STREAMLINING SOFTWARE RELEASE PROCESS AND RESOURCE MANAGEMENT FOR MICROSERVICE-BASED ARCHITECTURE ON MULTI-CLOUD

Samuditha Jayawardena

IT20074968

B.Sc. (Hons) Degree in Information Technology
Specializing in Software Engineering

Department of Software Engineering

Sri Lanka Institute of Information Technology
Sri Lanka

September 2023

STREAMLINING SOFTWARE RELEASE PROCESS AND RESOURCE MANAGEMENT FOR MICROSERVICE-BASED ARCHITECTURE ON MULTI-CLOUD

Samuditha Jayawardena

IT20074968

B.Sc. (Hons) Degree in Information Technology
Specializing in Software Engineering

Department of Software Engineering

Sri Lanka Institute of Information Technology
Sri Lanka

September 2023

Declaration

"I declare that this is my own work and this dissertation does not incorporate without acknowledgement any material previously submitted for a Degree or Diploma in any other University or institute of higher learning and to the best of my knowledge and belief it does not contain any material previously published or written by another person except where the acknowledgement is made in the text.

Also, I hereby grant to Sri Lanka Institute of Information Technology, the non-exclusive right to reproduce and distribute my dissertation, in whole or in part in print, electronic or other medium. I retain the right to use this content in whole or part in future works (such as articles or books).

Signature: Date: 11/09/2023

The above candidate has carried out research for the bachelor's degree Dissertation under my supervision.

Signature of the supervisor: Date:

Abstract

In today's world, most companies opt for open-source cloud service monitoring and visualization tools due to cost considerations for commercial requirements. However, despite the popularity of these tools, they have several limitations. To monitor and visualize different data, cloud engineers must switch between various tools since none of the current tools can efficiently monitor all the required data. Grafana, Prometheus, Jaeger, Riemann, Elasticsearch among others, are popular tools used in cloud engineering. Nevertheless, these tools have their limitations, and no single tool can efficiently monitor all the necessary data. Therefore, there is a need to develop a centralized platform that can efficiently collect and analyse data from multiple monitoring tools to enhance cloud service monitoring and visualization.

This research focuses on the development of a centralized platform for cloud service monitoring and visualization. Cloud computing has become a popular technology due to its ability to reduce resource management complexities. However, monitoring and analysing cloud data can be challenging due to the vast amounts of data involved. The proposed platform aims to streamline the monitoring process and simplify data analysis by providing a centralized location for data collection and analysis from various monitoring tools. The platform will address the limitations of current cloud data monitoring and visualization tools and provide a more efficient and effective solution to meet the requirements and desires of clients. The ultimate goal of this research is to ensure the Quality of Experience (QoE) and Quality of Services (QoS) of cloud computing services for end-users.

Keywords: Grafana, Prometheus, Jaeger, Riemann, Monitoring, visualization, Elasticsearch, API, Centralize

Acknowledgment

I would like to thank my supervisor, Dr Nuwan Kodagoda and co-supervisor, Mr. Udara Samarathunga, for the guidance and motivation to make this research a success. I would also like to thank the Department of Software Engineering of the Sri Lanka Institute of Information Technology and the CDAP lecturers and staff for providing the opportunity to conduct this research.

Table of Contents

List Of Tables

List Of Figures

List Of Abbreviations

CPU - Central Processing Unit
DataDog - Real-time monitoring, alerting, and analytics tool for microservices

Istio - An open-source service mesh platform that provides traffic management, policy enforcement, and telemetry collection.

JVM - Java Virtual Machine
OSI - Open Systems Interconnection
API - Application Programming Interface DNS - Domain Name System
HTTP - Hypertext Transfer Protocol
HTTPS - Hypertext Transfer Protocol Secure IP - Internet Protocol
TCP - Transmission Control Protocol
UDP - User Datagram Protocol

ASP.NET Core: A cross-platform, high-performance framework for building cloud applications.

ER Diagram: Entity-Relationship Diagram, a visual representation of data entities and their relationships.
MVC: Model-View-Controller, a software architectural pattern used for developing web applications.
GitLab: A web-based platform for version control and collaboration.
C#: A programming language developed by Microsoft.

UI: User Interface, the graphical layout and elements of an application that users interact with.

Git: A distributed version control system for tracking changes in source code during software development.

IDE: Integrated Development Environment, a software application for coding, testing, and debugging.

SDK: Software Development Kit, a set of tools and libraries for developing software applications.

1. INTRODUCTION

Monitoring [1] is a crucial aspect of cloud computing platforms. It serves as the foundation for network analysis, system management, job scheduling, load balancing, event prediction, fault detection, and recovery operations. Monitoring enables cloud computing platforms to assess resource utilization, identify service defects, understand user patterns, and facilitate resource allocation. As a result, it plays a pivotal role in enhancing the quality of service provided by cloud computing platforms. [2] Cloud monitoring tools serve five essential functions, namely data collection, filtering, aggregation, analysis, and decision-making. To carry out these functions, multiple agents are deployed throughout different areas of the cloud infrastructure. Figure 1 illustrates the general architecture of a cloud monitoring tool. It is designed to inject agents into various parts of the cloud infrastructure to enable effective data collection, filtering, aggregation, analysis, and decision-making processes.

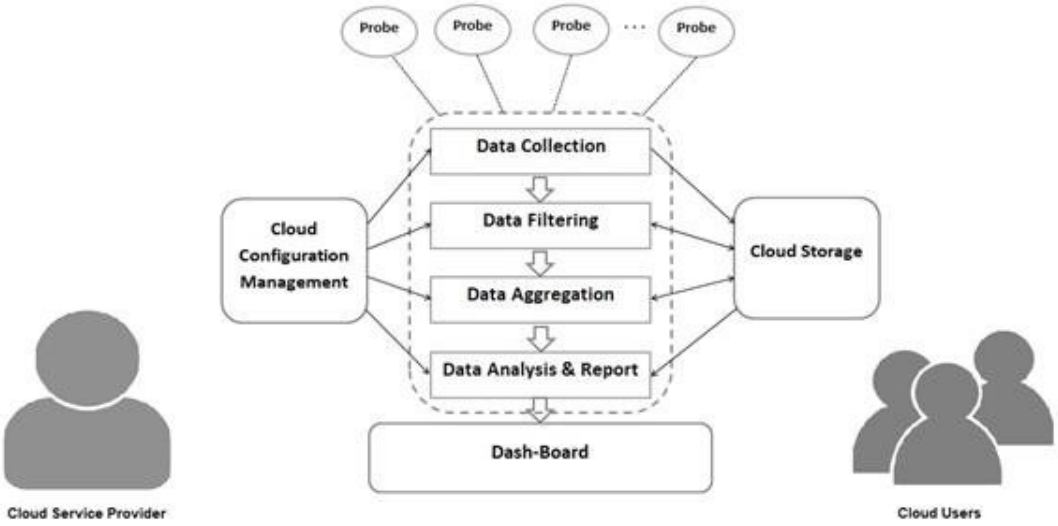

Figure 1: General architecture of cloud monitoring Tool

Source: ([3])

The field of microservices has revolutionized software development by breaking down monolithic applications into smaller, independently deployable services. However, with the increased complexity of microservice architectures, ensuring the proper functioning of these services has become a significant challenge. Monitoring the

performance of these services is critical to understanding the overall health of the application and troubleshooting any problems that arise. Effective monitoring involves collecting and analyzing data from various services to gain insight into their behavior and identify any issues that may affect the application's performance. This data can be obtained through defined metrics and logs that track the performance of each service and its interactions with other services.

In a microservice architecture, multiple monitoring tools are commonly used to collect, analyze, and visualize data. Some of the popular monitoring tools used in microservices include Jaeger, Prometheus, Grafana, and DataDog. These tools help in identifying potential issues in microservices, tracking the overall system performance, and optimizing the microservices' performance.

Jaeger is a distributed tracing system that can help identify issues within a microservice by collecting trace data. Prometheus is a popular monitoring tool that helps in monitoring the state of a microservice and providing real-time alerts. Grafana, on the other hand, provides an intuitive dashboard to visualize the data collected by Prometheus. DataDog is another tool that provides real-time monitoring, alerting, and analytics for microservices.

These monitoring tools can collect various types of data, such as the number of requests made, the response time, the error rate, the network traffic, and CPU usage. However, they may not be able to collect some data such as the business metrics, the data processing time, the user experience, and the logs generated by the microservices. Therefore, to have a complete picture of the microservices' performance, it is essential to use multiple monitoring tools together and integrate them into a centralized platform for data analysis and visualization.

1. Jaeger: Jaeger is primarily used for distributed tracing of requests across multiple microservices. It can collect data such as the duration of each operation, the dependencies between different services, the service response time, and the error rate. However, it cannot collect data such as the CPU usage, memory consumption, network traffic, or application logs.

Table 1: Jaeger limitations and monitoring data

Monitoring Data	**Limitations**
Distributed Traces	Can't system log Monitor
Service metrics: Requested processed by each service	Can't Network traffic Monitor
Error Rate	
Response Time	

2. Prometheus: [4]Prometheus is a monitoring tool that collects metrics and events data from different sources. It can collect data such as the number of requests made, response time, CPU usage, memory consumption, network traffic, and other system-level metrics. However, it cannot collect data such as the log data, business metrics, or user experience metrics.

Table 2: Prometheus limitations and monitoring data

Monitoring Data	**Limitations**
CPU Usage	Can't monitor text logs
Network Traffic	Can't monitor images
Disk Space	if you need more advanced visualization capabilities, you may want to use additional tools such as Grafana
Monitor and visualize numeric data	not designed to monitor or visualize non-numeric data.

3. Grafana: [3]Grafana is a visualization tool that helps to visualize and analyze the data collected by Prometheus. It can collect data such as the number of requests, response time, CPU usage, memory consumption, network traffic, and other metrics. However, it cannot collect data such as the log data, business metrics, or user experience metrics.

Table 3: Grafana Limitations and monitoring data

Monitoring Data	**Limitations**
Time-Series data	Can’t monitor text logs
CPU Usage	Can’t monitor images
Disk Space	Can’t monitor videos
Memory Usage	While Grafana can integrate with log management systems, it is not designed to handle large amounts of unstructured data, such as raw text logs.

4. Kiali: Kiali is a service mesh observability platform that provides real-time visibility into microservices and the Istio service mesh. Kiali collects data such as traffic volume, error rates, latency, and response times across microservices in real-time, allowing users to identify bottlenecks and troubleshoot issues quickly.

