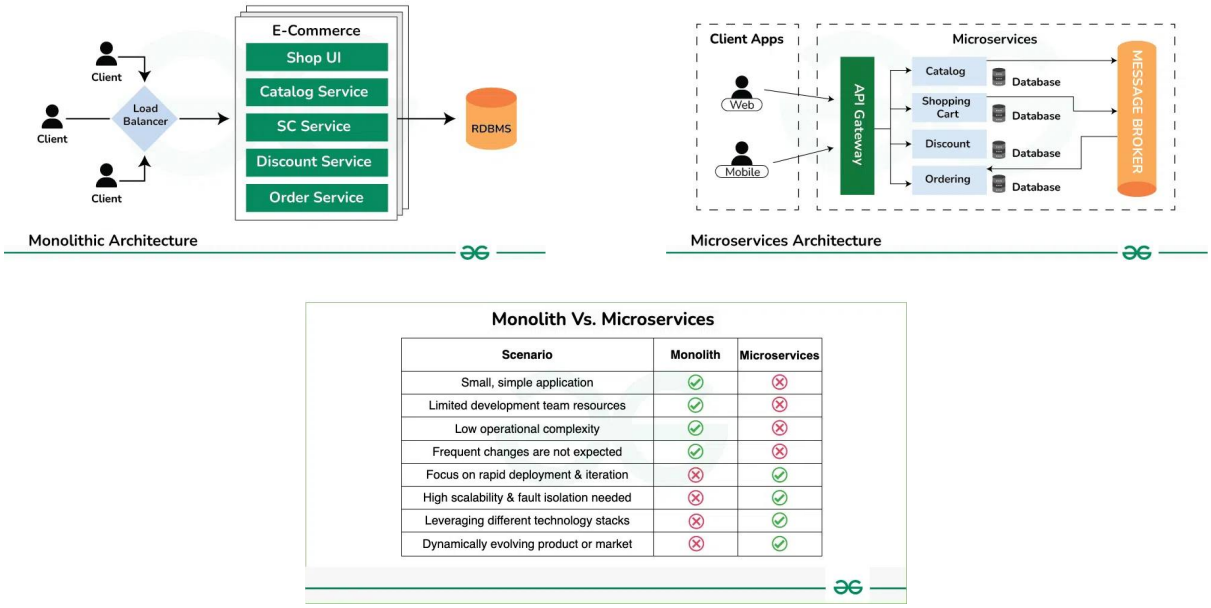
It is where research in lightweight monitoring tools for microservices within cloud-based architectures came in to answer their special needs. Unlike the traditional monolithic, Microservices allow for independent scaling and faster updates but bring up service discovery and data consistency. Containerization and tools like Kubernetes help manage these environments; however, current monitoring tools often introduce substantial overhead that defeats the purpose of microservices. This research attempts to identify low-overhead monitoring solutions that are efficient, scalable, and resilient in tracking real-time data. The results are bound to be valuable for cloud service providers, developers, and organizations looking for cost-effective, high-performance microservices deployments.

2 Related Work

The literature review gives the idea of the past research. The difference between monolithic and microservices approaches has turned out to be an emerging issue in the rapidly shifting landscape of software architecture. Monolithic architectures rely on a single codebase. It is easier to deploy but fails in terms of scalability and complexity when an application grows. On the other hand, microservices architectures break applications into standalone services, improving scalability, flexibility, and resiliency. This literature review reviews several studies about microservices and how lightweight monitoring solutions play a significant part in cloud environments. It focuses on the importance of adequate performance monitoring tools that cause less overhead of resources without losing the reliability of a system, especially for cloud-based applications that bear dynamic loads and various forms of service interactions.

2.1 Thematic Literature Review

2.1.1 Monolith and Microservices



Scenario	Monolith	Microservices
Small, simple application	✓	✗
Limited development team resources	✓	✗
Low operational complexity	✓	✗
Frequent changes are not expected	✓	✗
Focus on rapid deployment & iteration	✗	✓
High scalability & fault isolation needed	✗	✓
Leveraging different technology stacks	✗	✓
Dynamically evolving product or market	✗	✓

Figure 2: Monolithic vs Microservices Architecture(Source:geeksforgeeks.org, 2024)

A traditional software design approach is often referred to as a monolithic architecture, where components of an application, from the user interface to business logic and data access, are tightly integrated into one unified system Doubletapp (2023). This is generally quite simple to develop and deploy in that the entire application would be developed, tested, and then deployed as one unit. Monolithic architecture excels in simplicity; hence it is ideally suited for small and medium-sized projects where swift deployment and straightforward maintenance would be the prime priority. However, with applications going large and getting complex, monolithic structures become quite a challenge in management, scalability, and upgradation mostly leading to delays in deployment and also poor flexibility. A Microservices Architecture structures an application as an amalgamation of several small, independent services Holopainen (2021). A microservices architecture consists of services that can do a specific functionality, each of which can be independently developed, deployed, and scaled. It offers greater flexibility, scalability, and resilience because different teams work on different services using varied technology stacks. If one service fails, it doesn't bring down the entire application. However, handling different services poses a considerable overhead of network communications between the services, which might be a heavy burden for development and maintenance GeeksforGeeks (2020). This architecture is ideal for large and complex applications with independent scalability and agility.

2.1.2 Analyzing Microservices

System reliability requires the analysis of microservices, especially in complex, cloud-based architectures, where multiple services are combined to deliver a seamless user experience. Since microservices perform specific functions independently, it is essential to

maintain visibility into their performance so that potential issues are identified before they affect the larger application. Effective analysis would include monitoring each service's health, availability, and performance metrics, especially the API calls that facilitate communication between services Kosińska et al. (2023). Without real-time insight, organizational risk includes the "digital blind spot," where faults in one service propagate and feed back into the entire application overall. Traditional monitoring tools incur significant resource demands, impeding the flexibility and scalability granted by microservices. Monitoring tools that are lightweight let organizations monitor with efficiency- and low overhead while capturing desired metrics without interfering with overall system performance Loreti et al. (2020). This fits the aim of optimizing environments for cloud-based microservices, where resource efficiency matters. Lightweight monitoring tools would allow you to track API-related interactions, find inefficient endpoints, and be certain that your microservices run at their best, even as the system scales in either direction. It has considered lightweight solutions where strong yet resource-aware monitoring in cloud-based applications enhances systems in terms of resilience and brings out quicker, data-driven responses to emergence-related problems.

2.1.3 Real-time Data Processing and System Impact

Resilience in microservices is crucial for having the best performance, mainly in response to changing volumes of transactions and dynamic scaling conditions. The monitoring of microservices must be tested at all these various load levels to identify bottlenecks and ensure their services remain responsive enough during shifting demands Lähtevänoja (2021). Lightweight monitoring tools such as OpenTelemetry can adapt their sampling and collect granular data for real-time insights without burdening the system. It therefore means that the development teams and operation teams can go about analyzing the monitoring accuracy while updating the services. Here, precision is the order of the day, or there will be significant disruptions. Data consistency in normal and failure scenarios must also be addressed by monitoring solutions. Distributed tracing tools like Catchpoint and Istio when used with a service mesh provide visibility into latency and error rates, helping to track root causes and monitor the impacts of updates in service performance. Measuring end-to-end latency across services helps in determining how long delays of one service impact the general system performance, so teams can make data-driven adjustments towards reliability and great user experiences Moreira (2023). Lightweight microservices monitoring tools that align with cloud computing ensure applications are resilient and agile. They are, therefore prepared to handle the complexities of the new, service-oriented architecture of modern applications.

2.1.4 Performance Testing and Resource Analysis

Performance testing and resource analysis are important in a microservices architecture for optimal system operations and efficient resource utilization. Proper performance testing is considered the evaluation of the performance impact of monitoring tools on individual microservices so that such monitoring tools do not adversely affect the application performance. By measuring CPU and memory utilization, organizations can see how much computational overhead such monitoring processes introduce Noferesti and Ezzati-Jivan (2024). Ideal tools would consume less CPU and memory for each microservice and have remaining resources free to use in actual application operations without bringing down system performance.

