Aggregation and Observability in Microservice Accelerator

Submitted By

Jay Parmar

20MCEC07

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

INSTITUTE OF TECHNOLOGY

NIRMA UNIVERSITY

AHMEDABAD-382481

May 2022

Aggregation and Observability in Microservice Accelerator

Major Project

Submitted in partial fulfillment of the requirements

for the degree of

Master of Technology in Computer Science and Engineering

Submitted By

Jay Parmar

(20MCEC07)

Guided By

Dr. Pooja Shah

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

INSTITUTE OF TECHNOLOGY

NIRMA UNIVERSITY

AHMEDABAD-382481

May 2022

Certificate

This is to certify that the Major project entitled **" Aggregation and Observability in Microservice Accelerator "** submitted by **Jay Parmar (20MCEC07)**, towards the partial fulfillment of the requirements for the award of degree of Master of Technology in Computer Science and Engineering of Nirma University, Ahmedabad, is the record of work carried out by him under my supervision and guidance. In my opinion, the submitted work has reached a level required for being accepted for examination. The results embodied in this Major Project, to the best of my knowledge, haven't been submitted to any other university or institution for award of any degree or diploma.

Dr. Pooja Shah
Internal Guide & Associate Professor
CSE Department
Institute of Technology
Nirma University, Ahmedabad

Dr. Sudeep Tanwar
Professor & PG Coordinator (M.Tech - CSE)
CSE Department
Institute of Technology
Nirma University, Ahmedabad

Dr. Madhuri Bhavsar
Head of Dept.
CSE Department
Institute of Technology
Nirma University, Ahmedabad

Dr. Rajesh Patel
Director
Institute of Technology
Nirma University, Ahmedabad

Statement of Originality

I, **Jay Parmar**, **20MCEC07**, give undertaking that the Major Project entitled **"Aggregation and Observability in Microservice Accelerator"** submitted by me, towards the partial fulfillment of the requirements for the degree of Master of Technology in **Computer Science & Engineering** of Institute of Technology, Nirma University, Ahmedabad, contains no material that has been awarded for any degree or diploma in any university or school in any territory to the best of my knowledge. It is the original work carried out by me and I give assurance that no attempt of plagiarism has been made.It contains no material that is previously published or written, except where reference has been made. I understand that in the event of any similarity found subsequently with any published work or any dissertation work elsewhere; it will result in severe disciplinary action.

Signature of Student

Date:

Place:

Endorsed by

Dr. Pooja Shah

(Signature of Guide)

IFIN HR- 2022
27-04-2022

CERTIFICATE

To Whom It May Concern

Formal Data:

Student Name: Jay Kishor Parmar

Institution: Institute of Technology, Nirma University

Organization: Infineon Technologies India Pvt. Ltd.

Project Instructors/ Managers: Anusha Shetty

Evaluation of work:

Jay Kishor Parmar is working as a Student Trainee with us from 05-Jul-2021 till 31-May-2022 and is working on project "**Aggregation & Observability in Microservice Accelerator".**

Jay Kishor Parmar is an avid and independent learner, has good analytical & application skills and has shown exemplary performance during the internship period.

We wish Jay Kishor Parmar a long fruitful career and success in future endeavors.

For Infineon Technologies India Pvt. Ltd.

Thara Aiyanna
HR Manager

Acknowledgements

It gives me great pleasure to express my heartfelt gratitude to **Dr. Pooja Shah** for her continuous encouragement during this project. Her gratitude and unwavering guidance have served as a powerful motivator for me to strive for greater heights.

It gives me an immense pleasure to thank **Dr. Madhuri Bhavsar**, Hon'ble Head of Computer Science And Engineering Department, Institute of Technology, Nirma University, Ahmedabad for her kind support and providing basic infrastructure and healthy research environment.

A special thank you is expressed wholeheartedly to **Dr. Rajesh Patel**, Hon'ble Director, Institute of Technology, Nirma University, Ahmedabad for the unmentionable motivation he has extended throughout course of this work

- Jay Parmar

20MCEC07

Abstract

Various organizations need highly scalable and available architecture in this fast forwarding world. However, it is incredibly challenging to scale monolithic architecture based on user response. That is why numerous prominent tech giants use microservice architecture. Microservice architecture also comes with various challenges; however, it offers more. Various design patterns are available to overcome these challenges in a microservice architecture. In this paper, we are addressing observability design patterns. We can monitor, analyze, and inspect our application much more efficiently using this design pattern. As microservice architecture is rapidly evolving, various open-source tools are available in the market. However, there is no such hard-and-fast tool there for all the projects; it entirely depends on the business requirement of the project. This paper discussed some of the open-source tools and their advantages and how we can use these tools to achieve observability design patterns.

Abbreviation

Abbreviation	**Explanation**
APM	Application Performance Monitoring
API	Application Programming Interface
DOS	Denial Of Service
ELK	Elasticsearch, Logstash, and Kibana
IDC	International Data Corporations

List of Figures

List of Tables

Contents

Chapter 1

Introduction

This chapter discusses a monolithic application, why we should avoid it for scalable projects, and the benefit of microservice architecture over monolithic architecture.

1.1 Introduction

Recently due to the improving Information Technology (IT) infrastructure and advancing technologies, there has been an immense surge in the Internet usage worldwide. Global internet users have grown by more than 330 million in 2021, reaching a total of more than 4.7 billion at the start of April 2021[2]. Furthermore, it is predicted that data creation will grow to more than 180 Zetta Bytes (ZB) by the year 2025. Increasing internet usage causes demand for high-performance applications that can efficiently handle this data. There was a time when industry centered on solely one type of architecture, i.e., monolithic architecture. The concept of monolithic architecture is excellent in itself. However, this architecture has some problems regarding scalability and rapid development.