Some of the data that Kiali can collect includes:

- Traffic volume: Kiali can collect data on the amount of traffic flowing between microservices, allowing users to identify the most active services and traffic patterns.
- Error rates: Kiali can detect and collect data on error rates and status codes returned by microservices, helping users identify service-level errors and misconfigurations.
- Latency: Kiali can measure and collect data on latency across microservices, helping users identify slow or inefficient services.
- Service response times: Kiali can track and collect data on the response times of individual services, allowing users to identify bottlenecks or slow-running services.

However, Kiali cannot collect data such as:

- CPU usage and memory consumption: Kiali does not collect data on system-level metrics like CPU usage, memory consumption, or network traffic.
- Log data: Kiali cannot collect log data from microservices, which can be important for troubleshooting issues or identifying application-level errors.

The purpose of this research proposal is to outline the plan for developing a centralized platform that enables the collection and analysis of data from various monitoring tools. The proposed solution aims to address the challenge of managing and monitoring multiple systems and applications by providing a consolidated view of data from different sources.

1.1 Background & literature survey

Prometheus and Grafana are popular monitoring and visualization tools used for collecting and analyzing metrics data. These tools are often combined to create a comprehensive cloud data monitoring and visualization tool. However, this approach has its own set of problems and limitations. This literature review will explore the architecture of Prometheus and the limitations of combining Prometheus and Grafana for cloud data monitoring and visualization. Additionally, it will examine relevant research papers on this topic to provide a comprehensive review. Prometheus is an open-source monitoring system that is designed for metrics collection, monitoring, and alerting. It has a multi-dimensional data model that allows it to collect data from various sources and store it in a time-series database. The architecture of Prometheus consists of several components, including the Prometheus server, exporters, and alert manager.

The Prometheus server is responsible for collecting metrics data and storing it in a time- series database. It has a query language called PromQL, which allows users to query the collected data and perform analysis. The exporters are responsible for collecting metrics data from various sources, such as applications and operating systems, and converting it into a format that Prometheus can understand. The alert manager is responsible for processing alerts generated by Prometheus and sending them to various channels, such as email or Slack. Combining Prometheus and Grafana to create a cloud data monitoring and visualization tool has its own set of problems and limitations. One of the limitations is the need for expertise in both tools. Users must have knowledge of both Prometheus and Grafana to effectively use this combined

tool. Additionally, the combination of Prometheus and Grafana can lead to data duplication, which can lead to performance issues and storage problems. Another limitation is the lack of support for advanced visualization options. While Grafana provides a wide range of visualization options, it may not be suitable for all use cases. The lack of customization options can also limit the flexibility of the tool.

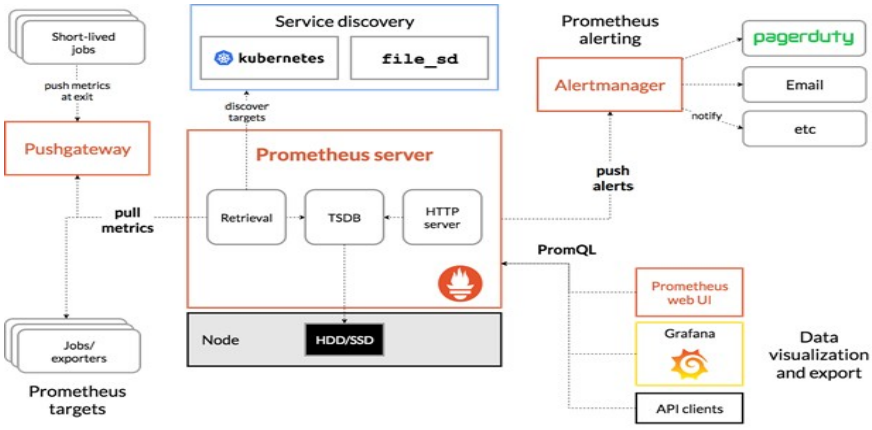

Figure 2 : Prometheus core architecture diagram

Source: ([4])

There are several research papers that have examined the use of Prometheus and Grafana for cloud data monitoring and visualization. For instance, a paper by Mehta et al. (2021) examined the use of Prometheus and Grafana to monitor and analyze Kubernetes cluster performance. The authors found that the combination of Prometheus and Grafana provided a comprehensive view of the system performance, but also highlighted the need for expertise in both tools. Another paper by Krishnan et al. (2021) examined the use of Prometheus and Grafana for monitoring and visualizing data from distributed systems. The authors found that while the combination of Prometheus and Grafana provided a powerful tool for monitoring and visualization, it required careful configuration to avoid data duplication and performance issues.

Jaeger is an open-source distributed tracing system that is commonly used for monitoring microservices. While Jaeger has several benefits, including real-time tracing and the ability to analyze complex systems, it also has several limitations. One of the limitations of Jaeger is its scalability. As the number of microservices in a

system increases, Jaeger's ability to handle the large volumes of data generated by these services becomes limited. This can lead to delays in data processing and analysis, making it difficult for users to make informed decisions about the health and performance of their systems.

Another limitation of Jaeger is its complexity. Jaeger's user interface can be difficult to navigate, especially for non-technical stakeholders. This complexity can result in a lack of clarity and understanding of the data, leading to incorrect conclusions and suboptimal decision-making. Other open-source cloud monitoring tools, such as Zabbix and Nagios, also have limitations. For example, these tools often lack real-time data analysis capabilities, making it difficult for users to identify and respond to issues in a timely manner. Additionally, these tools can be difficult to configure and maintain, requiring specialized knowledge and expertise.

Several research papers have discussed the limitations of Jaeger and other open-source cloud monitoring tools. For example:

1. Moustafa, N., Dawoud, D., & Wu, J. (2021). An Intelligent Multi-Layered Architecture for Microservices Monitoring Using Distributed Tracing. IEEE Access, 9, 135000-135015.

This paper discusses the limitations of Jaeger and other distributed tracing systems for monitoring microservices. The authors propose an intelligent multi-layered architecture for monitoring microservices that addresses some of the limitations of existing systems.

2. Wang, J., Wang, R., & Chen, S. (2021). Cloud Service Monitoring Based on Distributed Tracing. In Proceedings of the 2021 International Conference on Intelligent Computing and Sustainable System (pp. 188-193). Springer.

This paper discusses the limitations of Jaeger and other distributed tracing systems for monitoring cloud services. The authors propose a cloud service monitoring framework based on distributed tracing that addresses some of the limitations of existing systems.

3. Oussous, A., Lahcen, A. A., & Belfkih, S. (2021). Monitoring and Alerting for Cloud Services: A Comprehensive Review of the State-of-the-Art. Journal of Cloud Computing, 10(1), 1-35.

This paper provides a comprehensive review of the state-of-the-art in cloud monitoring and alerting. The authors discuss the limitations of Jaeger and other open-source cloud monitoring tools and provide recommendations for improving the effectiveness of these tools. In conclusion, while Jaeger and other open-source cloud monitoring tools have several benefits, they also have limitations that can make it difficult for users to effectively monitor and analyze their systems. Further research is needed to address these limitations and improve the effectiveness of these tools.

1.2 Research gap

Despite the availability of numerous monitoring solutions for gathering data on different system performance parameters, there isn't a centralised platform that can combine and analyse data from diverse monitoring tools. It might be difficult to see possible problems and improve system performance when several tools are used for various metrics, creating a fragmented view of system performance. Consequently, a centralised platform is required to enable effective analysis for better decision-making and to provide a consolidated picture of data from various monitoring systems. Despite the significance of this topic, little study has been done on the conception, creation, and assessment of such a platform. Therefore, additional study is required to fill this knowledge gap and provide a centralised platform that can effectively integrate and analyze data from multiple monitoring tools to optimize system performance. Most monitoring tools are currently designed to capture only one or two types of data. We can see some improvement in cloud data monitoring and visualisation in commercial monitoring tools (ex: Data Dog, SolarWinds). But that tool is very expensive. Because of that, we need a budget cloud monitoring and visualisation tool that can monitor most of the data and metrics and visualise them. We discussed with industry cloud experts, and they said they need a cloud visualisation and monitoring tool that can monitor metrics, logs, and traces using one tool without switching between different

tools. Because of that tool, they can save time and ease their work. Also, if we can develop that tool at a minimum cost, that is a advantage for future cloud monitoring. The most essential items for today's industries are the log filtering and monitoring features. Because the system developer frequently reads logs when users report bugs. However, it is a laborious and time-consuming task.However, having log monitoring and log filtering capabilities in a monitoring tool is crucial for maintaining productivity. Many cloud data monitoring technologies were tough to manage. Otherwise, most visualisation tools are challenging to configure with monitoring tools. Most monitoring tools are quite tough to configure with applications. Because of this, today's industry wants user-friendly, straightforward smart monitoring and visualisation tools.

1.3 Research problem

The ability to effectively analyze and visualize data generated by monitoring tools is crucial in making informed decisions about the performance and health of microservices. However, the current monitoring tools such as Prometheus, Grafana, and Jaeger have limitations that hinder their ability to provide a comprehensive view of the data. These tools do not allow for the integration of data from multiple sources, leading to fragmented and incomplete views of system performance. As a result, users face challenges in understanding the results and making informed decisions.

Furthermore, the current monitoring tools often lack user-friendly interfaces and intuitive dashboards, making it difficult for users to navigate the data and gain meaningful insights. The visualizations provided by these tools are often complex and require specialized knowledge, making it difficult for non- technical stakeholders to understand the data. This results in a lack of clarity and understanding of the data, leading to incorrect conclusions and suboptimal decision-making.

To address these problems, there is a need for a centralized platform that can collect data from multiple sources, analyze and visualize the data in a user- friendly manner, and provide meaningful insights for users. This platform should be designed with the

needs of both technical and non-technical stakeholders in mind, providing a simple and intuitive interface for data navigation and analysis. By developing such a platform, we can help organizations make informed decisions about the performance and health of their microservices, leading to improved efficiency, better resource allocation, and ultimately, improved business outcomes. There are no restrictions for handling and displaying big amounts of data in modern apps, which have a large volume of data. When data volume increases, users have numerous challenges when attempting to predict data.

1.4 Research objectives

1.4.1 Main Objectives

This research is aimed to invent an automated architecture and optimize the software release process system according to the architectural attributes such as performance, security, extendibility, reliability, modifiability.

1.4.2 Specific Objectives

1. To create a centralized platform that can collect and analyze data from various monitoring tools.
2. To simplify the data visualization process by integrating data from multiple tools into one platform.
3. To provide real-time insights into the performance and behavior of the monitored systems.
4. To develop a user-friendly interface for monitoring multiple systems and applications.
5. To enable users to customize the dashboard according to their preferences and requirements.
6. To provide alerts and notifications in real-time to help identify and resolve issues promptly.
7. To maintain a historical view of the performance and behavior of the monitored systems.
8. To provide the ability to perform ad-hoc analysis of the data to support troubleshooting and problem resolution.