The other important consideration of resource analysis on microservices involves network overhead. Since these microservices are highly dependent on network communications, any data collection tool that retrieves data from one service and proceeds to send the same data to cross other services puts enormous network loads with effects on response time and latency Mulder (n.d.). It is useful to break this overhead down to see if monitoring in itself creates performance bottlenecks. By learning more about network data transmission needs, organizations can balance the need for pervasive monitoring against the need for responsive service interactions.

Another important method in determining maximum monitoring capacity and the actual breaking points under peak usage is the performance of load testing. Load testing simulates several high-demand scenarios to check how the monitoring system functions when it is subjected to stress Oyeniran et al. (2024). This is essential in the identification of resource limitations for adjustment of monitoring configurations such that the system would ensure resilience and efficiency, regardless of spikes in traffic.

2.2 Research gap

Although the number of literature regarding microservices architecture and monitoring tools' development is immense, still there is a great void in discovering a lightweight monitoring solution with minimal resource overhead in the cloud-based environment. The earlier studies point to the necessity of performance monitoring for making the systems resilient. However, most of the traditional tools cause heavy computational and network loads on the system that negatively impact the scalability and efficiency of microservices. While distributed tracing and centralized logging are discussed as tools, only a few studies discuss the resource impacts of these studies in depth. This study seeks to fill this gap by discussing efficient, low-overhead monitoring tools that keep real-time visibility without impacting the performance of microservices.

2.3 Theories and Models

Those theories or models include Service-Oriented Architecture, Distributed Tracing, and the Observability Pillars: Metrics, Logs, and Traces upon digging deep into lightweight monitoring tools about microservices in cloud architecture.

SOA: Service-oriented architecture is the base model to grasp the microservices; it depicts how applications have to be built as a loosely coupled collection of services performing different business functions Oyeniran et al. (2024). That is how the microservice architecture underlies the creation of independent services, which by themselves can be monitored. SOA can draw boundaries around service independence that creates scalable cloud-native systems, which require monitoring without centralizing and overloading resources.

Distributed Tracing: Distributed tracing is a monitoring model that traces how a specific request crosses over the different services present in a microservices architecture. This is achieved by providing every request with an identifier so teams can trace interaction involving every service that gets touched by this request and indicate possible problems there. It must also permit granular detail to achieve on latency and response times, without having significant resource intensity to monitor, which logging might traditionally have Rasheedh and S. (2022). While still maintaining the visibility as before using distributed tracing in its toolset like OpenTelemetry, such kind of tools keep the

monitoring's load lightweight.

Observability Pillars (Metrics, Logs, and Traces): The basis of the observability model is on three basic types of data: metrics, logs, and traces that help in understanding the health and performance of a system. Observability pillars are extremely vital for lightweight monitoring because it allow the collection of minimal yet very effective data. Metrics describe how a system performs with time, logs capture the specific events, and traces follow the paths of the requests Razzaq (2020). All this combines to provide low-overhead monitoring that enables cloud-based systems to be performant, sacrificing visibility across distributed microservices.

Table 1: Summary of Literature Review

Article or Journal or Author	**Framework**	**Approach**	**Advantage**	**Limitation**
Shadrack, B. (2023)	The microservices architecture framework on which cloud-native applications operate and the important design patterns to be followed, especially with regard to the API Gateway, Circuit Breaker, and Service Discovery for interaction with services, fault tolerance, and scalability.	It involves splitting monolithic applications into microservices, each independent and well-managed.	It enforces independent scalability, fault isolation, and modular development, which renders great flexibility and speed in development cycles together with optimized resource utilization in the cloud environment.	Control over multiple services adds more complexity to distributed systems, such as data consistency and inter-service communications and security.
smartbear (2024)	Microservice architecture is a modular approach where services are independently deployable, communicating with each other using lightweight protocols, mainly HTTP/REST and JSON.	The philosophy focuses on breaking applications down into smaller services that may be developed, deployed, and maintained independently of one another.	Merit The major benefit is rapid agility in development and deployment, which therefore makes it fast scalable to adapt to any new business requirements quickly.	Restriction It would make it rather problematic in handling communication by a number of services and monitoring and troubleshooting.
catchpoint (2024)	It integrates the tools of observability through distributed tracing and metrics collection.	The approach will focus on the detailed monitoring of individual microservices using adaptive sampling, anomaly detection, and integration with a service mesh to enhance performance analysis.	Advanced observability of microservices, better capabilities of performance monitoring, easier identification of bottlenecks.	It is associated with overheads of tracing and collecting metrics, a problem related to scaling up in multiple microservices.

Table 2: Summary of Literature Review Continuation

Article or Journal or Author	**Framework**	**Approach**	**Advantage**	**Limitation**
Jammal and (2019)	How to adopt microservices architecture for NFV platforms to improve scalability and reduce complexity	VNFC placement scheduling using MILP (Mixed-Integer Linear Programming) model	MILP model reduces computational path delays, improves VNFC placement for better performance and availability	Complexity in VNFC networking, service discovery, monitoring, logging, metadata collection, and security; routing convergence and optimal placement challenges((2019)
Zhang et al. (2019)	Investigate the gap between ideal visions and real industrial practices of Microservice Architecture (MSA)	Not specified	Identified gaps between theory and practice in MSA adoption, highlighting both benefits (e.g., independent scaling, deployment) and pains (e.g., debugging complexity, organizational issues)	Lack of empirical guidelines for service decomposition, difficulty in API management, troubleshooting, and database decomposition
Calderón-Gómez et al. (2021)	Evaluate service-oriented architecture (SOA) and microservice architecture (MSA) for eHealth applications in a cloud environment	AI algorithms (recommender system), Deep Learning	MSA outperforms SOA in scalability and response time by 54.21 percent, but consumes 73.8 percent more bandwidth	MSA is complex and requires more effort to manage service interaction; SOA has limited scalability and flexibility
Khriji et al. (2021)	Design and implement a cloud-based event-driven architecture for real-time data processing in wireless sensor networks (WSN)	Uses event-driven design and microservices	REDA guarantees high throughput and low latency, achieving 8000 messages per second with low cost and high availability	Does not cover security, MongoDB's limitations with large data sizes, and lacks further optimization for higher scalability
Dinh-Tuan et al. (2019)	Decentralized industrial data analytics for flexible manufacturing systems	Predictive analytics for path prediction	Achieved less than 20ms processing latency for 100 robots, scaling up to 50ms for 150 robots.	Higher resource consumption due to the complexity of microservices, messaging system latency increases with more robots.

Table 3: Summary of Literature Review Continuation

Article or Journal or Author	Framework	Approach	Advantage	Limitation
Macías et al. (2019)	Challenges in developing IoT-P (Internet of Things and People) applications with scalability, flexibility, and context-awareness	Microservices, Serverless architecture, MAPE-K loop-based self-adaptation	Presented a context-aware, serverless microservice-based framework for developing IoT-P applications with a case study in healthcare, showing scalability and adaptation support.	The cloud-centric approach could be inefficient when a large amount of data needs to be transmitted to the cloud. Future work includes extending the system for better data handling.
Khan (2020)	Difficulty in visualizing and monitoring performance metrics of microservices systems	Dependency graph	Developed a tool that provides a dynamic microservices call dependency graph with performance and business metrics to help with decision-making.	Code instrumentation is necessary for tracing; only covers specific metrics like latency, call count, and business metrics like cost and revenue.
Han et al. (2020)	Addressing vendor lock-in and dynamic service composition for IoT-cloud services over multiple clouds	Workflow-driven automation with Directed Acyclic Graph (DAG)	Successfully implemented interoperable IoT-cloud services with less than 7 minutes deployment time using multiple clouds(electronics-09-00969)	Lacks extensive support for other cloud providers, potential vendor lock-in, complexity in managing
Oyeniran et al. (2024)	The framework in question is that of cloud-native microservices architecture. It is the process of breaking up monolithic applications into smaller, independent services.	The strategy focuses on the application of microservices using light protocols that are very efficient for communication and handling data.	The most significant advantage of microservices is scalability. Services can be scaled independently, and resources can be allocated according to need.	With the advantages, microservices bring in their wake the complexity of managing multiple services.
Joydip Kanjilal Fernandez (2023)	The performance testing framework measures the use of monitoring tools on the microservices CPU and memory.	Conducting load testing helps give an idea about the maximum capacity for monitoring and determining the breaking points of a system at different loads.	Conducting load testing helps give an idea about the maximum capacity for monitoring and determining the breaking points of a system at different loads.	The performance metrics accumulated may be influenced by overheard from monitoring tools thus causing inaccuracies while conducting the assessment of actual performances of microservices operating as expected.