There was a time when industry centered on solely one type of architecture, i.e., monolithic architecture. The concept of monolithic architecture is excellent in itself. However, this architecture has some problems regarding scalability and rapid development. Monolithic architecture provides tight coupling between all the services, where the dilemma reaches for scalability. All the services are tightly coupled with each other, which means that all services are dependent on each other and to scale such application itself is a complex task that directly impacts the performance. Another problem with monolithic is that if any single service failed it cause a complete system failure. Deploying the monolithic application is also one of the significant concerns for the project; as for a single

service update, we need to redeploy the complete system again. Rapid development and scalability are the main worries for monolithic architecture; here comes the rescuer we have, microservices architecture.

N. Dragoni describes *Microservice architecture* as the development of a series of small services that work as a single application. It provides loose coupling between services which eventually helps scalability and rapid delivery.

1.2 Understanding Microservice Architecture

Microservice architecture is creating a boom in the current market. The main reason is that it is more scalable, faster, reliable, and can quickly adopt new technologies than monolithic architecture. The International Data Corporation (IDC) indicated that by the ending of the year 2021, around 80% of cloud based software would be developed using microservice architecture style [3]. There were several claims about the originator of microservices. In one of the cloud computing conferences, The word "micro web services" was first time used by Dr. Peter Rogers in the year 2005 [4]. "Microservices" themselves premiered at an event for software architects in 2011, where the word was used to represent a style of architecture that numerous attendees were exploring at the time. "Microservices" itself was first exhibited at an software design event in 2011, where the term was used to represent the architectural style that many people were exploring at the time. Microservice architecture in itself is a magnificent idea for accelerated and scalable development. We can use this architecture in any of the projects for rapid development. Here is the representation of microservice architecture.

Figure 1.1 shows the pictorial representation of microservice architecture. It demonstrating that how a client can reach various microservices using API Gateway. On the client-side, he/she may not aware of the scenario of microservices. For them, it will be like a traditional API call but from a developer perspective, it provides more adaptability, scalability, and rapid development.

Nowadays, many large-cap companies, such as Amazon, Netflix, have moved their applications and systems to the cloud, because cloud computing allows these organizations to scale their computing resources as per their usage [5]. And in those companies microservices are widely used in their applications.

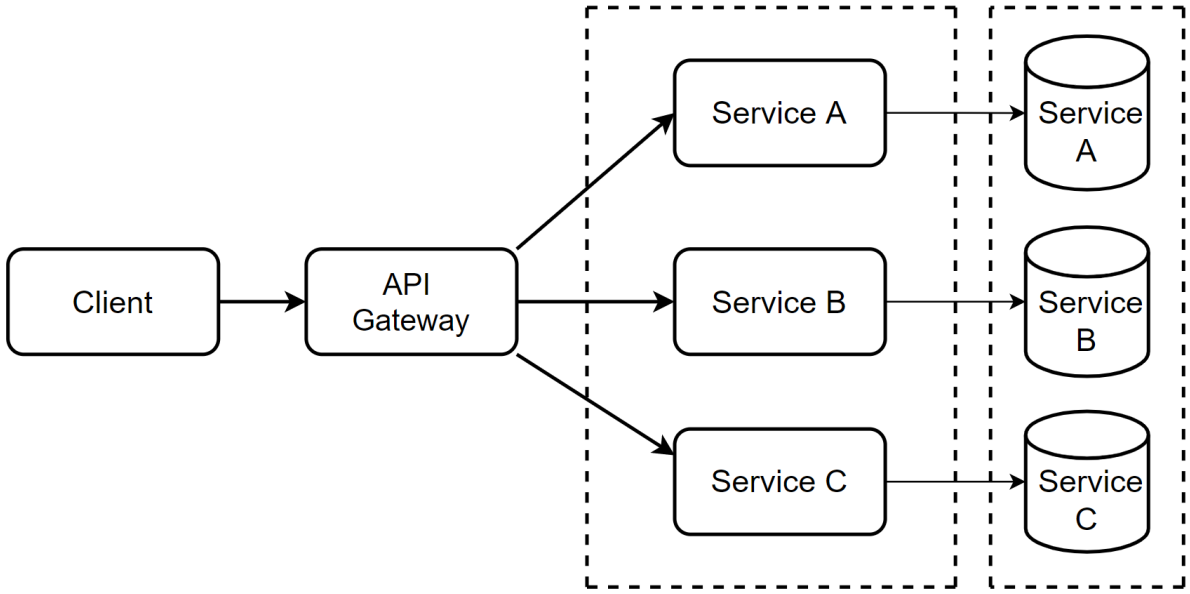

Figure 1.1: Microservice Architecture

1.3 Microservice vs Monolithic

There are numerous advantages of adopting microservice architecture over monolithic architecture when it comes to scalable applications. In this section, we will see the remarkable useful advantages of microservices.

1. Coupling: as monolithic architecture has tight coupling with each service, it is very challenging to make any changes on the system directly. On other side, microservice architecture provides loose coupling between services which makes them independent of each other and extremely useful for rapid development.

2. Scalability: scaling monolithic applications is not an unachievable task but the plenty of energy and time it needs will undoubtedly be more than the amount demanded by any microservice application as each service is independent.

3. Testing: testing microservice application is easy as compared to monolithic applications as in microservice applications each service can be individually tested where as in monolithic applications it is very complex to test each service separately.

4. Frequent Development: frequent Development is primary concerns with a monolithic application as for single development/changes we need to the redeployed whole application again on the server. If we confronted this situation with microservice

applications, it outperforms the monolithic application as we can efficiently deploy individual services as per the demand without harming other services.