2. METHODOLOGY

The proposed solution will gather and integrate data from Prometheus, Elasticsearch and Jaeger into a single platform. It will also design and implement an efficient data storage and retrieval mechanism and develop an intuitive user interface for navigating and analyzing data. The research team will be responsible for conducting performance evaluations to ensure the scalability and reliability of the proposed solution.

2.1 Requirement Gathering

To conduct this research, I need to gather requirements to identify the problems with the current monitoring and visualization tools. Additionally, I will review previously conducted research in this area to understand the improvements made in that research and the issues associated with existing tools. Furthermore, I will engage in discussions with industry experts to gather details about the challenges they have encountered with existing tools.

2.2 Feasibility Study

The technical feasibility of the research on cloud monitoring and microservices performance is high. The field of cloud monitoring is well-established, and there are many tools available to collect, filter, aggregate, analyze, and make decisions based on data. Similarly, the field of microservices has grown significantly in recent years, and there are several popular monitoring tools available, such as Jaeger, Prometheus, Elasticsearch that help collect and analyze data. These tools can collect metrics such as the number of requests, response time, error rates, network traffic, CPU usage, and memory consumption. However, the tools have limitations in terms of the data they can collect, such as business metrics, data processing time, user experience, and logs generated by microservices. Therefore, integrating multiple monitoring tools and using them together in a centralized platform for data analysis and visualization is essential to have a complete picture of the microservices' performance. Overall, the technical

feasibility of the research is high, and there are many well-established tools available to support the research.

2.3 Identity Existing Systems

Find additional existing systems for monitoring and displaying data from cloud monitoring during the requirements gathering and research gap identification phases. Currently, the industry uses two different types of systems. Commercial surveillance tools come first. Although those solutions offer users more features, their price is rather costly. Most businesses, as a whole, think about costs. Data Dog, the Azure Monitoring Tool, the AWS Monitoring Tool, and others are examples of commercial tools. Open-source monitoring tools are another. There are currently many open-source monitoring programmes available; however, they don't have any restrictions and are inexpensive. Prometheus, Jager, and other open-source tool examples.

Figure 3 : Existing Cloud Monitoring Tools

2.4 System Analysis

The following is an overview of the suggested software solution. The following are the major components of the approach.

- UI for data visualize.
- Backend service for integrate multiple open-source monitoring tools
- Database service

2.5 Technical Feasibility

Front-end Development with Angular: Angular is a widely-used JavaScript framework that allows for the creation of complex web applications. With its comprehensive toolset and vast library of components, Angular is ideal for developing user interfaces for cloud visualization tools. It provides a clean and organized structure for front-end development, which makes it easier to build scalable and maintainable web applications. Back-end Development with .NET Core C#: .NET Core is a cross-platform open- source framework for building cloud applications. It provides a powerful platform for developing the back-end components of cloud visualization tools. With its vast libraries and tools, .NET Core C# allows developers to build scalable, efficient, and secure web applications that can handle high volumes of data. Seamless Integration: Angular and .NET Core C# are designed to work together seamlessly, providing a powerful toolset for building cloud visualization tools. The two technologies are complementary and work in harmony to create efficient and powerful web applications.

2.5.1 System Development & Implementation

Before starting the development, the development environment setup is the most challenging part. After a technical feasibility study, I decided to use ASP.NET Core as a backend framework and Angular as a frontend framework for the implementation. First, clone the project in GitLab and create the main folder in the common resource folder in the project. As a software backend, the most important part is the.NET Core MVC architecture, which is a very famous architecture for building web applications. Otherwise, this is a cloud data monitoring tool, not an easy project. First, I need to draw an ER diagram to get an idea of the system, how to collect data, and how to pass data to the visualisation tool. As a developer, always try to use best practices during development. Write logs, component breakdowns, use proper name conventions, etc.

2.5.2 Centralize Monitoring Tool

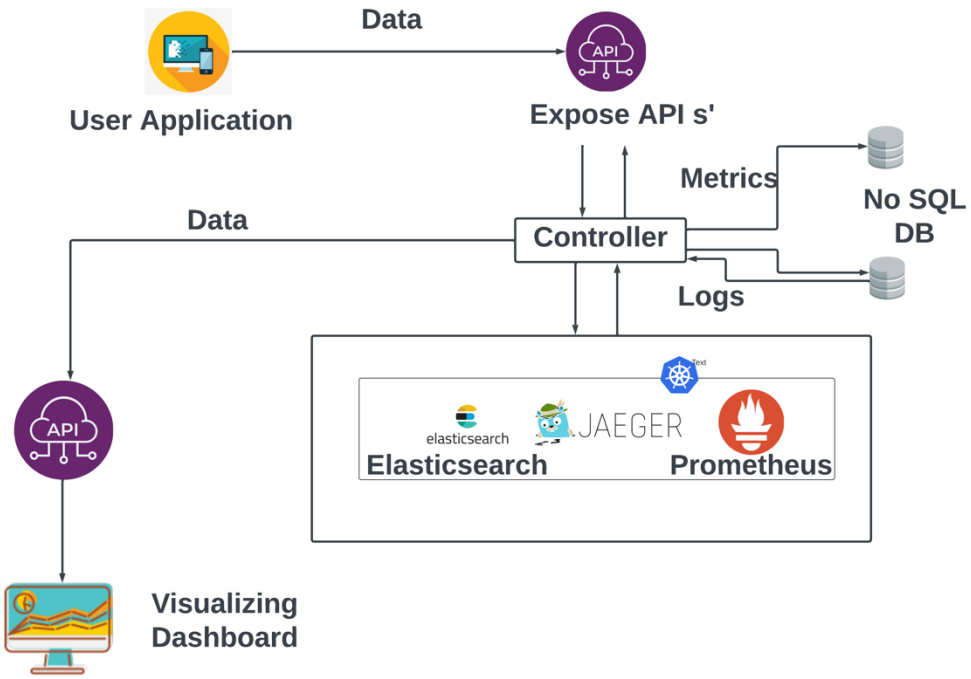

Figure 4 : Architecture of Centralize Monitoring Tool

In Figure 4, the implementation process of a centralized monitoring tool is depicted, highlighting the collection of metrics, logs, time series data, and other relevant information from user applications. The diagram also illustrates the seamless transfer of data between the visualization and monitoring tools. Primarily, the proposed system creates a centralized platform capable of collecting extensive data, thus eliminating the need to switch between multiple tools. There are three open-source monitoring tools have been utilized, with Prometheus being the first one. The utilization of Prometheus in the proposed solution is motivated by its numerous advantages, including high-quality data, queries, operations, and a wide range of libraries [13]. While Prometheus excels in monitoring and analyzing metrics, it does not support trace data. Consequently, Jaeger is incorporated into the proposed solution to address this limitation and enable efficient monitoring of trace data. ElasticSearch is capable of effectively monitoring log data. This inclusion is crucial to ensure comprehensive monitoring capabilities within the proposed solution. To establish the proposed solution, the first step involves hosting all the monitoring tools. By leveraging the cloud infrastructure, the monitoring tools can be efficiently deployed and managed. These tools expose their REST APIs, which can be accessed by making API calls to the endpoints provided by Prometheus, Jaeger, and Elasticsearch.

The proposed tool is designed to collect metrics, logs, and time series data from applications in a cloud environment. This tool utilizes APIs and libraries from open-source tools to gather data and expose these APIs to users. By utilizing these APIs, users can configure applications with the centralized tool. All the collected data is stored in MongoDB. Users can monitor and visualize metrics, logs, and time series data, eliminating the need to switch between different tools. The user can experience real-time updating of centralized cloud data visualization tools through the proposed solution.

When we discussed the code base, Used several libraries and packages for implementations. Some of the libraries shown in the following figure

```
using Jaeger.Reporters;
using Jaeger.Samplers;
using Jaeger.Senders.Thrift;
using Jaeger;
using Prometheus;
using OpenTracing.Util;
using Serilog;
using Serilog.Extensions.Hosting;
using OpenTracing;
using MongoDB.Driver;
using Serilog.Sinks.Elasticsearch;
using Serilog.Formatting.Elasticsearch;
using MonitoringAPI.Controllers;
using Microsoft.AspNetCore.Authentication.Cookies;
using Microsoft.AspNetCore.Authentication.OpenIdConnect;
using Microsoft.IdentityModel.Protocols.OpenIdConnect;
using Microsoft.AspNetCore.Authentication.JwtBearer;
using Microsoft.IdentityModel.Tokens;
using System.Text;
```

Figure 5 : Libraries and Packages

In the implementation, I handled separate models for metrics, logs, and traces. Then provide clear documentation for the user with API endpoints and data formats. Which data format allows centralising monitoring tools likewise. The following endpoints and data types are used to gather data from the user's application and prepare that data for visualisation.

API Endpoint:

POST:

https://releasex.tech/api/Metrics/metric

```
{
  "name": "string",
  "value": 0,
  "description": "string",
  "labels": {
    "additionalProp1": "string",
    "additionalProp2": "string",
    "additionalProp3": "string"
  },
  "timestamp": "2023-09-10T16:18:30.109Z"
}
```

https://releasex.tech/api/Metrics/log

```
{
  "message": "string",
  "logLevel": "string",
  "timestamp": "2023-09-10T16:18:30.107Z"
}
```


https://releasex.tech/api/Metrics/timeseries


```
{
  "metricName": "string",
  "value": 0,
  "timestamp": 0
}
```


https://releasex.tech/api/Metrics/traces


```
{
  "traceId": "string",
  "spanId": "string",
  "operationName": "string",
  "startTime": "2023-09-10T16:19:40.227Z",
  "duration": {
    "ticks": 0
  }
}
```


Above endpoints used to gather users application data. Users can pass metrics, logs, and other data by configuring their applications with those endpoints.

GET:

https://releasex.tech/api/Metrics/metrics/
https://releasex.tech/api/Metrics/logs
https://releasex.tech/api/Metrics/custommetrics
https://releasex.tech/api/Metrics/customtraces

Above endpoints for users to pass the data to the visualisation tool. Each endpoint customises each format for visualisation.

2.5.3 Visualization Tool

Monitoring tools can create data formats for visualization. Then monitoring tools provide APIs to pass collected data to visualisation tools. Then visualisation tools can show that data using graphs, charts, tables, and other proper methods. Visualising is a very tough part of this development because I have to use a few chart libraries and do some calculations to visualise the data in the charts. Otherwise, the user can get a good idea about the analysis of charts. Because of that, the visualisation tool should correctly show the data. In this development, Chart.js, D3, and ngx.js chart libraries are used for proper visualisation.