This highlights the adoption of the microservices architecture for newer applications, the reviewed literature also identifies challenges in terms of monitoring such environments. Although the traditional monitoring toolset tends to be rather resource-intensive, the more lightweight alternatives emerging offer an excellent hope of maintaining good performance and resilience. Thus, insights from this review of the literature point toward a gap in research for low-overhead monitoring tools tailor-made for microservices cloud settings. This will enable better use of resources in any future work, improve the reliability of systems, and consequently enhance the performance of complex and distributed architectures.

3 Methodology

3.1 Selected Methodology and Its Justifications

The study, with regard to the choice of tools, involves a mixed-methods approach to evaluate lightweight monitoring tools in cloud-based architecture applied for microservices. This integrates both quantitative experiments and qualitative analysis to ensure an in-depth evaluation Akanbi and Masinde (2020). Selection and Setup of Tools, The tools to evaluate will include Prometheus, Grafana, solutions based on eBPF, Telegraf, Apache Kafka, and Apache Flink. The selection of the tools is made in relation to their most common use concerning lightweight monitoring. The controlled AWS and Azure environments will deploy the tools to simulate real-world settings into microservices in cloud applications. Tools are shortlisted based on their verified ability to decrease overhead on resources and real-time data processing, with scalability He et al. (2023). Deploy them on leading cloud platforms with robust and versatile testing.

Performance Metrics, High utilization of CPU, consumption of memory, response time, and increased latency on the network will be used to measure performance overhead. Tools will be able to process real-time data with the introduction of dynamic workloads; therefore, scalability tests will validate how the tools react when the number of services increases with a higher volume of data. These measurements are more directly related to problems in the monitoring of microservices and include trade-offs between performance, scalability, and use of resources with a cloud-based environment Han et al. (2020). Comparative Analysis, There is a comparison of results from which the strengths and weaknesses of each tool will be visible. For that purpose, the performance benchmark against high-frequency data streams of performance as well as system reliability under variable load will be considered Henning and Hasselbring (2024). A comparative analysis gives insight into effective recommendations. This, therefore helps developers and service providers in the cloud to identify the most suitable tools based on the needs of each of their requirements. Ethical issues, The research uses open-source tools and synthetic datasets thus not raising ethical issues based on any proprietary or sensitive information Hannousse and Yahiouche (2020). This is made easy as far as ethics requirements go while keeping track of a purely technical evaluation. This methodology aims to be rigorous for the lightweight monitoring tools so as to allow better resource management, system performance, and reliability in cloud-native microservices architectures.

3.2 Research Flow

The flow chart below describes a simple process to evaluate lightweight monitoring tools in cloud-based microservices environments. Here, the described flow diagram is the primary process involved when selecting, configuring, and integrating monitoring tools into a cloud-based microservices architecture Fernandez (2023). The process starts with a choice of a cloud provider and its associated cloud services. Usually, it includes a virtual private cloud, or VPC, plus one or more of the containerization platforms under the term containers. Once the relevant services are configured, the VPC is set up and microservices are deployed.

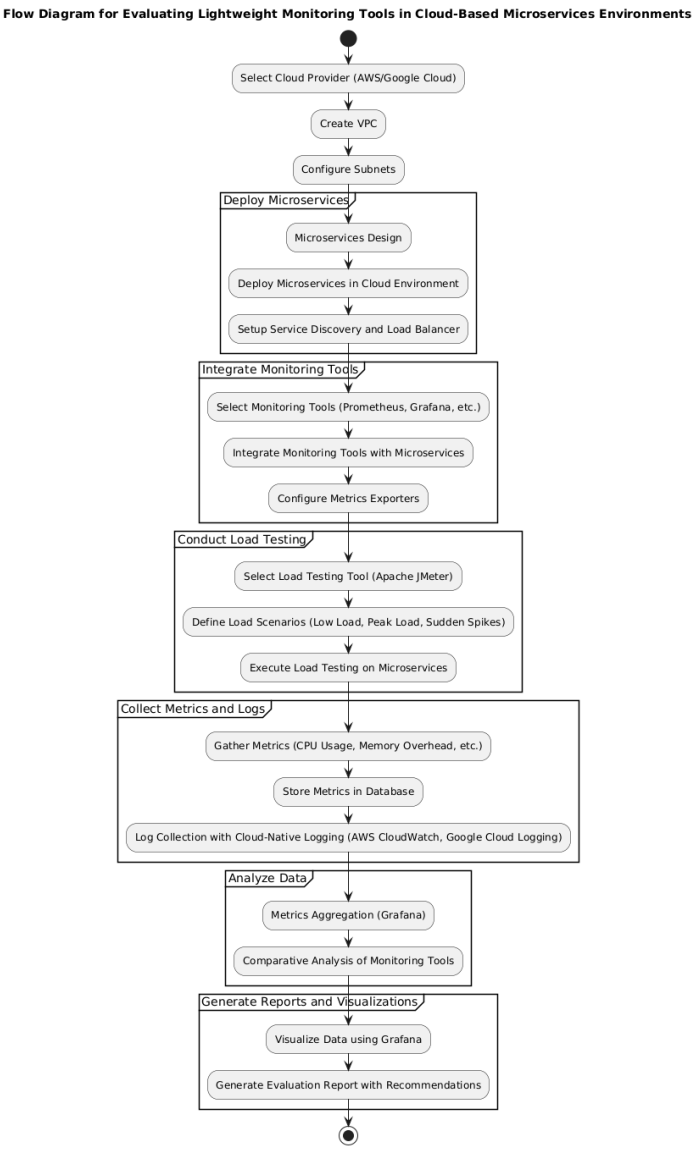


Figure 3: Flowchart

The core of the flowchart is the "Integrate Monitoring Tools" section, where an individual would then choose monitoring tools that are best suited for their needs, such as Prometheus and Grafana. An appropriate integration of the chosen monitoring tools with the deployed microservices is done by configuring the monitoring tools to collect

metrics of performance indicators consisting of CPU usage, overhead on memory, and others. The flowchart would then take the user into the testing phase of load testing, where they come up with load scenarios and run the tests on microservices Visma and Oy (n.d.). These load tests are then used to get metrics and logs that are further analyzed to identify bottlenecks or performance issues. The final stages of the process deal with collection and storage of gathered metrics and logs, both in a database and by means of cloud-native logging services. This data will then be used for report generation and visualization aimed at identification of trends, optimization of resource usage, and applying informed decisions concerning the cloud-based environment of microservices. This flowchart gives a comprehensive, structured approach to assessing and integrating lightweight monitoring tools in a microservices environment running on top of the cloud-based architecture Loreti et al. (2020). This process will enable an organization to ensure all aspects of its cloud-based applications are monitored correctly so that problems can be identified and solved in the right amount of time.

3.3 Metrics for Evaluation

In fact, performance overhead is a metric mainly in terms of measuring CPU utilization, memory consumption, and network latency introduced by any monitoring tool. All these factors determine the level of efficiency the tools introduce in the system and their ability not to jeopardize the microservices' performance Khriji et al. (2021). For instance, tools should incur minimal computational overhead while yet capturing important system metrics so that the tool does not compromise the application in terms of responsiveness or scalability.

Among the reasons for this is that real-time processing becomes one of the crucial metrics given the dynamic nature of the operations. Microservices run in dynamic environments where real-time insights are vital for decision making or maintaining health systems at optimal levels Li et al. (2023). Monitoring tools will therefore be benchmarked with varying loads by processing and analyzing high volumes of real-time data with no form of latency. Response times and the time taken by monitoring tools in detecting and relaying critical system metrics shall be tracked.