5. Mix of technologies: it can be obtainable with both the architectures, but when we have to do it efficiently that we can say that microservice architecture can easily accomplish mixing of technology.

6. Data Isolation: isolating data for a very difficult task for any application. for monolithic applications, data isolation is a quite challenging task as multiple resources are using the same data across the system and it is pretty obscure to isolate data for each service wherein microservice architecture we can comfortably specify a database for each service through which we can efficiently obtain data isolation.

Data described in Table 1.1 are summarized based on analysis of multiple research papers where we have compared several parameters for sentencing both architectures.

Parameter	**Monolithic Architecture**	**Microservice Architecture**
Reliability	Less	More
Scalability	Less	More
Adoptability	Challenging	Well-to-do
Rapid Development	Complex	Moderate
Testing	Easy	Complex
Data Isolation	Difficult	Simple
Coupling	Tight	Loose

Table 1.1: Comparison between Microservice and Monolithic Architecture.

1.4 Microservice Accelerator

Microservices Accelerator provides container-ready microservices that involve a project configuration and a collection of modules that implement common requirements for standard microservice-based solutions.

1.5 Challenges with Microservice Architecture [1]

One oftentimes imagines that if we hold this many advantages from the microservice architecture then why can't we stop the monolithic application and start using the only microservice application. This is not the case, there is another side of the coin that organizations need to pay for using microservice architecture.
There are various challenges we may encounter in using microservice architecture. The few most common hurdles are listed below.

1. Versioning: we oftentimes update any service for a specific target and as multiple services possibly are updated at any delivered moment, so without thoughtful design, we might produce a lot of compatibility issues which may end up in system malfunction which will be hard to resolve.

2. Data integrity: as each microservice has its data persistence, it is quite complicated to maintain the integrity of data shared across multiple microservices.

3. Monitoring and observability: for a large-scale project, we may have plenty of services for our application and if many services are interdependent and any of them is failed, it is notable difficult to identify, log, and trace the failure.

4. Testing: testing individual services it is usually a simple task to do in microservices but the problem arrives when more services are connected to each other and testing such a scenario in microservice is really challenging.

These are a few common challenges we wrote down, apart from these we have many more difficulties in integrating microservice architecture. So, if the business use case is small we can go with the monolithic architecture. But for a large-scale application, it is worthy of going with microservice architecture.

1.6 Design Patterns

As migrating from monolithic to microservice is not a straightforward task. We need to follow standard patterns to adopt microservice architecture in existing monolithic application. We are primarily focused on API Gateway Pattern and Observability Pattern for this project.

1.6.1 API Gateway Patterns

In a microservice architecture, client applications are often required to use more than one microservices and if we directly provide the endpoint for each required microservices to the client then it leads to a security threat to our application as well as if we change anything in endpoints client also need to changes this in their application. The solution for that is we have an API gateway.

It is a type of link between microservices and client browsers. It is very helpful to protect the real endpoint of microservices. The Gateway API is the only server access point in the system. It integrates the internal system configuration and provides an API designed for each client.

1.6.2 Observability Gateway

This pattern is beneficial for understanding and observing microservice applications efficiently. There is a typical miss perception that people usually assume that monitoring and observability are the same. However, there are two different terms. Monitoring will tell us that there is a problem; however, observability will tell us where the problem occurred. We cannot monitor our application until we observe it thoroughly. This pattern consists of Log aggregation, Application metrics, Audit logging, Distributed tracing, Exception tracking, and Health check API.

1.7 Scope of Work

There is numerous functionality we can achieve using microservice architecture such as logging, tracing, monitoring, security, scalability, rapid development, etc. By the scope of this project, we are major working on security through API gateway as well as monitoring and observability of microservice architecture.

Chapter 2

Literature Survey

2.1 Paper I: "A Comparative Review of Microservices and Monolithic Architectures"

2.1.1 Objective

In this research paper [6], the authors mainly concentrate on comparing monolithic architecture and microservice architecture in terms of performance, to conclude how these architectures perform in different situations using various experimental setups. The authors decided to further investigate more about the performance of both the application in more depth as many differences in the literature are available.

2.1.2 Methodology

Authors use the platform "JHipster" for developing and analyzing microservice and monolithic applications. Web applications consist of popular Java framework Spring Boot and for front-end Angular JS frameworks.
The application that was developed for this particular paper consisted of three services.

First, JHipster Registry is a primary part of the microservices architecture. Second, the microservice application will produce the backend capabilities by API. Third, the microservices gateway is the front-end of the entire system which will incorporate all the APIs of every microservice application in the system.

JMeter was also utilized to test the performance of these applications. They perform various testing scenarios such as load testing, concurrency testing, endurance testing, and

results are different for different scenarios.

2.1.3 Result and Conclusion

From the result, they have analyzed that for small load monolithic performs well while load starts increasing microservice outperform monolithic application. Regarding throughput as they have fixed the number of requests and observe that monolithic application shows higher throughput as compared to microservices.

2.1.4 Performance matrix

Throughput and response time;

2.2 Paper II: "Microservices: Yesterday, Today, and Tomorrow"

2.2.1 Objective

The foremost objective for this paper is to provide a survey that essentially addresses comprehensive guidance on microservice to anyone new to this terminology and the possible concern with this architecture.

2.2.2 Methodology

This paper [7] informs us about three-phase of microservices 'yesterday', 'today', and 'tomorrow'. It mainly illustrated that how popular applications are influencing by microservice architecture over the monolithic architecture. It also explains that how microservices are powerful to accommodate flexibility, modularity, evolution, independency, and size reduction in infrastructure.