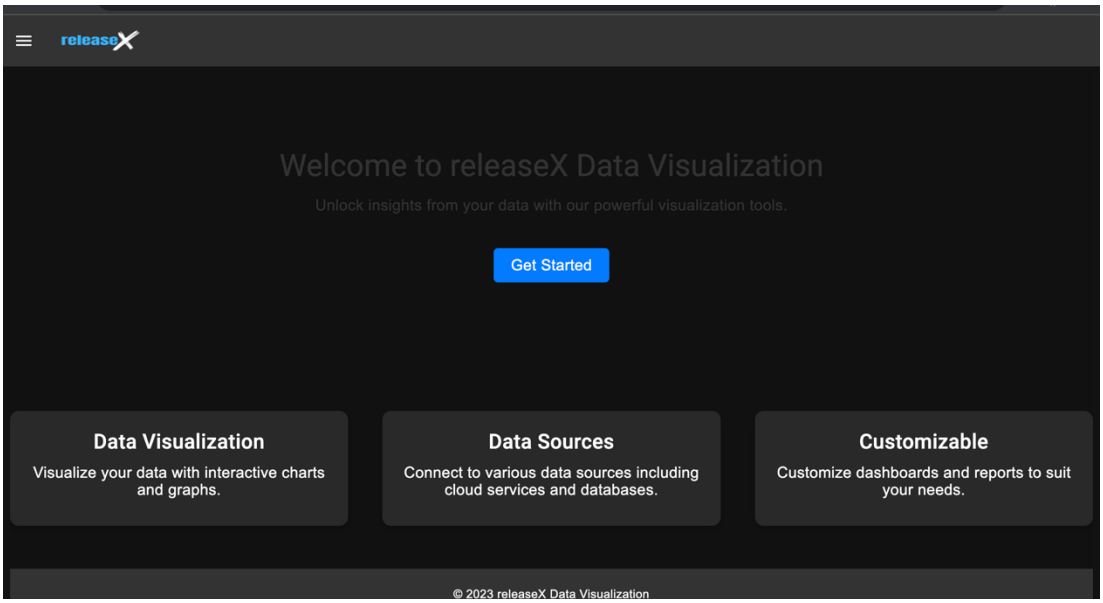

Figure 6 : Landing page of the Visualizing Tool

In the figure 6, the home screen of the visualisation tool is shown. This UI is very user-friendly, and the user can easily understand how to select data sources, visualise data, and create dashboards.

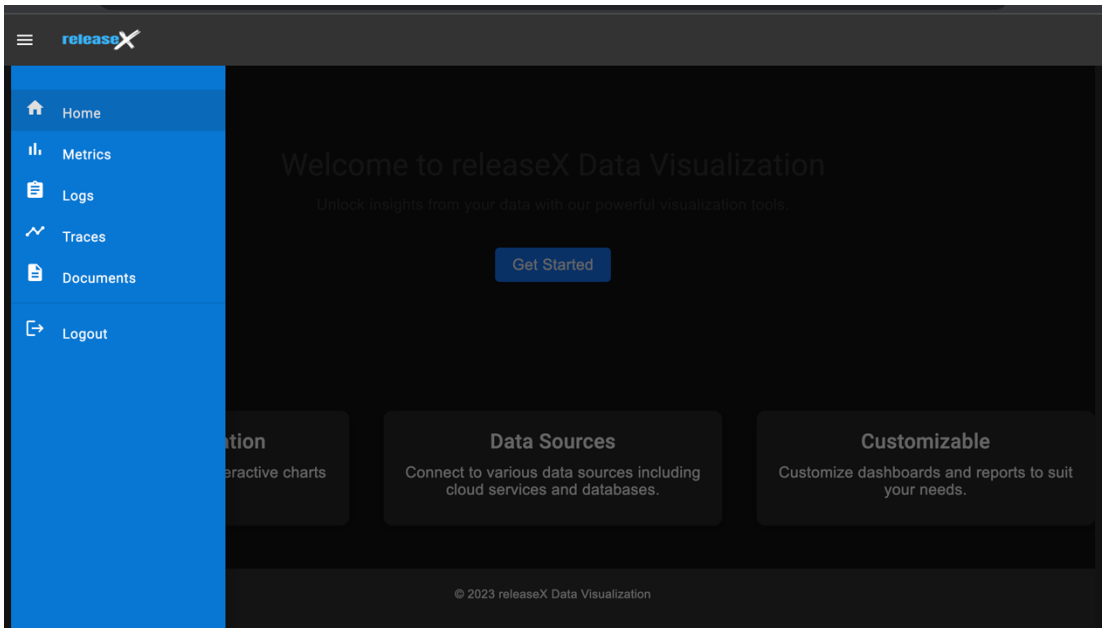

Figure 7 : Side Menu bar of visualizing Tool

The visualisation tool should be user-friendly and easy to understand. The proposed visualisation tool has a side menu bar. That menu bar includes current data types and documentation paths. The user can easily navigate to those screens by clicking on menu bar titles.

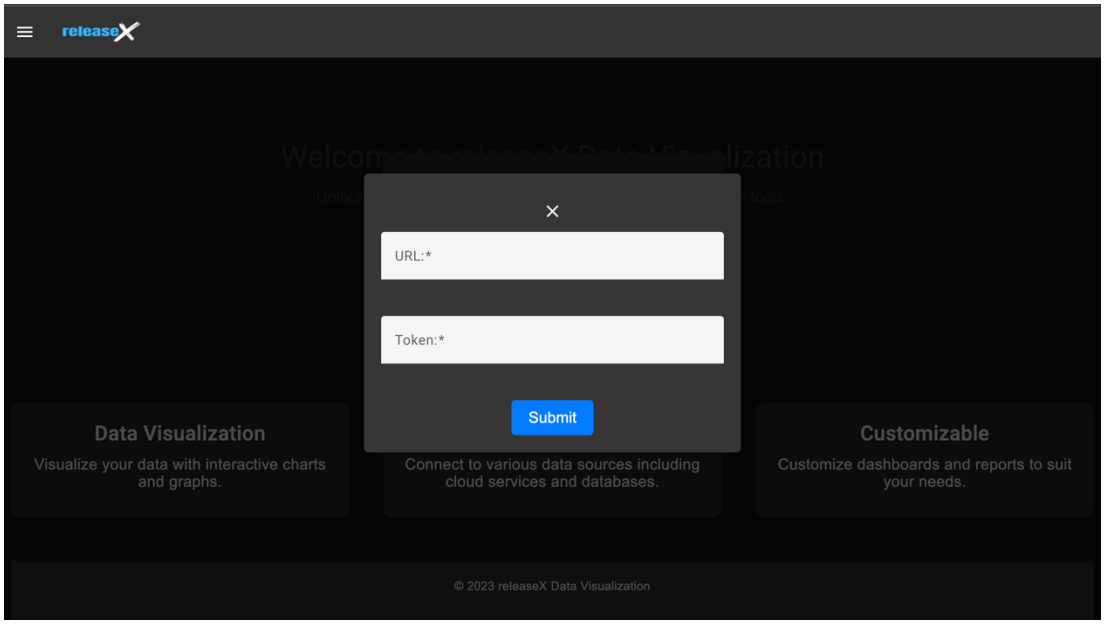

Figure 8 : Data Source Configuration

The user can configure their data source by typing the URL and JWT token in the data source screen. Then, after submitting the data source, it is connected with the visualisation tool.

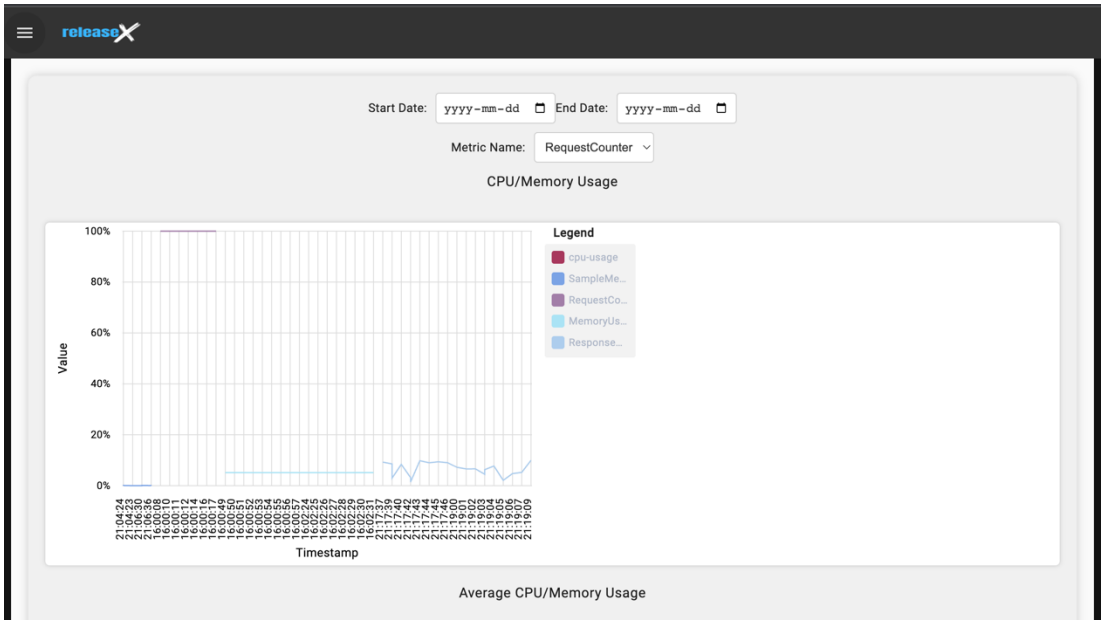

Figure 9 : CPU/Memory Usage chart-Metrics

This visualisation tool has proper data visualisation methods. This chart is one of the best for users. The user can analyze CPU and memory usage using this chart. Also proposed is a tool that provides chart filtering facilities to users. By default, users can see today's data in the charts, but they can analyse data by filtering by start date, end date, and metric name. That is a very attractive feature.