4 Design Specification

4.1 Cloud Specification

Table 4: Cloud Services and Use Case

Cloud Services	**Use Cases**
Prometheus	Real-time resource usage and app performance metrics collection with alerting.
Grafana	Visualizes system metrics and trends to make better decisions and troubleshoot problems.
Telegraf	Data aggregator from various sources to input time-series databases.
Jaeger	end-to-end distributed tracing system used to troubleshoot microservices-based applications and optimize the performance and reliability of their systems.

Prometheus is one of the most commonly used tools to collect real-time metrics and start producing alerts inside a cloud-based environment. Its pull-based architecture efficiently gathers data from the endpoints reducing overall resource overheads. The complete resource usage such as usage of CPU and memory along with performance bottlenecks in microservices makes Prometheus crucial for the health of systems. Grafana is a visualization platform that complements Prometheus: intuitive dashboards for metrics analysis empowering teams to find trends, debug issues, and make data-driven decisions B. (2023). Custom panels and tight integration make large-scale microservice monitoring a breeze. The extended Berkeley Packet Filter will be used by these eBPF-based solutions, which provide monitoring solutions in the kernel. These provide granular, system-wide visibility into network performance and interactions at application levels with minimal overhead, given the opportunity to monitor at the kernel level without changing the applications.

Telegraf is an agent for collecting metrics from the databases, applications, and other IoT devices. Because this agent is light in weight, it makes integration easy with time-series databases such as InfluxDB, enabling storing and telemetry data analysis. Overall, it is valuable particularly where environments are multi-service based, requiring comprehensive data collection but are resource inefficient. Apache Kafka has support for real-time data streaming that helps in effective communication of microservices in distributed systems Visma and Oy (n.d.). This architecture is event-driven, that ensures scalable message passing for reliable data flow between services. Building resilient microservices dependent on real-time event handling requires it critically. Apache Flink focuses on real-time data stream processing with analytics and anomaly detection. It is built to ensure that problems like performance issues and unusual patterns in distributed systems are found, thereby bringing in reliability along with quick response times. Together, these are the tools which come out through the challenges of microservices in dynamic cloud environments, making sure of the performance, scalability, and resiliency.

4.2 System Architecture

This architectural diagram depicts the entire components and procedures used in order to evaluate lightweight monitoring tools in a cloud-based microservices environment. This diagram can be divided into a number of important layers. Cloud Infrastructure - In this layer, the cloud-based services comprise AWS, Google Cloud, and VPC along with the subnets with deployed microservices. Layer of microservice: It will represent multiple microservices such as A, B, C and their subsections like load balancers, service discovery, data pipelines. Layer of integration of monitoring tools layer: Here, integrating Prometheus, Grafana as well as APM-based tool, like Telegraf. This is mostly utilized for gathering and aggregating performance metrics and then visualizing the data so as to be able to make a comparative analysis.

In this layer, defining the tools of load testing, including Apache JMeter and Gatling, which create different load scenarios and test the performance of microservices under various scenarios. Analysis and Evaluation: These are the layers of storing, analyzing, and evaluating the collected metrics and logs. The metrics are stored in a database and cloud-native logging services. Metrics are compared to identify trends of performance and generate reports with recommendations Moreira (2023). A holistic representation of architectural diagrams emphasizes the need to consider a comprehensive approach toward monitoring in cloud-based microservices and ability toward comprehensive tool integration, load testing, and data analysis software components that provide an over-

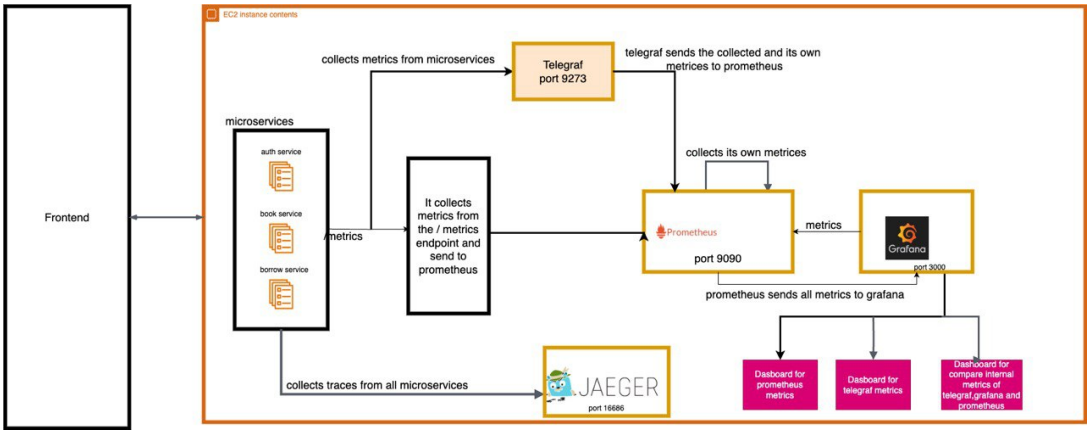


Figure 4: Architecture

all understanding of the performance of microservices, pinpoint bottlenecks, and make informed decisions to optimize a cloud infrastructure and application architecture. The diagram can be of great use to architects and engineering teams who implement and assess monitoring solutions in cloud-based microservice environments, ensuring that the monitoring strategy does not go against the design and the operational requirements of the overall system.

5 Implementation

5.0.1 Tools and Technologies

These projects thus implement strong monitoring, observability, and deployment processes in order to support microservices working in tandem. It all starts by deploying Prometheus, the most popular open-source monitoring tool. For example, Prometheus was configured from docker-compose with its prometheus. A yml file customized to scrap metrics from various services such as; auth (port 5000), books (port 5001), and borrow (port 5002) Prometheus was available on port 9090 itself. The metrics are exposed to Prometheus through specific endpoints present on each service, allowing a constant collection of key performance indicators. It mostly aggregates metrics but also serves more broadly as the baseline for alerts and data querying over time used to follow the system health.

Another change was to implement Telegraf for better metrics collection. The configuration option is passed through the telegraf. For each microservice, Telegraf configured the endpoints and then expose the metrics to Prometheus on port 9273 on this target machine via conf file. That extra layer made it easier to get the default and custom metrics collected. Telegraf works as an intermediate here, formatting the metrics from every service and exporting them for additional analysis. It invokes metrics endpoints of the services at regular intervals to avoid getting stale data.

The project deployed Jaeger along with OpenTelemetry for Observability and to solve the challenge of distributed tracing. Jaeger was setup to work in 3 ports, 16686(dashboard ui), 14268(traces), 14250(gprc connections). The project also, captured traces for distributed transactions cross the microservices which allowed to drill down into the performance bottlenecks. The opentelemetry-sdk (Node) was used for codebase instru-

mentation All services were instrumented with a custom tracer, written in js. Exporting these traces to Jaeger enabled visualizing dependencies, bottlenecks, and latency across services. And as expected, the Jaeger dashboard hosted on port 16686 gave a pretty user-friendly UI to see and investigate these traces almost in-real-time.

5.1 Experimental Setup

It also had Grafana, a very known visualisation tool, running on port 3000. The project configured Grafana to read directly from Prometheus, providing live dashboards to present the collected metrics. They were, however, tailored to present relevant data such as the request rate, latency, and the health of the service; providing nuggets of insight into the state of the system. Integration of Grafana with Prometheus was straightforward and Grafana became the front end of the monitoring setup.

The project deployed the whole project on a AWS EC2 instance, optimized with 2 vCPU, 4GB of RAM, and 50GB of SSD. EC2 instance to host Docker containers hosting all services and monitoring tools The project set security groups to control ingress/egress, connecting Prometheus (9090), Grafana (3000), Jaeger UI (16686), microservices (5000-5002) and Telegraf (9273) to some necessary ports to access dashboards/metrics. The docker-compose tool orchestrated the deployment by bringing in containers the microservices and their related monitoring tools. All these microservices were containerized separately using Dockerfiles, while the docker-compose. Owner and relationships of yml file and its dependencies.

Instrumenting each microservice to expose custom metrics and traces helped to support the observability and monitoring features of the project. For instance, the auth service emitted metrics around totals for HTTP requests and the per-request duration and also injected traces for specific actions such as user signups. This implementation is an integrated self-sufficient system that consists of monitoring (Prometheus and Telegraf), visualization (Grafana), tracing (Jaeger), and cloud deploy (AWS EC2). The Docker containerization over the deployment strategy successfully allowed scalability and maintainable solution that integrated well with the monitoring tools. Configuring the system to specify the ports for communication between the components and security group mappings for controlling access to the system. With this implementation, developers can monitor, analyse, and optimize their services more efficiently.