2.2.3 Result and Conclusion

As a result of this research, the authors provided the reader with references to the literature and guidance of services and microservices. They also discussed the possible dilemmas with microservices concerning security, network complexity, and heterogeneity.

2.3 Paper III: "Infrastructure Cost Comparison of Running Web Applications in the Cloud Using AWS Lambda and Monolithic and Microservice Architectures"

2.3.1 Objective

The main objective of this paper [8] is to presents a cost comparison of web application development and deployment using three different methods: the first is monolithic architecture, the second is microservice architecture Managed by the cloud customer, and the third is microservice architecture Managed by the cloud provider. The goal is to examine how the infrastructure costs are affected by the development of each architecture.

2.3.2 Methodology

As a purpose method, the authors decided to build a similar application with all three strategies to examine the cost and scalability of the application. Two services are created, first one is to generate the payment plan which includes a set of payments (from 1 to 6 months) and the second service is used to retrieve the existing plan and its corresponding set of payments. They have created these two services for each architecture for investigating performance for each structure.

2.3.3 Result and Conclusion

From the case study, the authors observed that if an application has a meager number of users (hundreds or thousands of users) it is eternally more beneficial to go with monolithic architecture for faster performance. Moreover, they concluded that microservice helps reduce the infrastructural cost in comparison with a monolithic architecture. They also discovered that cloud services such as AWS lambda allow companies to reduce their infrastructure cost up to 77.08%.

2.3.4 Performance matrix

Cost comparison, Response time, Performance test;

2.4 Paper IV: "Emergent Microservices in Emergent Ecosystems"

2.4.1 Objective

In this research paper [9], the objective of the author is to propose the fresh concept of emergent microservices and emergent ecosystems. Basically, idea is to add autonomous loops in each service and making them able to evolve their behavior to sustain run-time adaptation.

2.4.2 Methodology

Authors brought this concept to allow the entire microservice ecosystem to shift the control of management and adaptation processes from human to machine. This comes to reduce the human efforts for doing administration and all. By this proposed concept authors are altering the human role from developer to DevOps.

2.4.3 Result and Conclusion

By proposing this concept, the authors concluded that their concept will reduce human efforts and led the microservice ecosystem to automation. It is a very recent theory that yet more analysis is required in this concept for achieving load balancing and horizontal auto-scaling and also evaluating the impact of this idea on the whole microservice-base system.

2.5 Paper V: "Automated deployment of a microservice-based monitoring infrastructure"

2.5.1 Objective

The main objective of this research paper [10] is to deployed automatic monitoring infrastructure for microservice applications. Specific consideration is given to protect the distinction between core and custom functionalities, and the on-demand creation of a cloud service.

2.5.2 Methodology

They started by defining an easy model of monitoring infrastructure that gives an interface among user and cloud management systems. The prototype implemented by the authors shows the applicability of the abstract control flow. They have used plain Java as the application programming language (a Ruby version is on the way) and Docker for the containers.

2.5.3 Result and Conclusion

In the conclusion of this paper, the investigated automated deployment of monitoring infrastructure in container-based distributed systems and their result determines that profoundly customizable monitoring infrastructure can be effectively given as a service.

2.6 Paper VI: "A dashboard for microservice monitoring and management"

2.6.1 Objective

Benjamin Mayer and Rainer Weinreich [11] describes the concept of monitoring and managing microservices. It is an experiment dashboard idea that helps an organisation collect runtime information of microservices. The dashboard includes a system overview, runtime info of microservices, service interaction, and comparison of different services.

2.6.2 Methodology

They initially describe basic concepts and present important usage scenarios and views currently supported in the monitoring dashboard. Authors divided visualization into two categories: static and dynamic information. Static information includes commits per developer, the last commit for a service, etc. Dynamic information contains response time, total request count, failure rate, etc.

2.6.3 Result and Conclusion

In the conclusion of this paper, the authors have created an experimental dashboard that can combine information from various sources. They were still working on full fledge at the time paper was written.

2.7 Paper VII: "Demonstration of an observability framework for cloud native microservices"

2.7.1 Objective

As cloud-native microservices gains popularity, it also comes along with their challenges. Marie-Magdelaine *et al.* [12] demonstrate an observability framework for cloud-native microservices.

2.7.2 Methodology

Their proposed approach defines four logic layers for four different tasks, namely collection & retrieval of data, raw data storage, processing & correlation, and visualization & alerting. They also explain that the observability framework can effectively analyze the internal behaviour of microservice.

2.7.3 Result and Conclusion

This paper provides some understanding of the concepts, features and prototypes of observability and cloud-native applications. It also proposed the observability framework, which can be used to understand the monitor complex distributed systems.

Chapter 3

Working Environment

Following are the technologies that we explored and implemented for this project and we have used .NET core as base programming language.

Figure 3.1: Technologies and Working Environment.

- Ocelot: is an open-source library provided by Three-Mammals for integrating API gateway in .NET. It also provides various features such as Routing, Request Aggre-

gation, Service Discovery with Consul Eureka, Authentication and Authorization, Rate Limiting, Caching, Load Balancing, and many more.

- Redis: Redis is an open-source in-memory database that can be used for cache and message brokers. It can handle millions of pieces of data within a short duration of time. It also provides supports for multiple languages.

- Kafka: provides messaging service in publisher/subscriber fashion. We will use it to set up inter-communication between microservice.

- OpenShift: is provides a platform where we can deploy our application as a container image. We can also manage this container-based application on openshift. It is widely used at the enterprise level.

- Jaeger: is an open-source distributed tracing tool. It will help us to identify the execution flow of requests. It has various filters available to filter request and debug it in a more efficient fashion.