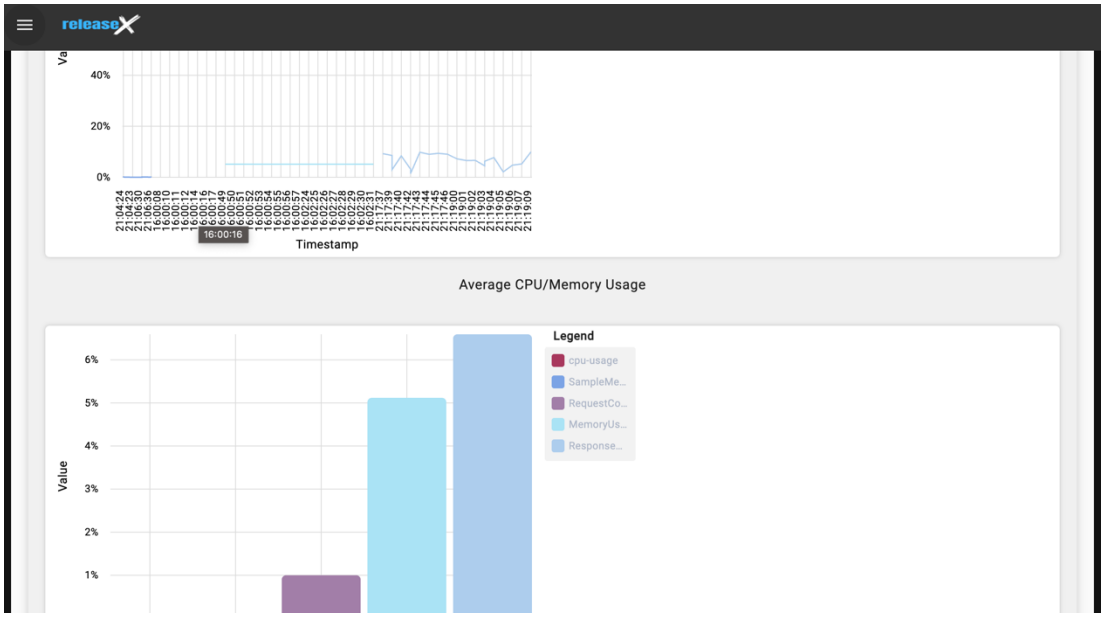

Figure 10 : Average CPU/Memory usage chart in visualization tool

When we can see data in different formats, that is very useful for clear analysis.

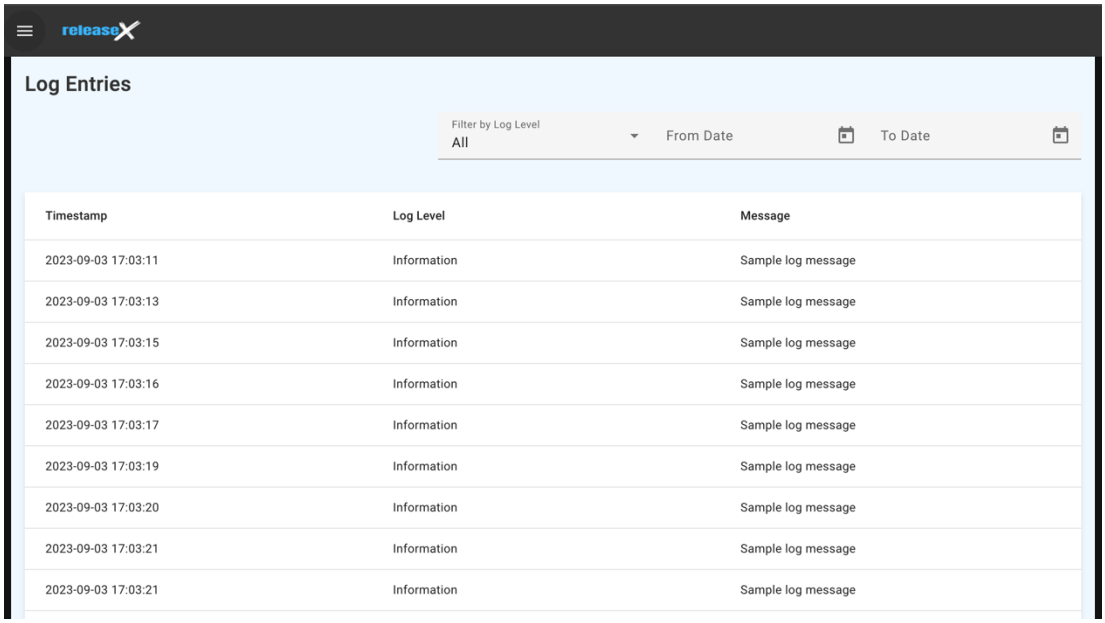

Figure 11 : Logs UI in visualization Tool

The proposed solution is a centralised data monitoring and visualisation tool. This is the advantage of this tool. The user can visualise metrics and logs using the same tool. When we discussed the log UI, it was very user-friendly and simple. The user can

simply analyse logs by log level, time stamp, and message. Also, users can filter logs by date range and log level. This feature can't see other existing visualisation tools.

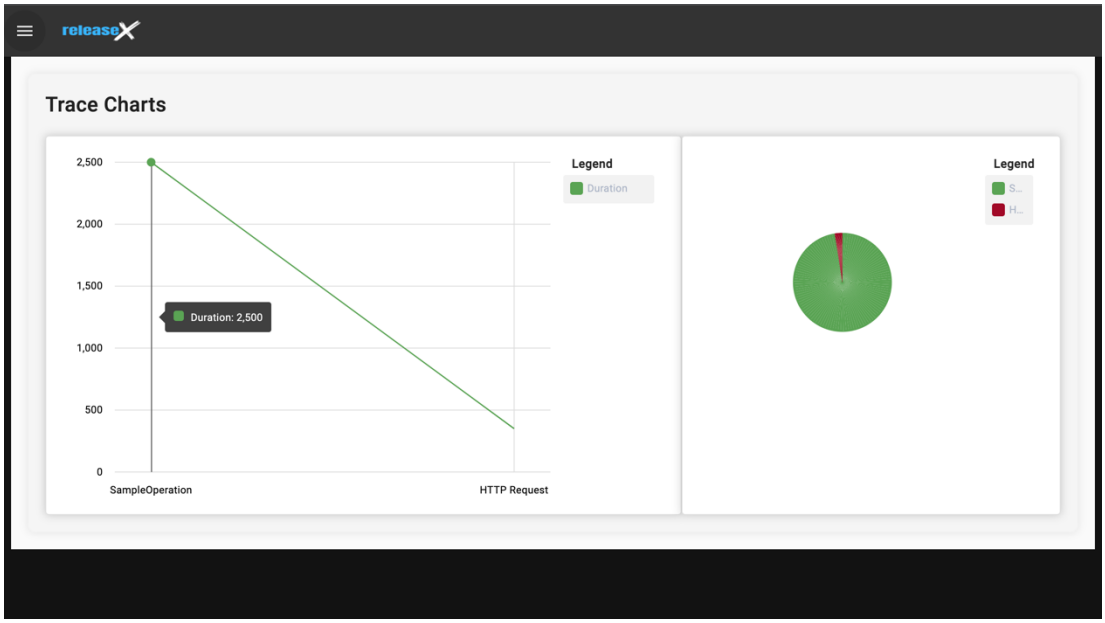

Figure 12 : Trace chart in visualization Tool

This is another data type that users can visualise using this visualisation tool. After configuring the centralised monitoring tool with the user's application, the user can easily visualise metrics, logs, and traces using the visualisation tool. This visualisation tool is demo-only. We can develop this tool with more advanced features in future developments.

2.6 Project Requirements

2.6.1 Functional Requirements

Data Integration:

- The platform must be able to integrate data from various monitoring tools such as Prometheus, Grafana, and Jaeger.
- It should support the incorporation of data from multiple sources, including logs, metrics, traces, and external data streams.

Data Visualization:

- The platform must offer a range of visualization options, including charts, graphs, tables, and dashboards.
- Users should be able to customize and create their own dashboards easily.

Integration and Compatibility:

- The platform must have APIs and connectors for seamless integration with existing tools and services.
- Compatibility with popular monitoring tools and data formats is essential.

Search and Filtering:

- Users should have the ability to search for specific data points and apply filters for data refinement.

2.6.2 Non-Functional Requirement

Performance:

- The platform should be highly responsive, with low latency for data retrieval and visualization.
- It should be capable of handling large volumes of data efficiently.

Cost Management:

- The platform should offer tools and features to monitor and optimize costs, especially as data volume increases.

Monitoring and Logging:

- The platform should include robust monitoring and logging features to track system performance and user activities.

2.7 Commercialization

The ability to offer thorough insights and real-time data visualisation is at the heart of a centralised monitoring and visualisation tool's commercial value. Businesses rely largely on the availability of data and the capacity to make informed decisions in today's tech-driven environment. Even a brief interruption in monitoring or an incomplete picture of the operation of the system can lead to expensive inefficiencies and less-than-ideal decision-making.

By combining data from numerous monitoring sources into a single platform, the suggested centralised tool provides an answer to these problems. By doing this, it ensures a comprehensive understanding of system performance and successfully addresses the drawbacks of current tools like Prometheus, Grafana, and Jaeger. Data from many sources, including as logs, metrics, traces, and external data streams, can be effortlessly integrated by users. Additionally, the tool emphasises user usability strongly. It has a user-friendly interface that is simple to use and intended for both technical and non-technical stakeholders. This user-centric approach makes data analysis and navigation simpler, enabling all users to gain useful insights from the given visualisations. This tool prioritises clarity and ease of understanding above sophisticated and specialised visualisations, which can be difficult for non-technical users to perceive. This encourages accurate conclusions and well-informed decision-making.

In essence, this solution enables organisations to make better informed decisions about the performance and health of their microservices by centralising data collecting, analysis, and visualisation in a user-friendly manner. The end result is superior business outcomes, higher efficiency, and better resource allocation. The capacity to handle and present this data coherently is a game-changer for organisations looking for predictive skills and data-driven insights in a world where current apps create enormous amounts of data.