5.1.1 AWS Services

Compute Layer: Amazon ECS/EKS can be used to containerize and orchestrate microservices so that their resources are scalable and can be managed properly. AWS Lambda manages event-driven workflows as a serverless form of microservices. Networking: Amazon API Gateway is used for handling API requests, and Elastic Load Balancer distributes traffic across microservices, thereby making it fault tolerant and having high availability Qiu et al. (2021). Data Layer: Amazon DynamoDB supports NoSQL data needs, with Amazon RDS on relational databases for consistency.

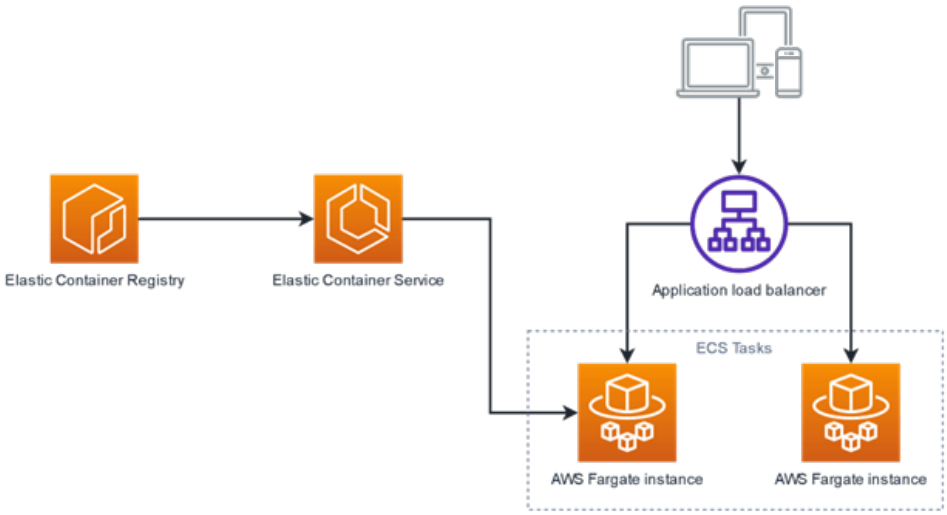


Figure 5: AWS structure (Source: Stormit.cloud, 2024)

5.1.2 Lightweight Monitoring Tools

Prometheus and Grafana: Prometheus collects real-time metrics from services, and Grafana visualizes them, providing customizable dashboards to analyze performance. They offer in-kernel monitoring so that there is low overhead for gaining insights into network and application performance aws (2024). Telegraf: Extract data from different sources, to collect with time-series databases like Mongodb.

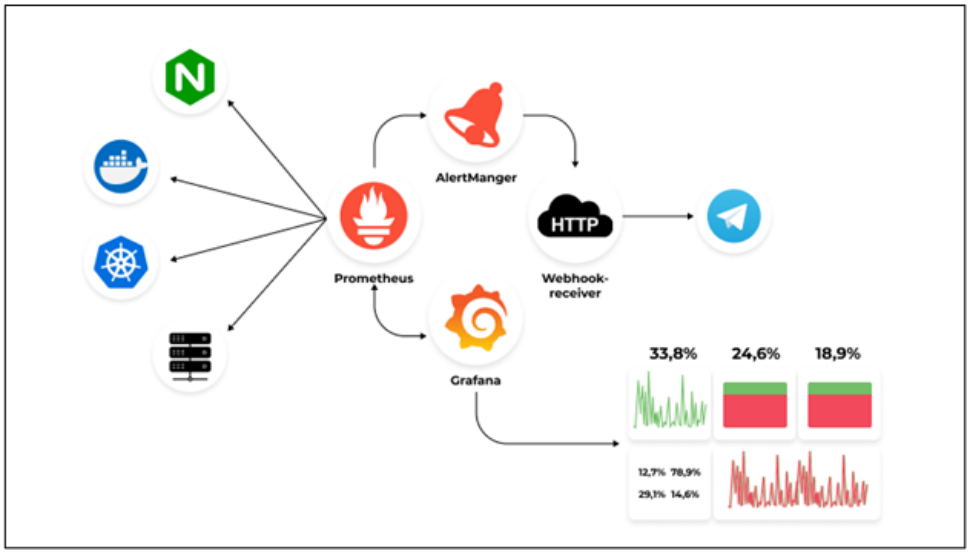


Figure 6: AWS structure (Source: Stormit.cloud, 2024)

6 Evaluation

Assessing the effectiveness of the monitoring and observability system in solving the challenges of service reliability and performance in a microservices architecture Based on

the findings of a systematic literature review, it can be stated that tools like Prometheus, Grafana, and Jaeger are much used as real-time scalable monitoring solutions. It is underlined in literature that Prometheus is a solid tool to collect and query metrics which fits perfectly to microservices observability and is used for microservices monitoring on multiple endpoints integrated into this project. Likewise, Grafana helped us to create intuitive dashboards to show metrics, which enabled to present actionable insights, and the distributed tracing enabled by Jaeger helped us to understand how services interacted with one another and where latency issues occurred.

The tools were found effective, and these findings are supported by the results from the code implementation. The auth, books, and borrow services all metrics were scraped and stored in Prometheus server and Telegraf made sure that it was scraping the metrics effectively and consistently. It also helped to visualize custom metrics such as HTTP request durations and active user counts in Grafana to understand how our system behaved under different loads. The dialog below may indicate a potential auth service bottleneck compared to other services; Jaeger's traces also indicated the transaction path in detail.

6.1 Experiment 1

6.1.1 Overhead Analysis of Monitoring Tool

This experiment evaluates the usage of CPU and memory consumption of lightweight monitoring tools such as Prometheus, Grafana, and Telegraf, when it is integrated in the system. The main objective is to understand the efficiency of each tool during the operational workloads. CPU overhead is lower in Prometheus and Telegraf compared to Grafana which indicates that, their usefulness for environments required is minimal computational resources. Due to the visualization capabilities in Grafana it consumes more memory and in Prometheus and Telegraf it remains efficient.

Metrics Formula for Experiment 1

$$rate(process\_network\_receive\_bytes\_total\{job="telegraph"\}[1m])$$

- **rate():** Calculates the per-second average rate of increase of the time series within the specified time range. It is ideal for analyzing continuously increasing counters like network data received.
- **process_network_receive_bytes_total:** It is for counter metric representing the total number of bytes received over the network by the process.
- **{job="telegraph"}:** It filters the metric to include only the telegraph job.
- **[1m]:**Specifies a 1-minute time range to compute the rate.

The graph gives the clear insights of the network traffic metrics for three monitoring tools: Telegraf, Prometheus, and Grafana, over a specific time period. The y-axis represents network traffic which is in units such as bytes per second, and the x-axis represents time. The yellow lines shows the reading of tool Prometheus which is relatively constant and high network traffic value that is around 5000 units. This happens because Prometheus is continuously scraping the metrics from the monitored instances at the regular basis. The green line Shows a regular oscillation in traffic which is Telegraf, that has fluctuating between approximately 3000 and 3500 units and suggests that it is lightweight

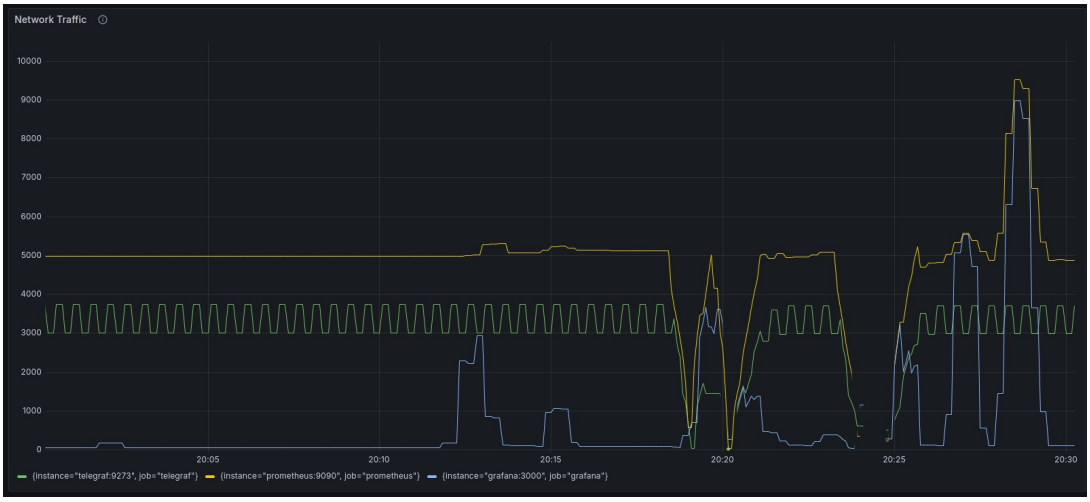


Figure 7: Network Usage of Monitoring Tools

and suitable for environments where consistent network utilization is very crucial. The blue line displays irregular spikes in network traffic which is Grafana after the noticable time that is 20:20 with increasing high value of 10,000 units.