- OpenTelemetry: is an open-source observability framework. It collects the telemetry data from the application and forwards its respective exports(ex. Prometheus or Jaeger).

- ELK Stack: combines three different tools, namely Elastic, Logstash, and Kibana. It is a log aggregation tool, but it does more than that; we can also visualize those logs with the help of Kibana.

- SigNoz: is an open-source tool for metric and tracing data visualization. It was founded in 2020 to deliver better observability corresponded to existing tools.

- ZipKin: is an open-source distributed tracing system to collect trace information. Zipkin assists us to locate precisely where a request to the application has failed or spent a long time.

- Prometheus Grafana: Prometheus is an open-source monitoring system that scraps application metrics data. Grafana is a great data visualization tool.

Chapter 4

Architecture and Flow

Microservice architecture is creating a boom in the current market. The main reason is that it is more scalable, faster, reliable, and can quickly adopt new technologies than monolithic architecture. Migrating from monolithic to microservice architecture is not straight forward task. We need to follow specific tools and standard patterns to achieve it. So we used the concept of a microservice accelerator. This concept emphasizes Rapid application development. We intend to enhance the development activities by providing standard patterns and code templates. This project will reduce the time and effort required to implement common microservice functionalities. It also enables rapid integration with CI/CD activities.

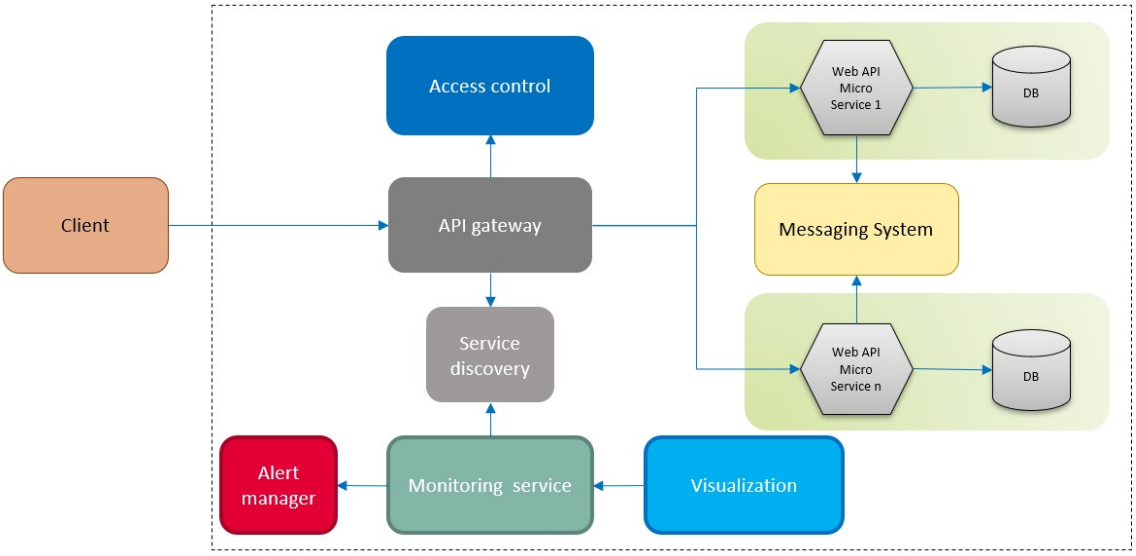

Figure 4.1: Architecture Diagram.

Figure 4.1 is an architecture diagram of this project where the primary pinpoint of

focus is API Gateway with Access control and Monitoring service. As you can see in the figure, API Gateway is the only entry point in the application, making it vulnerable. So we must have a restriction for requests at the gateway level. This problem can be resolved using the API Gateway pattern. We are using an access control layer over the top of the API Gateway. This project's second module resolves a debugging problem in microservices using the Observability pattern. This pattern will include logging, tracing, metrics, and standard visualization.

We have placed access control on top of API Gateway, and it will provide functionalities such as Rate-limiting, IP whitelisting/blacklisting, and caching. On the other hand, in the observability pattern (monitoring service), we will have a log aggregator that will aggregate the logs and forward them to the log visualization tool. Similarly, it will process traces and metrics.