2.8 Testing

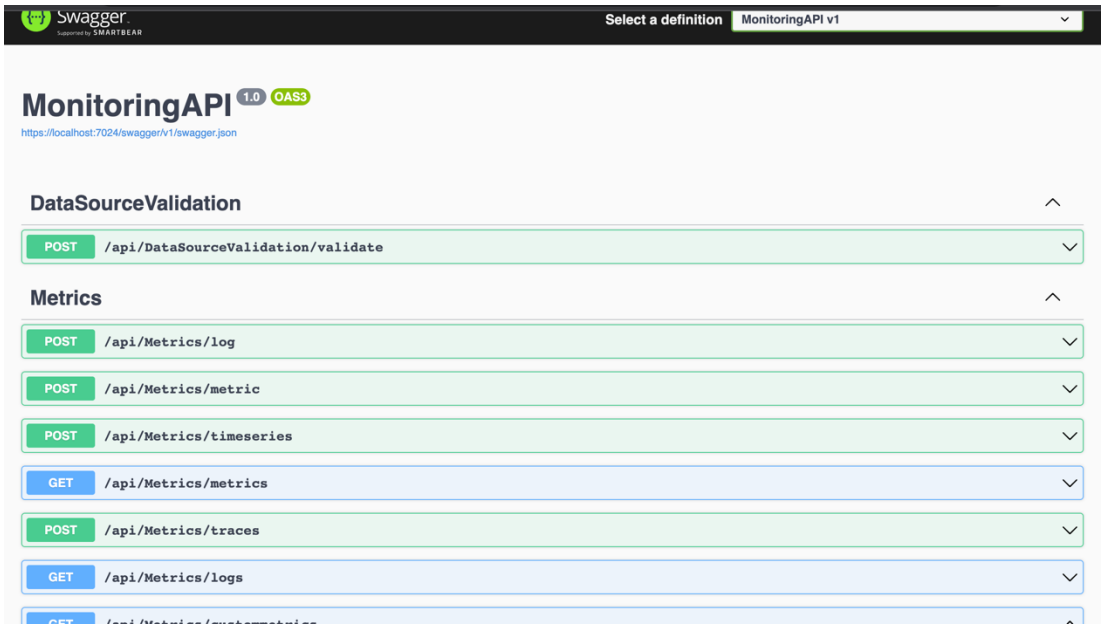

Figure 13 : Swagger UI of API Testing

For this implementation, use the ASP.NET Core MVC architecture. Swagger is the most suitable API testing platform for C# developments. After configuring the test application with end points, POST APIs should collect data. And GET APIs should retrieve collected data.

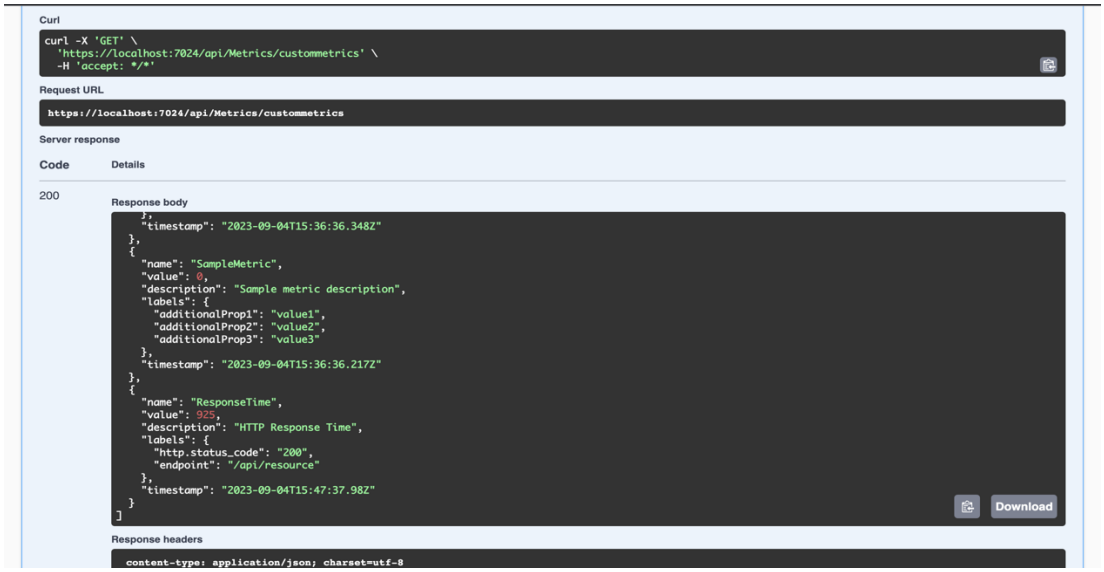

Figure 14 : testing data

2.8.1 Data Storing

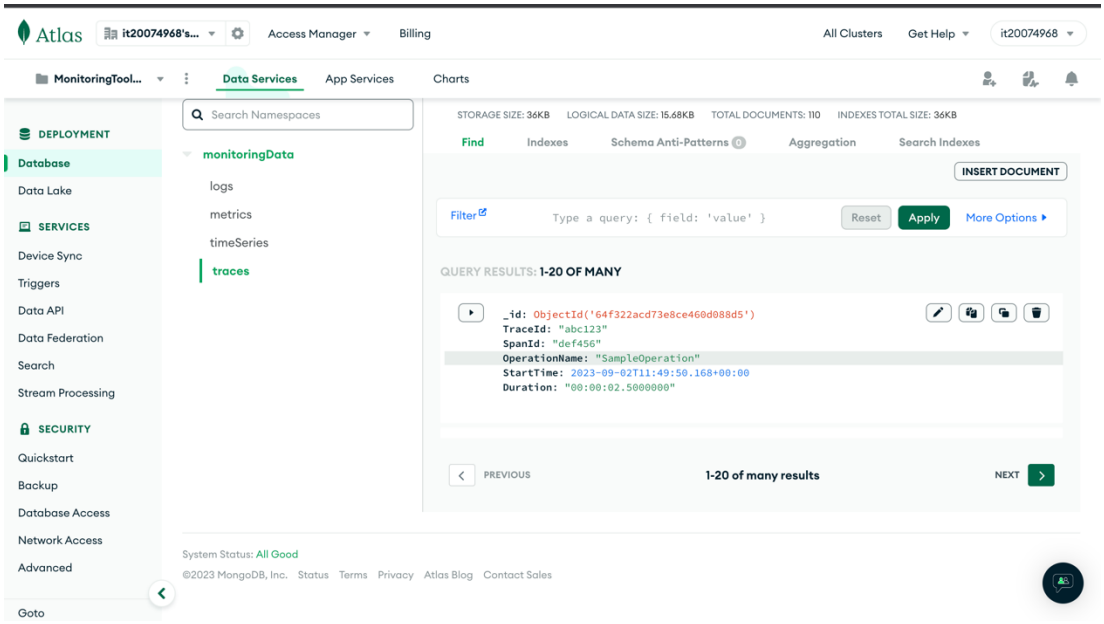

Figure 15 : MongoDb monitoring data collections.

The proposed centralised monitoring tool used MongoDB as a database for data storage. Because of that, MongoDB is a no-SQL database and supports time series data and other data.

3. RESULTS & DISCUSSION

3.1 Results

The centralized monitoring and visualization solution has a lot of benefits that organizations can anticipate. They can see all of their pertinent information in one spot, which enables them to make quicker and more informed decisions. This may enable them to work more effectively and prevent difficulties before they escalate. Additionally, the platform encourages the entire team to use data to make better decisions because it is simple for everyone to use—not just IT professionals. This enables them to make efficient use of their resources, prevent system outages, and generally perform better in their line of work. Additionally, the tool keeps them adaptable and prepared for expansion in the future because it can manage a lot of data.

3.2 Conclusion

In conclusion, our research journey has shown how important centralised monitoring and visualisation tools are in our data-driven environment. We've observed how outdated tools frequently fail, leaving businesses with disjointed data and challenging user interfaces. However, the solution, as it is described in this thesis, is quite intriguing. It all comes down to putting information from various sources in one place and making it extremely simple for everyone to understand. Smoother operations, quicker problem-solving, better resource management, and wiser decision-making are just a few of the potential advantages. Additionally, scalability ensures that your business is future-proof at a time when data is constantly expanding. Consequently, it is evident that adopting centralised monitoring and visualisation is not only an option but a must as technology continues to advance and data continues to accumulate.

3.3 Further Improvements

The need for centralised monitoring and visualisation tools in our data-driven environment has been clarified by this research's conclusion. We've identified the shortcomings of current technologies and seen the potential of a centralised solution that harmonises data sources and places a premium on usability. There is, however, always opportunity for advancement and additional study.

The incorporation of cutting-edge machine learning and artificial intelligence methods is one direction that needs more research. These technologies can provide proactive insights and even recommend the best courses of action based on past data by merging automation and predictive analytics. Additionally, the platform may become even more accessible with ongoing improvements to the user interface and experience. Iterative changes that cater to a larger user base can be guided by putting user feedback into practice and performing usability tests.

Furthermore, it is still crucial to solve security and compliance issues. Future studies can go further into protecting sensitive data, investigating encryption methods, and keeping up with changing compliance standards.

4. REFERENCES

[1] Meixia Yang, Ming Huang, “An Microservices-Based Openstack Monitoring Tool”,19 March 2020

[2] Lei Chen, Ming Xian, Jian Liu, “Monitoring System of OpenStack Cloud Platform Based on Prometheus”,12 July 2020

[3] Abhishek Pratap Singh,” A Data Visualization Tool- Grafana”, January 2023.

[4] Mahantesh Birje, Chetan Bulla,” Commercial and Open Source Cloud Monitoring Tools: A Review”, January 2020

[5] Jaeyong Choi; Suan Lee; Sangwon Kang; Jinho Kim,” A graphical administration tool for managing cloud storage system”, 02 April 2015

[6] Moustafa, N., Dawoud, D., & Wu, J. (2021). An Intelligent Multi-Layered Architecture for Microservices Monitoring Using Distributed Tracing. IEEE Access, 9, 135000-135015.

[7] Wang, J., Wang, R., & Chen, S. (2021). Cloud Service Monitoring Based on Distributed Tracing. In Proceedings of the 2021 International Conference on Intelligent Computing and Sustainable System (pp. 188-193). Springer.

[8] Oussous, A., Lahcen, A. A., & Belfkih, S. (2021). Monitoring and Alerting for Cloud Services: A Comprehensive Review of the State-of-the-Art. Journal of Cloud Computing, 10(1), 1-35.

[9] Meixia Yang, Ming Hung, “A Microservice-Based OpenStack Monitoring Tool”, 19 March 2020, doi:10.1109/ICSESS47205.2019.9040740.

[10] Lei Chen, Ming Xian, JianLiu, “Monitoring System of OpenStack Cloud Platform Based on Prometheus”,10-12 July 2020, doi:10.1109/CVIDL51233.2020.0-100.