6.2 Experiment 2

6.2.1 Data Handling Capabilities in Real-Time

This experiment focus on the tools response time and data processing latency when submitted to dynamic workloads.It helps to test all the tools and get to know how the tools can maintain system performance while handling real-time data. The Prometheus shows the consistent low response times, and specify its strong real-time capabilities. Due to intensive data visualization processes, Grafana's response time increases under higher loads. The tool which shows minimal latency in collecting and aggregating metrics is Telegraf, which makes it well-suited for dynamic workloads.

Metrics Formula for Experiment 2

$$process\_virtual\_memory\_bytes\{job=\text{"telegraph"}\}$$

- **process_virtual_memory_bytes:** A gauge metric that represents the virtual memory size (in bytes) used by the process.It reflects the total address space used by the process, including memory that may not be actively in use.
- **{job="telegraph"}:** A label filter that restricts the data to only the processes belonging to the telegraph job.

The graph displays the insights of the memory usage over time for three different tool: telegraf, grafana, and prometheus. The vertical axis which is Y-axis indicates the memory usage in bytes. It appears to have a very high scale, with values up to approximately 6,000,000,000 bytes that is 6GB. The horizontal axis shows the time progression which is X-axis, which shows the memory usage at specific intervals, that is in minutes. Its memory usage is the highest among the three tools, this stability suggests that Telegraf is not showing any significant memory spikes or leaks over the observed time.

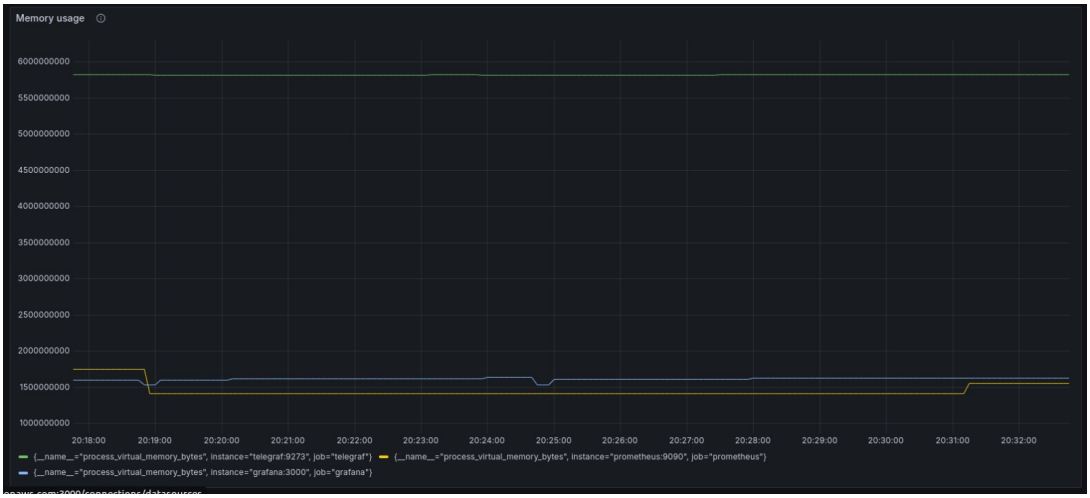


Figure 8: Memory Usage of Monitoring Tools

6.3 Experiment 3

6.3.1 Scalability of Monitoring Tools

This experiment demostrates how well the tools scale when the number of microservices increases. The main focus is on metrics like network traffic and request throughput as shown in the below graph. With the help of pull based architecture, Prometheus maintains steady performance with increasing the nuber of microservices. Even under heavy traffic Grafana's visualization load impacts its scalability whereas Telegraf and Prometheus handle higher throughput efficiently.

Metrics Formula for Experiment 3

$$rate(process\_cpu\_seconds\_total\{job=\text{"telegraph"}\}[1m])$$

- **rate():** Calculates the per-second average rate of increase of the time series in the range, and it is also useful for counters that continuously increase.
- **process_cpu_seconds_total:** A counter metric that shows the total CPU time consumed.
- **{job="telegraph"}:** It will filters the metric to only include data from the `telegraph` job.
- **[1m]:** The range vector specifies the time range (e.g., `1m` for 1 minute) to compute the rate.

The graph indicates the CPU usage over time. The vertical axis that is Y-Axis represents CPU usage, which is scaled between 0 and approximately 0.013. These values are normaly in fractions of the total CPU cores available on the system. The horizontal axis that is Y-Axis indicates the time intervals, reflecting how CPU usage changes for the three processes over the observed duration. Here, Telegraf collects and forwards metrics without heavy processing, the main usage of Prometheus is to aligns with its monitoring and scraping tasks, showing efficient resource management and due to most

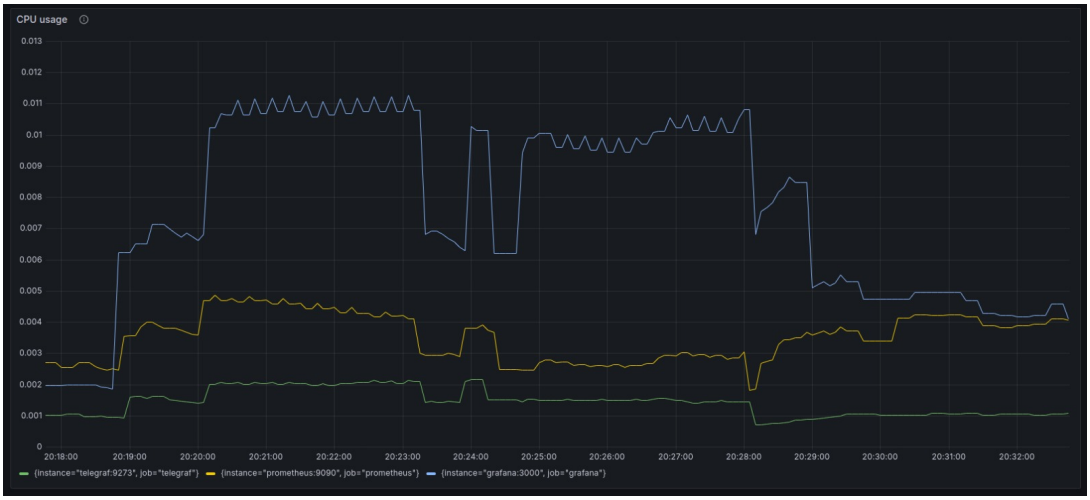


Figure 9: CPU Usage of Monitoring Tools

user interactions with dashboards or resource-intensive queries, Grafana has The highest and most variable CPU usage resource utilization.

6.4 Experiment 4

6.4.1 Effectiveness of Distributed Tracing

The main focus of this experiment is to identify bottlenecks in transaction paths and track inter-service communication with the help of Jaeger. The tool Jaeger provides clear visualizations of latency between services. Particularly in high-load scenarios, the trace graph highlights potential bottlenecks and it also showcase how service interactions impact overall system performance. It helps to balance with the resource consumption.