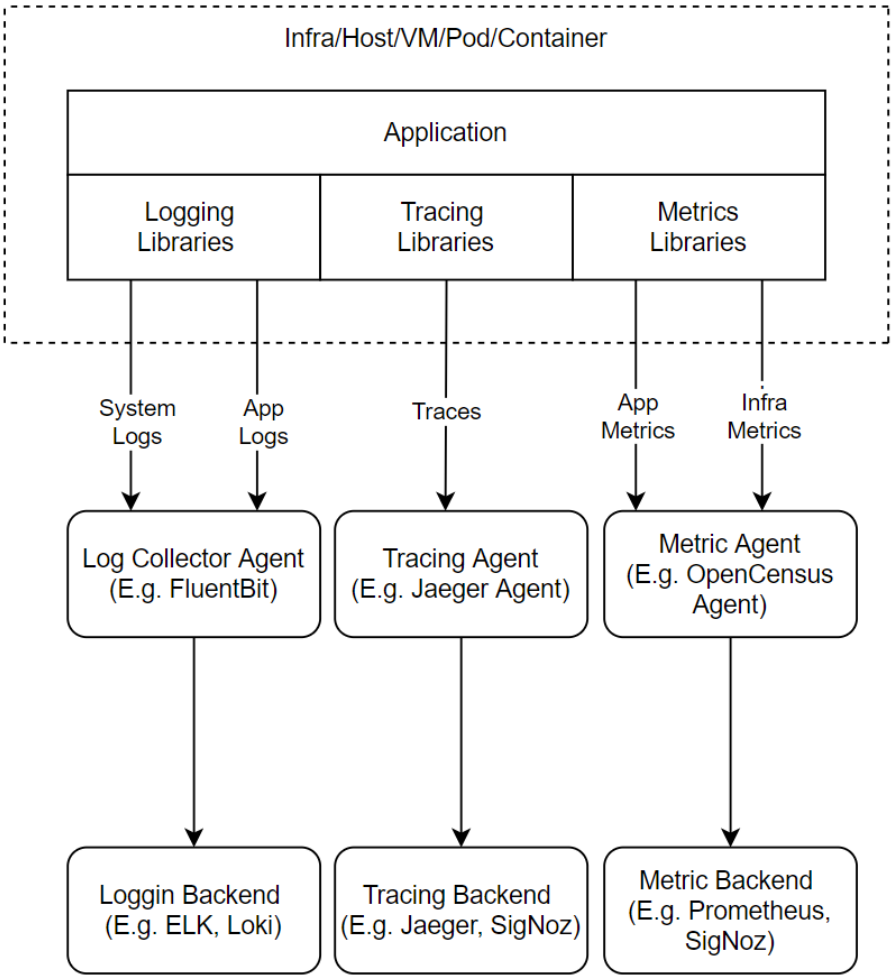

Figure 4.2: Architecture Diagram.

Observability in microservices is an extension of monitoring that provides an internal

understanding system [12]. It includes logging, tracing, and metrics alerting that are helpful for better understanding of the system. Additionally, this data can also be beneficial in debugging applications efficiently. The observability design pattern is beneficial for understanding and scanning microservice applications efficiently. This pattern consists of log aggregation, application metrics, audit logging, distributed tracing, exception tracking, and health check API.

Figure 4.2 is a complete overview of the observability goal we want to achieve in this project. With the help of the instrumentation libraries, we have sent application information to various backend. The objective is to accomplish efficient observability and correlate logs, metrics, and tracing data. By achieving this, we can observe our microservice effectively. Figure 4.2 also demonstrates various open-source tools which we have used as backend.

- Logging - The tools used for logging are ELK stack and Loki.
- Tracing - Jaeger, Elastic APM, ZipKin, and SigNoz are used for tracing.
- Metrics - For monitoring application's metric data we have used Prometheus with Grafana and SigNoz for better visualization.

Chapter 5

Execution and Implementation

5.1 Access Control

Access control will provide functionalities such as Rate-limiting, IP whitelisting/blacklisting, and caching. It will ensure that income requests have some restrictions, which will help us to protect our system from potential vulnerability. Figure 5.1 is a demonstration of the access control layer. We have used .NET core for implementing these feature and we used Ocelot [13] package for implementing API Gateway. These functionalities are discussed in detail below subsections.

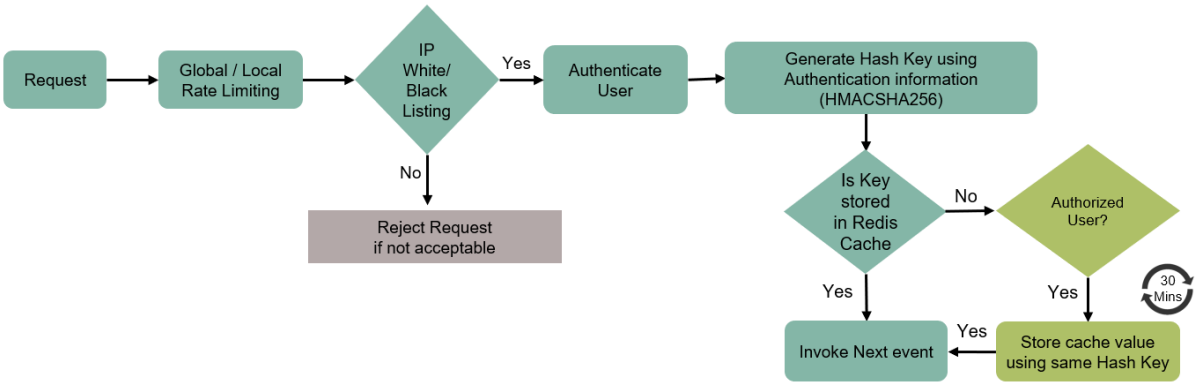

Figure 5.1: Caching mechanism using Redis.

5.1.1 Rate-limiting and IP White/black listing

When any request arrives at the API gateway, we are checking for the maximum limit allowed for that service; for example, if we have set five as the maximum limit, then more than five requests for a single user it not allowed for a particular duration. And

even if so any user tries to gain access to that service, our rate-limiting feature will show a configurable message that 'You are not allowed to request more than five times in 1 second'. Rate-limiting give us an advantage by making our resources available to users, and it also prevents the Denial Of Service (DOS) attack.