[11] Abhishek Pratap Singh, “A Data Visualization Tool- Grafana”, Jan 26 2023.
[12] Andrea Janes, Xiaozhu Li, Valentina Lenarduzzi, “Open tracing tools: Overview and critical comparison”, Vol. 204, October 2023, 111793.
[13] Mahantesh Birje, Chetan Bulla, “Commerical and Open Source Cloud Monitoring tools: A Review”, January 2020, doi: 10.1007/978-3-030-24322-7_59

5. APPENDIX

5.1 [Appendix 1: API Controller]

```
using Microsoft.AspNetCore.Mvc;
using Prometheus;
using Serilog;
using MongoDB.Driver;
using MonitoringAPI.Models;
using Microsoft.AspNetCore.Authentication;
using Microsoft.AspNetCore.Authorization;
using Serilog.Events;
using Jaeger;
using MonitoringAPI.DTO;

namespace MonitoringAPI.Controllers
{

   [ApiController]
   [Route("api/[controller]")]
   public class MetricsController : ControllerBase
   {
      private static readonly Counter MetricCounter = Metrics.CreateCounter("my_application_metric", "My Application Metric");
      private readonly HttpClient prometheusClient;
      private readonly Serilog.ILogger logger;
      private readonly OpenTracing.ITracer tracer;
      private readonly IMongoClient mongoClient;
      private readonly IMongoDatabase database;
      private readonly IMongoCollection<LogModel> logCollection;
      private readonly IMongoCollection<MetricModel> metricCollection;
      private readonly IMongoCollection<TimeSeriesModel> timeSeriesCollection;
      private readonly IMongoCollection<TraceModel> traceCollection;
      private readonly IAuthenticationService _authenticationService;
      private static readonly IMetricServer MetricsServer = new MetricServer(hostname: "localhost", port: 9090);

      public MetricsController(Serilog.ILogger logger, OpenTracing.ITracer tracer, IMongoClient mongoClient, IAuthenticationService authenticationService)
      {
         prometheusClient = new HttpClient();
         prometheusClient.BaseAddress = new Uri("https://releasex.tech/prometheus/api/v1/");
         this.logger = logger;
         this.tracer = tracer;
         this.mongoClient = mongoClient;
         this.database = mongoClient.GetDatabase("monitoringData");//database name
         this.logCollection = database.GetCollection<LogModel>("logs");
         this.metricCollection = database.GetCollection<MetricModel>("metrics");
         this.timeSeriesCollection = database.GetCollection<TimeSeriesModel>("timeSeries");
         this.traceCollection = database.GetCollection<TraceModel>("traces");
         this._authenticationService = authenticationService;

      }

      [HttpPost("log")]
      public IActionResult LogData([FromBody] LogModel logModel)
      {
         try
         {
            if (!ModelState.IsValid)
            {
               return BadRequest(ModelState);
```

```
            }

            // Define a mapping for custom log level strings
            var logLevelMappings = new Dictionary<string, LogEventLevel>
            {
                ["information"] = LogEventLevel.Information,
                ["warning"] = LogEventLevel.Warning,
                ["error"] = LogEventLevel.Error
            };

            var logLevel = logModel.LogLevel?.ToLower();
            if (!logLevelMappings.TryGetValue(logLevel, out var serilogLevel))
            {
                serilogLevel = LogEventLevel.Information;
            }

            var timestamp = logModel.Timestamp != default ? logModel.Timestamp : DateTime.UtcNow;

            logger.Write(serilogLevel, "Log Message: {Message} - Timestamp: {Timestamp}", logModel.Message,
timestamp);


            logCollection.InsertOne(logModel);

            return Ok();
        }
        catch (Exception ex)
        {
            logger.Error(ex, "An error occurred while processing the log data.");
            return StatusCode(500, "An error occurred while processing the request.");
        }
    }

    [HttpPost("metric")]
    public IActionResult MetricData([FromBody] MetricModel metricModel)
    {
        try
        {
            if (!ModelState.IsValid)
            {
                return BadRequest(ModelState);
            }

            // Insert the metric data into MongoDB
            metricCollection.InsertOne(metricModel);

            // Process metric data
            var gauge = Metrics.CreateGauge(metricModel.Name, metricModel.Description);
            gauge.Set(metricModel.Value);

            var histogram = Metrics.CreateHistogram(metricModel.Name, metricModel.Description);
            histogram.Observe(metricModel.Value);

            // Increment the MetricCounter
            MetricCounter.Inc();

            return Ok();
        }
        catch (Exception ex)
        {
            Log.Error(ex, "An error occurred while processing the metric data.");
            return StatusCode(500, "An error occurred while processing the request.");
        }
```

```
    }


    [HttpPost("timeseries")]
    public IActionResult TimeSeriesData([FromBody] Models.TimeSeriesModel timeSeriesModel)
    {
      try
      {
        var metric = Metrics.CreateGauge("my_time_series_metric", "My Time Series Metric");
        metric.Set(timeSeriesModel.Value);

        MetricCounter.Inc();

        timeSeriesCollection.InsertOne(timeSeriesModel);
        return Ok();
      }
      catch(Exception ex)
      {
        Log.Error(ex, "An error occurred while processing the time series data.");
        return StatusCode(500, "An error occurred while processing the request.");
      }

    }


    [HttpGet("metrics")]
    public async Task<IActionResult> GetMetricsByQuery(string query)
    {
      try
      {
        // Validate the query parameter
        if (string.IsNullOrWhiteSpace(query))
        {
          return BadRequest("Query is required.");
        }

        var prometheusBaseUrl = "https://releasex.tech/prometheus"; // Prometheus server URL
        var encodedQuery = Uri.EscapeDataString(query);

        using (var client = new HttpClient())
        {
          client.Timeout = TimeSpan.FromSeconds(10); // Set a timeout

          var response = await
client.GetAsync($"{prometheusBaseUrl}/api/v1/query?query={encodedQuery}");

          if (response.IsSuccessStatusCode)
          {
            var content = await response.Content.ReadAsStringAsync();
            return Content(content, "application/json");
          }
          else
          {
            // Handle Prometheus server errors
            var errorMessage = await response.Content.ReadAsStringAsync();
            return StatusCode((int)response.StatusCode, errorMessage);
          }
        }
      }
      catch (HttpRequestException ex)
      {
        // Handle network-related errors
```

```
        return StatusCode(500, $"Error communicating with the Prometheus server: {ex.Message}");
    }
    catch (TaskCanceledException)
    {
        // Handle timeouts
        return StatusCode(504, "The request to Prometheus timed out.");
    }
    catch (Exception ex)
    {
        // Handle other exceptions
        return StatusCode(500, $"An error occurred: {ex.Message}");
    }
}


[HttpPost("traces")]
public IActionResult CollectTraces([FromBody] Models.TraceModel traceModel)
{
    try
    {
        using (var scope = tracer.BuildSpan("UserApplicationTrace")
        .WithTag("user_id", traceModel.TraceId)
        .StartActive(true))
        {

            var childSpan = tracer.BuildSpan("ChildOperation").StartActive(true);
            try
            {

                childSpan.Span.Log("ChildOperation started");

            }
            finally
            {

                childSpan.Dispose();
            }
            scope.Span.Log("UserApplicationTrace completed");
        }
        traceCollection.InsertOne(traceModel);
        return Ok();
    }
    catch(Exception ex)
    {
        logger.Error(ex, "Error occurred while collecting traces.");
        return StatusCode(500, "An error occurred while processing traces.");
    }

}

[HttpGet("logs")]
public IActionResult GetLogs()
{
    try
    {
        // Retrieve and format log data in the LogDtoClass format
        var logsData = logCollection.AsQueryable()
            .Select(log => new LogDtoClass
            {
                Message = log.Message,
                LogLevel = log.LogLevel,
                Timestamp = log.Timestamp
            })
```

```
            .ToList();

        return Ok(logsData);
    }
    catch (Exception ex)
    {
        Log.Error(ex, "An error occurred while fetching logs.");
        return StatusCode(500, "An error occurred while processing the request.");
    }
}


[HttpGet("custommetrics")]
public IActionResult GetCustomMetrics()
{
    try
    {
        // Retrieve and format metrics data in the MetricDtoClass format
        var metricsData = metricCollection.AsQueryable()
            .Select(metric => new MetricDtoClass
            {
                Name = metric.Name,
                Value = metric.Value,
                Description = metric.Description,
                Labels = metric.Labels,
                Timestamp = metric.Timestamp
            })
            .ToList();

        return Ok(metricsData);
    }
    catch (Exception ex)
    {
        Log.Error(ex, "An error occurred while fetching custom metrics data.");
        return StatusCode(500, "An error occurred while processing the request.");
    }
}


[HttpGet("customtraces")]
public IActionResult GetCustomTraces()
{
    try
    {
        // Retrieve and format trace data in the TraceDtoClass format
        var traceData = traceCollection.AsQueryable()
            .Select(trace => new TraceDtoClass
            {
                TraceId = trace.TraceId,
                SpanId = trace.SpanId,
                OperationName = trace.OperationName,
                StartTime = trace.StartTime,
                Duration = trace.Duration
            })
            .ToList();

        return Ok(traceData);
    }
    catch (Exception ex)
    {
        Log.Error(ex, "An error occurred while fetching custom trace data.");
        return StatusCode(500, "An error occurred while processing the request.");
```

```
        }
    }
  }
}
```