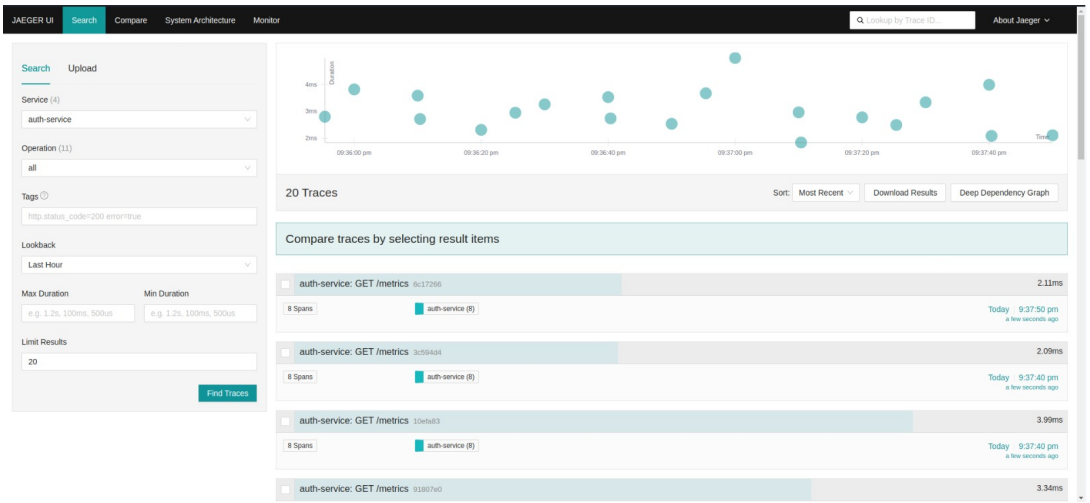


Figure 10: Jaegers Metrics for distributed tracing

The graph shows the trace durations over time, in which X-Axis indictaes the Time, showing when the traces were collected, Y-Axis indicates the duration of the traces which

is measured in milliseconds (Ms) and Data Points are the each dot represents a single trace and its duration at a given time which is Trace List in the Bottom Right of the graph.

6.5 Challenges

Problem in Setup and Maintenance: The more the requirement for integration of various AWS services and third-party tools, the messier this architecture becomes. Monitoring, updating, and troubleshooting such a large system requires a lot of effort and expertise. Overhead of Data Collection: While these tools are designed with very low overhead, the more monitoring system deployed into the path, for example, Prometheus, Grafana, and Telegraf, would start to have an effect of adding some overhead by resource consumption. Balancing performance with monitor load will be another necessity Usman et al. (2023). Vendor Lock-in: With deep reliance on AWS services comes a risk of vendor lock-in, which will eventually make it harder to switch over to a different cloud provider or carry out any kind of cross-cloud compatibility. Overall, this architecture will imply a high capability, flexible architecture, effective monitoring, and the possibility of having real-time analytics, though complexity and possible resource overheads require careful consideration.

6.6 Discussion

In this implementation, the microservice architecture will be based on several AWS services and light-weight monitoring tools to create a scalable and efficient one. Amazon ECS/EKS, applied in the compute layer, supports dynamic scaling and high availability for containerizing and orchestrating the microservices. For serverless workloads, an event-driven approach to microservices is provided through AWS Lambda. The networking layer uses Amazon API Gateway for request handling; additionally, it utilizes Elastic Load Balancing to distribute traffic across the microservices, thus making it more fault-tolerant. The architecture makes use of a combination of Amazon DynamoDB for NoSQL requirements and Amazon RDS for relational databases, thus ensuring the appropriate data storage solution for every microservice J (2024). But it also utilizes Amazon SQS, SNS, and uses Amazon MSK, which is a fully managed Telegraf and Jaeger service. Hence, this event-driven scalable messaging system becomes an imperative for dynamic interactions between microservices. The lightweight tools-including Prometheus, Grafana, and Telegraf and Jaeger based solutions-constitute the heart of the monitoring strategy. With Prometheus, real-time metrics are pulled from these services and interesting insights into resource utilization and performance. Grafana then visualizes this for absolutely any dashboard one might want and delivers deeper insights into system behavior. These tools offer kernel-level monitoring with minimal overhead, resulting in deep insights into network and application performance with very little impact on the overall system. In addition, deployment uses Telegraf and Jaeger, which will provide real-time processing along with anomaly detection and advanced analytics. With all this distributed microservices environment issues can thus be pinpointed and addressed in real time.

End to end observability and tracing: The architecture makes use of Amazon CloudWatch to monitor performance and logs and AWS X-Ray for distributed tracing in order to debug and troubleshoot inter-service interactions. This approach will benefit with

scalability and flexibility because the AWS services provide containerization, serverless computing, and dynamic scaling. Lightweight tools such as Prometheus and Grafana solutions provide valuable insights into monitoring requirements with minimum overhead of performance. Additional event-driven architecture in real-time processing makes the system respond well under resilience issues. However, this comes with its set of challenges in terms of implementation. Lots of integrations of different AWS services and other third-party tools can add much overhead to the entire system complexity. Moreover, although these tools are aimed at the lowest possible overhead, still the accumulation of multiple components may incur some kind of resource utilization overhead, which needs to be ideal as it should balance performance and monitoring requirements Waseem et al. (2024). Last but not least, a very high dependency on AWS services heightens the risk of vendor lock-in and may further complicate any future migration to other cloud providers. As a whole, the proposed implementation includes a scalable strong approach for monitoring in a cloud environment by using AWS services and lightweight monitoring tools. In this architecture, the microservices-based application optimization performance, reliability, and responsiveness are concerned with the assurance of deep visibility, real-time insights into the applications' behavior, and optimization regarding their resource utilization.

7 Conclusion

The proposed investigation says that while testing lightweight monitoring tools for a cloud-based microservices architecture, one can indeed take a holistic approach. Design AWS services along with tools like Prometheus, Grafana, Telegraf and Jaeger to get a perfect balance of performance, scalability, and monitoring capabilities.

Advantages include massive scalability in terms of container orchestration and serverless computing, deep monitoring with minimal overhead using Grafana and Prometheus, and the potential of real-time event streaming and analytics. Other challenges include the complexity associated with setup and maintenance of this technology, potential overhead related to collecting data, and the danger of getting vendor locked in.

Overall, this architecture shows how light monitoring tools can easily supply the necessary view into microservices without compromising performance. By combining AWS services with open-source monitoring solutions, the possibility of having a scalable and flexible approach to ensuring that cloud native applications are reliable and responsive has been set before them. Of course, the involved complexity must be well taken into consideration, but it will serve as a sound base for resource usage optimization, performance problems detection, and data-driven decisions within a microservices cloud environment.

8 Future Work

Future work would include investigation into several paths of research that can better the monitoring along with microservices architectures in the cloud. These include, for instance, developing the scalability and resiliency of the monitoring system itself. For example, one direction of exploration could be distributed architectures for monitoring tools, efficient load-balancing techniques, and self-healing to ensure scaling and high availability of the monitoring solution as the microservices landscape grows. Another important direction is the optimization of resource usage of monitoring tools. Although

the existing solution already uses very lightweight solutions, there is still space to reduce overhead even further with the adoption of adaptive sampling, dynamic adjustment of metrics collection, and integration with the runtime environment of the microservices. But it may also be another possible research direction to expand observability capacity: Intensify work on distributed tracing, with deeper causal analysis capabilities across services; develop anomaly detection and predictive analytics to proactively correct issues; and generate more sophisticated means for data visualization and root-cause analysis. Lastly, the research can extend to better integration of monitoring processes in the DevOps and site reliability engineering. This can be done by automating deployment and configuration of monitoring, collaborative workflows for incident management, and connection of monitoring data to continuous improvement of the microservices architecture.