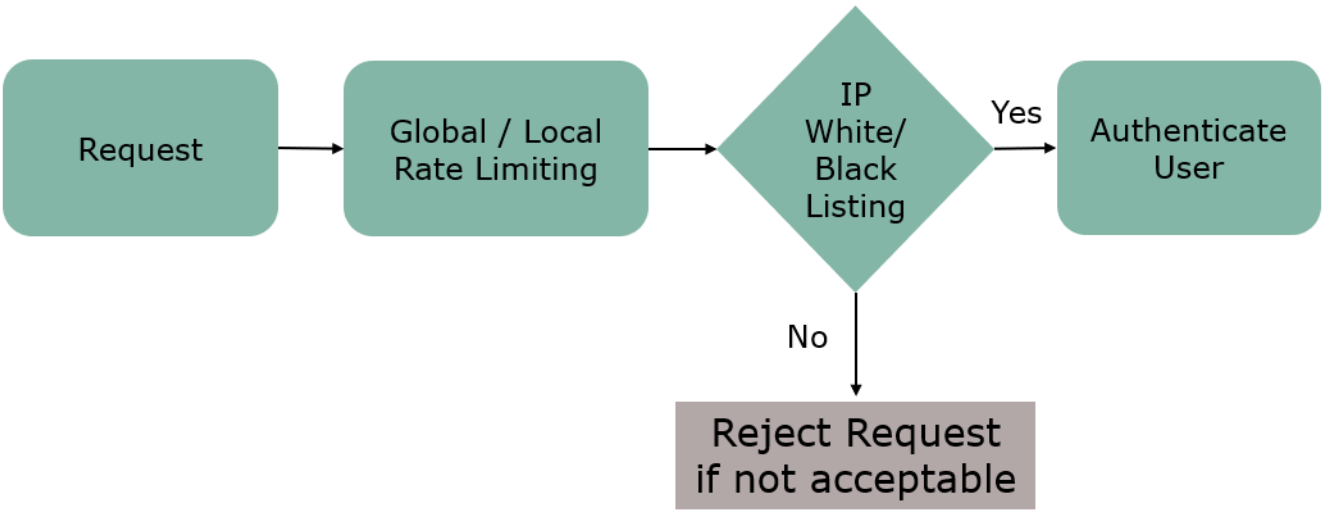

Figure 5.2: Rate-limiting with IP White/black listing.

We have implemented two types of rate-limiting. One is global, and the second one is local. We can use the global rate-limiting configuration to provide the same rate limit to all the endpoints present in microservices. On the other hand, we can use local rate-limiting if want different rate limit for different endpoints.

Figure 5.2 is a workflow diagram of how rate limiting works with IP white/black listing. Once the request passes through rate limiting, we check whether the coming IP address can access microservices. So in the first case, let say the IP address that is requesting resources is not in whitelisting IP's then the application will reject its request. To allow any IP to access resources, we need to add that IP in the configuration variable to access microservices. Once it is done, the given IP can consume the microservices.

5.1.2 Redis Caching

Figure 5.3 is a workflow diagram of how caching works in the application. We have used Redis[14] for storing and processing cache. Once the user is authenticated, we will generate a hash key using the HMACSHA256 algorithm based on this authentication information. After that, we check whether this hash key is already available in cache or

not. In the first scenario, let's say the key is present, then we will forward the request to the next event. And in oppose to this scenario, if the key is not present in the cache, we will check for authorization of the user. If the user is authorized, we will store the cache value in the Redis server using the same hash key that we generated earlier. We

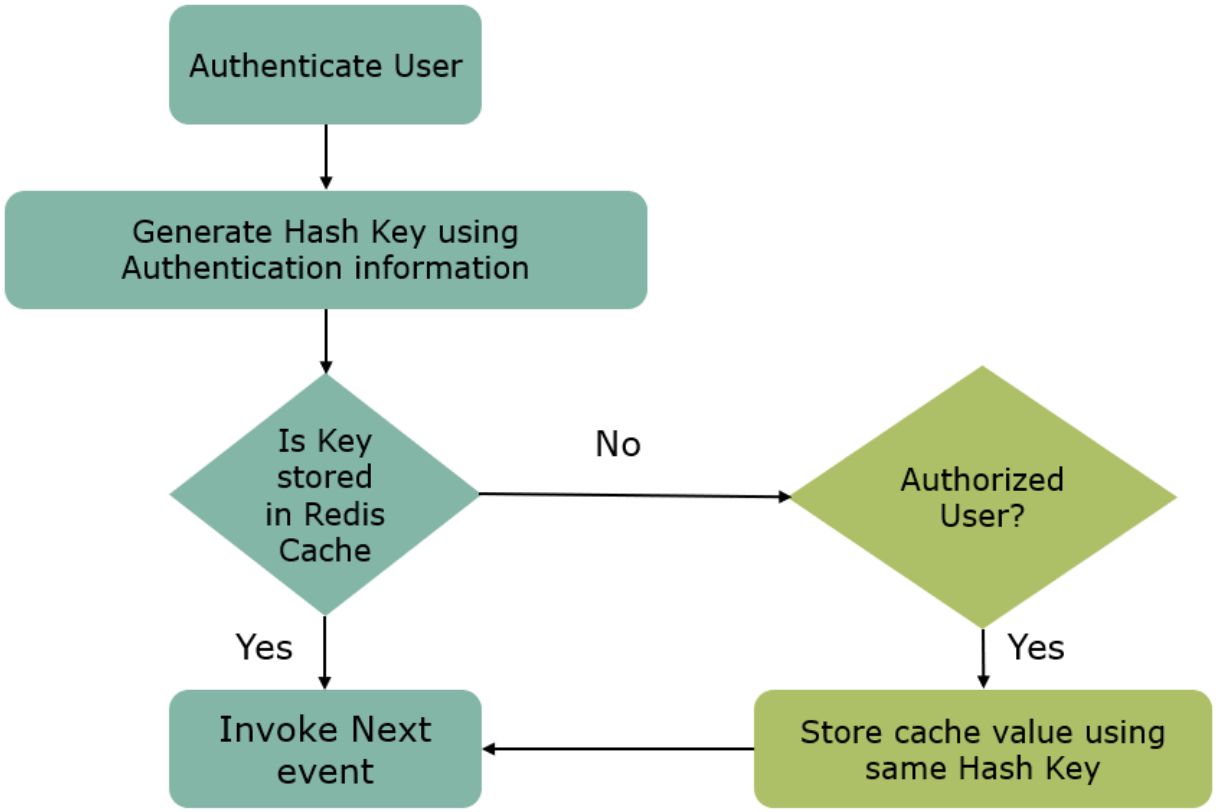

Figure 5.3: Caching mechanism using Redis.

have also implemented the functionality of deleting the cache so that after a particular duration, cache will automatically be deleted from the Redis.