5.2 [Appendix 2: Open-Source tools configuration and database configuration]

```
using Jaeger.Reporters;
using Jaeger.Samplers;
using Jaeger.Senders.Thrift;
using Jaeger;
using Prometheus;
using OpenTracing.Util;
using Serilog;
using Serilog.Extensions.Hosting;
using OpenTracing;
using MongoDB.Driver;
using Serilog.Sinks.Elasticsearch;
using Serilog.Formatting.Elasticsearch;
using MonitoringAPI.Controllers;
using Microsoft.AspNetCore.Authentication.Cookies;
using Microsoft.AspNetCore.Authentication.OpenIdConnect;
using Microsoft.IdentityModel.Protocols.OpenIdConnect;
using Microsoft.AspNetCore.Authentication.JwtBearer;
using Microsoft.IdentityModel.Tokens;
using System.Text;

try
{
   var builder = WebApplication.CreateBuilder(args);
   var auth0Domain = builder.Configuration["Auth0:Domain"];
   var auth0ClientId = builder.Configuration["Auth0:ClientId"];
   var auth0ClientSecret = builder.Configuration["Auth0:ClientSecret"];
   var audience = builder.Configuration["Auth0:Audience"];

   // Add services to the container.
   builder.Services.AddControllers();
   builder.Services.AddEndpointsApiExplorer();
   builder.Services.AddSwaggerGen();
   builder.Services.AddHttpClient();
   builder.Services.AddSingleton<DiagnosticContext>();
   builder.Services.AddSingleton<Serilog.ILogger>(Log.Logger);



   // Configure CORS
   builder.Services.AddCors(options =>
   {
      options.AddPolicy("AllowAll", builder =>
      {
         builder.AllowAnyOrigin()
            .AllowAnyMethod()
            .AllowAnyHeader();
      });
   });
```

```
// Register the ITracer service
builder.Services.AddSingleton<ITracer>(sp =>
{
    // Create and configure the Jaeger tracer instance
    var tracer = new Tracer.Builder("jaeger-agent")
        .WithReporter(new RemoteReporter.Builder()
            .WithSender(new UdpSender("jaeger-agent-hostname", 16686, 0))
            .Build())
        .WithSampler(new ConstSampler(true))
        .Build();

    // Set the tracer as the global tracer
    GlobalTracer.Register(tracer);

    return tracer;
});


// Register the IMongoClient service
builder.Services.AddSingleton<IMongoClient>(sp =>
{
    var connectionString = builder.Configuration.GetConnectionString("MongoDB");
    return new MongoClient(connectionString);
});


builder.Services.AddAuthentication(JwtBearerDefaults.AuthenticationScheme)
.AddJwtBearer(options =>
{
    options.TokenValidationParameters = new TokenValidationParameters
    {
        ValidateIssuer = true,
        ValidateAudience = true,
        ValidateLifetime = true,
        ValidateIssuerSigningKey = true,
        ValidIssuer = "https://dev-7x8lcbp8wgf0exw0.au.auth0.com",
        ValidAudience = "http://localhost:4200/",
        IssuerSigningKey = new SymmetricSecurityKey(Encoding.UTF8.GetBytes("your-secret-
key"))
    };
});

builder.Services.AddAuthorization(options =>
{
    options.AddPolicy("AdminPolicy", policy => policy.RequireRole("admin"));
});

// Configure Serilog for logging
Log.Logger = new LoggerConfiguration()
.MinimumLevel.Information()
//.MinimumLevel.Override("Microsoft", LogEventLevel.Information)
//.Enrich.FromLogContext()0
.WriteTo.Console()
.WriteTo.MongoDB("mongodb://localhost:27017/monitoringData", "logs")
.WriteTo.Elasticsearch(new ElasticsearchSinkOptions(new Uri("http://localhost:9200"))
{
```

```
        IndexFormat = "your-index-name-{0:yyyy.MM.dd}",
        CustomFormatter = new ElasticsearchJsonFormatter(renderMessage: true),
        AutoRegisterTemplate = true
    })
    .CreateLogger();

    var app = builder.Build();

    // Configure the HTTP request pipeline.
    if (app.Environment.IsDevelopment())
    {
        app.UseSwagger();
        app.UseSwaggerUI();
    }

    app.UseHttpsRedirection();
    app.UseCors(builder => builder.AllowAnyOrigin().AllowAnyHeader().AllowAnyMethod());
    app.UseRouting();
    app.UseHttpMetrics();
    app.UseAuthentication();
    app.UseAuthorization();
    app.MapControllers();
    app.UseMetricServer(); // Expose Prometheus metrics endpoint



    app.UseSerilogRequestLogging();


    app.MapControllerRoute(
        name: "visualization",
        pattern: "api/visualizationconfiguration/{action}",
        defaults: new { controller = "VisualizationConfiguration" }
    );

    app.MapControllers();

    app.Run();
}
finally
{
    // Flush and close the logger
    Log.CloseAndFlush();
}
```

5.3 [Appendix 3: Metrics Visualization Component]

```
import { Component, OnInit } from '@angular/core';
import { MonitoringToolService } from '../service/monitoring-tool.service';

@Component({
  selector: 'app-metrics',
  templateUrl: './metrics.component.html',
  styleUrls: ['./metrics.component.css'],
})
```

```
export class MetricsComponent implements OnInit {
  floatingCharts: any[] = [];
  metricsData: { name: string; series: { name: string; value: number }[] }[] = [];
  chartData: { title: string; data: any[] }[] = [];
  metrics: string[] = [];
  // ngx-charts options
  view: [number, number] = [700, 400]; // Square chart dimensions
  scheme = {
    domain: ['#5AA454'],
  };
  yAxisLabel = '%';
  // Apply ngx-charts dark theme
  theme = 'dark';

  // Sample data for the bar chart
  barChartData: any[] = [];

  // Filter properties
  startDate: Date = new Date(); // Initialize with today's date
  endDate: Date = new Date();   // Initialize with today's date
  selectedMetric: string = '';


  constructor(private monitoringToolService: MonitoringToolService) {}

  ngOnInit() {
    this.selectedMetric = 'RequestCounter';
    this.fetchMetricsData();

  }
  private extractMetricNames(data: any[]): string[] {
    const metricNamesSet = new Set<string>();

    data.forEach((item) => {
      if (item.name) {
        metricNamesSet.add(item.name);
      }
    });

    return Array.from(metricNamesSet);
  }
  fetchMetricsData() {
    this.monitoringToolService.getMetrics().subscribe((data: any[]) => {
      // Process and transform data for the line chart
      const metricNames = this.extractMetricNames(data);

    // Update the metrics array with the extracted metric names
    this.metrics = metricNames;
      this.metricsData = this.transformDataForChart(data);

      // Process and transform data for the bar chart
      this.barChartData = this.transformDataForBarChart(data);
    });
  }

  // Implement your data transformation logic for the line chart here
  private transformDataForChart(data: any[]): any[] {
```

```
  const transformedData = [];
  const today = new Date(); // Get today's date

  // Group data by metric name
  const groupedData = this.groupDataByMetricName(data);

  for (const metricName in groupedData) {
    if (groupedData.hasOwnProperty(metricName)) {
      const dataPoints = groupedData[metricName]
        .filter(item => this.isToday(new Date(item.timestamp), today)) // Filter by today's date
        .map(item => ({
          name: this.formatTimestamp(item.timestamp),
          value: metricName === 'RequestCounter' ? item.value : item.value / 100,
        }));

      if (metricName === 'RequestCounter') {
        transformedData.push({
          name: metricName,
          series: dataPoints,
        });
      } else {
        transformedData.push({
          name: metricName,
          series: dataPoints,
        });
      }
    }
  }

  return transformedData;
}

private isToday(date: Date, today: Date): boolean {
  return (
    date.getDate() === today.getDate() &&
    date.getMonth() === today.getMonth() &&
    date.getFullYear() === today.getFullYear()
  );
}

private formatTimestamp(timestamp: string): string {
  const date = new Date(timestamp);
  const hours = date.getHours().toString().padStart(2, '0');
  const minutes = date.getMinutes().toString().padStart(2, '0');
  const seconds = date.getSeconds().toString().padStart(2, '0');
  return `${hours}:${minutes}:${seconds}`;
}


private transformDataForBarChart(data: any[]): any[] {
  const transformedData = [];
  const today = new Date(); // Get today's date

  // Group data by metric name
  const groupedData = this.groupDataByMetricName(data);

  for (const metricName in groupedData) {
```

```
    if (groupedData.hasOwnProperty(metricName)) {
      const dataPoints = groupedData[metricName]
        .filter(item => this.isToday(new Date(item.timestamp), today)) // Filter by today's date
        .map(item => ({
          name: metricName,
          value: item.value / 100, // Always divide by 100
        }));

      const averageValue = this.calculateAverage(dataPoints);

      transformedData.push({
        name: metricName,
        value: averageValue,
      });
    }
  }

  return transformedData;
}

// Calculate the average value for a given array of data points
private calculateAverage(dataPoints: any[]): number {
  if (dataPoints.length === 0) {
    return 0;
  }

  const sum = dataPoints.reduce((total, dataPoint) => total + dataPoint.value, 0);
  return sum / dataPoints.length;
}


private groupDataByMetricName(data: any[]): { [key: string]: any[] } {
  return data.reduce((grouped, item) => {
    if (!grouped[item.name]) {
      grouped[item.name] = [];
    }
    grouped[item.name].push(item);
    return grouped;
  }, {});
}

// Add a method to create new charts
addFloatingChart(chartData: any) {
  this.floatingCharts.push(chartData);
}

// Filter data based on date range and metric name
filterData() {
  if (!this.startDate && !this.endDate) {
    // Filter data to show today's values for the selected metric only
    const today = new Date();
    this.barChartData = this.transformDataForBarChart(
      this.barChartData.filter(item => this.selectedMetric === 'RequestCounter' || this.isToday(new
Date(item.name), today))
    );
    this.metricsData = this.transformDataForChart(
```

```
      this.metricsData.filter(metric => this.selectedMetric === 'RequestCounter' || this.isToday(new
Date(metric.series[0].name), today))
    );
   } else if (this.selectedMetric === 'RequestCounter') {
    // Apply filtering logic for 'RequestCounter' metric without date filtering
    this.barChartData = this.transformDataForBarChart(this.barChartData);
    this.metricsData = this.transformDataForChart(
     this.metricsData.filter(metric => metric.name === 'RequestCounter')
    );
   } else {
    // Apply your regular filtering logic here
    this.barChartData = this.transformDataForBarChart(this.barChartData);
    this.metricsData = this.transformDataForChart(this.metricsData);
   }
  }


  // Add this function to your component class
yAxisTickFormatting(value: number): string {
  return `${value}%`; // Format the value as a percentage with '%' symbol
}

}
```

5.4 [Appendix 4: Logs Visualization Component-Angular]

```
import { Component, OnInit, ViewChild } from '@angular/core';
import { MonitoringToolService } from '../service/monitoring-tool.service';
import { LogEntry } from 'src/DTO/logs.dto';
import { MatPaginator } from '@angular/material/paginator';
import { MatTableDataSource } from '@angular/material/table';

@Component({
  selector: 'app-logs',
  templateUrl: './logs.component.html',
  styleUrls: ['./logs.component.css'],
})
export class LogsComponent implements OnInit {
  logs: LogEntry[] = [];
  selectedLogLevel: string = 'All';
  fromDate: Date | null = null;
  toDate: Date | null = null;
  pageSize: number = 10;

  displayedColumns: string[] = ['timestamp', 'logLevel', 'message'];
  dataSource: MatTableDataSource<LogEntry> = new MatTableDataSource<LogEntry>(this.logs);

  @ViewChild(MatPaginator) paginator: MatPaginator | null = null;

  constructor(private monitoringToolService: MonitoringToolService) {}

  ngOnInit() {
    this.fetchLogData();
  }
```

```
  ngAfterViewInit() {
    if (this.paginator) {
      this.dataSource.paginator = this.paginator;
    }
  }

  fetchLogData() {
    this.monitoringToolService.getLogs().subscribe((data: any[]) => {
      this.logs = data.map((log) => ({
        message: log.message,
        logLevel: log.logLevel,
        timestamp: new Date(log.timestamp),
      }));

      this.applyFilter();
    });
  }

  applyFilter() {
    const filteredLogs = this.logs.filter((log) => {
      const isLogLevelMatch = this.selectedLogLevel === 'All' || log.logLevel ===
this.selectedLogLevel;
      const isDateRangeMatch =
        (!this.fromDate || log.timestamp >= this.fromDate) &&
        (!this.toDate || log.timestamp <= this.toDate);

      return isLogLevelMatch && isDateRangeMatch;
    });

    this.dataSource.data = filteredLogs;
  }
}
```