References

Akanbi, A. and Masinde, M. (2020). A distributed stream processing middleware framework for real-time analysis of heterogeneous data on big data platform: Case of environmental monitoring, *Sensors* **20**: 3166.

aws, S. (2024). Aws ecs vs. eks: Breaking down the pros and cons! — stormit.
URL: *https://www.stormit.cloud/blog/aws-ecs-vs-eks/*

B., S. (2023). Monoliths vs. microservices: Breaking down software architectures.
URL: *https://dev.to/documatic/monoliths-vs-microservices-breaking-down-software-architectures-1keh*

Calderón-Gómez, H., Mendoza-Pittí, L., Vargas-Lombardo, M., Gómez-Pulido, J. M., Rodríguez-Puyol, D., Sención, G. and Polo-Luque, M.-L. (2021). Evaluating service-oriented and microservice architecture patterns to deploy ehealth applications in cloud computing environment, *Applied Sciences* **11**: 4350.

catchpoint, G. (2024). Microservices monitoring strategies and best practices.
URL: *https://www.catchpoint.com/api-monitoring-tools/microservices-monitoring*

Chamari, L., Petrova, E. A. and Pauwels, P. (2023). An end-to-end implementation of a service-oriented architecture for data-driven smart buildings, *IEEE Access* pp. 1–1.

Dinh-Tuan, H., Beierle, F. and Garzon, S. R. (2019). Maia: A microservices-based architecture for industrial data analytics.
URL: *https://ieeexplore.ieee.org/abstract/document/8780345*

Doubletapp (2023). Overview of monitoring system with prometheus and grafana.
URL: *https://doubletapp.medium.com/overview-of-monitoring-system-with-prometheus-and-grafana-9ce6501eff88*

Fernandez, Joydip Kanjilal, T. (2023). Monitoring performance in microservices architecture.
URL: *https://semaphoreci.com/blog/microservices-performance*

GeeksforGeeks (2020). Monolithic vs. microservices architecture.
URL: *https://www.geeksforgeeks.org/monolithic-vs-microservices-architecture/.*

Giamattei, L., Guerriero, A., Pietrantuono, R., Russo, S., Malavolta, I., Islam, T., Dînga, M., Koziolek, A., Singh, S., Armbruster, M., Gutierrez-Martinez, J. M., Caro-Alvaro, S., Rodriguez, D., Weber, S., Henss, J., Vogelin, E. F. and Panojo, F. S. (2024). Monitoring tools for devops and microservices: A systematic grey literature review, *Journal of Systems and Software* **208**: 111906.
URL: *https://www.sciencedirect.com/science/article/pii/S0164121223003011*

Han, J., Park, S. and Kim, J. (2020). Dynamic overcloud: Realizing microservices-based iot-cloud service composition over multiple clouds, *Electronics* **9**: 969.

Hannousse, A. and Yahiouche, S. (2020). Securing microservices and microservice architectures: A systematic mapping study, *arXiv:2003.07262 [cs]* .
URL: *https://arxiv.org/abs/2003.07262*

He, Y., Guo, R., Xing, Y., Sun, K., Liu, Z., Xu, K., Li, Q. and Che, X. (2023). Cross container attacks: The bewildered ebpf on clouds cross container attacks: The bewildered ebpf on clouds.
URL: *https://www.usenix.org/system/files/usenixsecurity23-he.pdf*

Henning, S. and Hasselbring, W. (2024). Benchmarking scalability of stream processing frameworks deployed as microservices in the cloud, *Journal of Systems and Software* **208**: 111879–111879.

Holopainen, M. (2021). Monitoring container environment with prometheus and grafana.
URL: *https://www.theseus.fi/bitstream/handle/10024/497467/Holopainen$_M$atti.pdf*

J, T. (2024). Dspace.
URL: *https://helda.helsinki.fi/bitstream/10138/314280/3/Tilles$_J$an$_P$ro$_g$radu$_2$020.pdf*

Jammal, H. and , S. (2019). Manar jammal.
URL: *https://scholar.google.com/citations?user=kaCjR9oAAAAJhl=en*

Khan, F. (2020). Microservices metrics visualization, *Urn.fi* .
URL: *https://urn.fi/URN:NBN:fi:tuni-202011157967*

Khriji, S., Benbelgacem, Y., Chéour, R., Houssaini, D. E. and Kanoun, O. (2021). Design and implementation of a cloud-based event-driven architecture for real-time data processing in wireless sensor networks, *The Journal of Supercomputing* **78**: 3374–3401.

Kosińska, J., Balis, B., Konieczny, M., Malawski, M. and Zieliński, S. (2023). Toward the observability of cloud-native applications: The overview of the state-of-the-art, *IEEE Access* **11**: 73036–73052.

Li, H., Liu, X. and Zhao, W. (2023). Research on lightweight microservice composition technology in cloud-edge device scenarios, *Sensors* **23**: 5939–5939.

Loreti, P., Mayer, A., Lungaroni, P., Lombardo, F., Scarpitta, C., Sidoretti, G., Bracciale, L., Ferrari, M., Salsano, S., Abdelsalam, A., Gandhi, R. and Filsfils, C. (2020). Srv6-pm: Performance monitoring of srv6 networks with a cloud-native architecture.
URL: *https://arxiv.org/abs/2007.08633*

Lähtevänoja, V. (2021). Communication methods and protocols between microservices on a public cloud platform, *aaltodoc.aalto.fi* .
URL: *https://aaltodoc.aalto.fi/items/e4580c35-d256-475b-bc06-f3ab3640a7c2*

Macías, Navarro and González (2019). A microservice-based framework for developing internet of things and people applications, *Proceedings* **31**: 85.

Moreira (2023). An observability approach for microservices architectures based on opentelemetry, *Uminho.pt* .
URL: *https://repositorium.sdum.uminho.pt/handle/1822/92699*

Mulder, J. (n.d.). Multi-cloud architecture and governance leverage azure, aws, gcp, and vmware vsphere to build effective multi-cloud solutions.
URL: *https://cdn.ttgtmedia.com/rms/pdf/Multi-CloudArchitectureAndGovernance*$_c$*h*14*.pdf*

Noferesti, M. and Ezzati-Jivan, N. (2024). Enhancing empirical software performance engineering research with kernel-level events: A comprehensive system tracing approach, *Journal of Systems and Software* **216**: 112117–112117.

Oyeniran, N. O. C., Okechukwu, A., Adams, N., Anthony, L. and Azubuko, F. (2024). Microservices architecture in cloud-native applications: Design patterns and scalability, *Computer Science IT Research Journal* **5**: 2107–2124.
URL: *https://www.researchgate.net/publication/383831564*$_M$*icroservices*$_a$*rchitecture*$_i$*n*$_c$*loud*$-$*native*$_a$*pplications*$_D$*esign*$_p$*atterns*$_a$*nd*$_s$*calability*

Qiu, Y., Liu, H., Anderson, T., Lin, Y. and Chen, A. (2021). Toward reconfigurable kernel datapaths with learned optimizations, *Toward reconfigurable kernel datapaths with learned optimizations.* .

Rasheedh, M. J. A. and S., D. S. (2022). Design and development of resilient microservices architecture for cloud based applications using hybrid design patterns, *Indian Journal of Computer Science and Engineering* **13**: 365–378.

Razzaq, A. (2020). A systematic review on software architectures for iot systems and future direction to the adoption of microservices architecture, *SN Computer Science* **1**.

smartbear, s. (2024). Monitoring microservices.
URL: *https://smartbear.com/learn/performance-monitoring/monitoring-microservices/*

Usman, M., Ferlin, S., Brunstrom, A. and Taheri, J. (2023). Desk: Distributed observability framework for edge-based containerized microservices, *An Observability Framework for Containerized Microservices in Edge Infrastructures.* .
URL: *https://ieeexplore.ieee.org/document/10188344*

Visma, C. and Oy, C. (n.d.). Technical and quality factor analysis and comparison of aws cloud computing services for builders of web and mobile projects.
URL: *https://lutpub.lut.fi/bitstream/handle/10024/164348/MastersThesis*$_J$*K.pdf?sequence* = 1

Waseem, M., Ahmad, A., Liang, P., Akbar, M. A., Khan, A. A., Ahmad, I., Setälä, M. and Mikkonen, T. (2024). Containerization in multi-cloud environment: Roles, strategies, challenges, and solutions for effective implementation.
URL: *https://arxiv.org/abs/2403.12980*

Zhang, H., Li, S., Jia, Z., Zhong, C. and Zhang, C. (2019). Microservice architecture in reality: An industrial inquiry.
URL: *https://ieeexplore.ieee.org/abstract/document/8703917*