5.1.3 Error Message

We have implemented custom middleware for authentication and authorization at the API gateway level. The primary role of this middleware is to re-route requests according to the check that we have implemented. Figure5.22 shows the custom error message generated when an unauthenticated user tries to access the application. We have generated these error messages for both authentication and authorization.

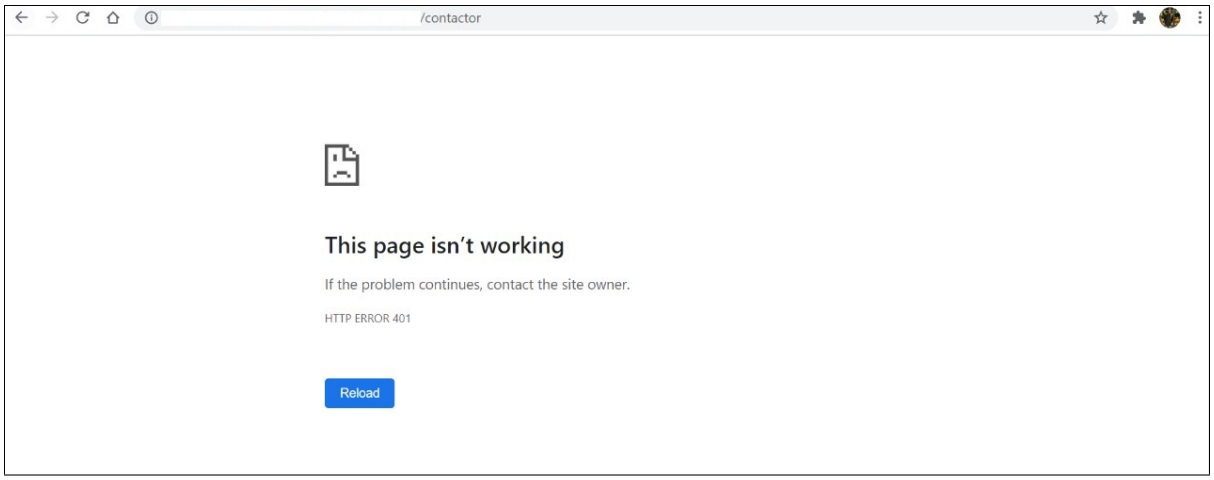

Figure 5.4: Error Message for user

5.2 Logging

Logging plays a crucial role in any application. It will tell you that there is something wrong happened in the system. Generating logs in the console for microservice will not be an efficient solution. We need some visualization tool that can help us visualize this logging data. For this project, we choose ELK for log aggregation and visualization.

5.2.1 ELK Stack

In Figure 5.5, we have conceptual flow of the logging mechanism. We have used Filebeat[15] as a slipper to forward application log data to the ELK cluster. So, once the application generates logs files, our Filebeat container will fetch those logs and deliver them to the ELK cluster.

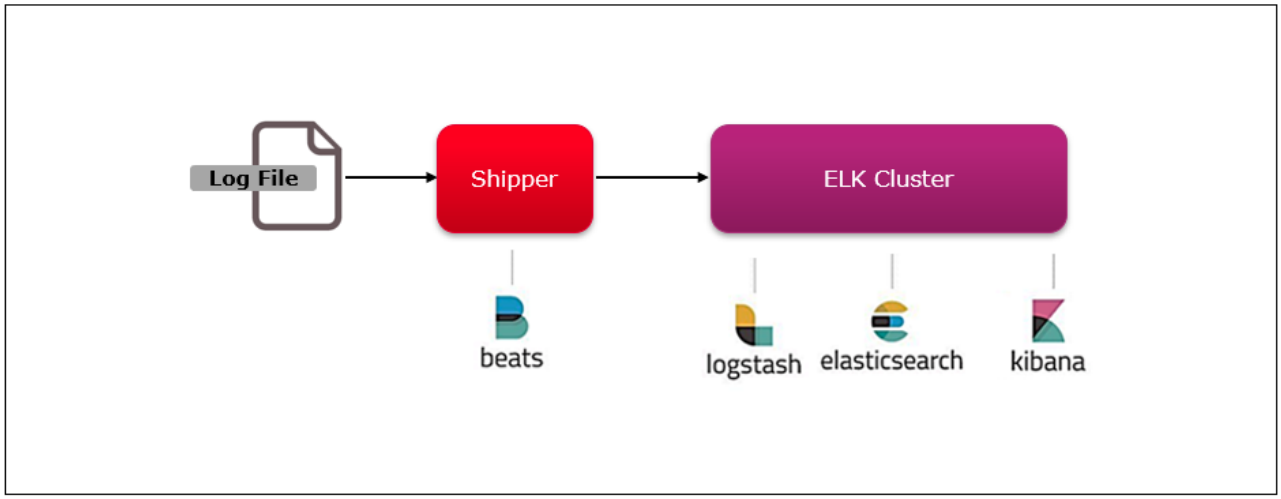

Figure 5.5: Conceptual view of logging mechanism.

ELK [16] combines three technologies: Logstash, Elastic search, and Kibana. Logstash will fetch the data from the outside world, and then it will forward it to Elasticsearch, and Elasticsearch will store this data into it. And then, with the help of the kibana dashboard, we can visualize our logs data.

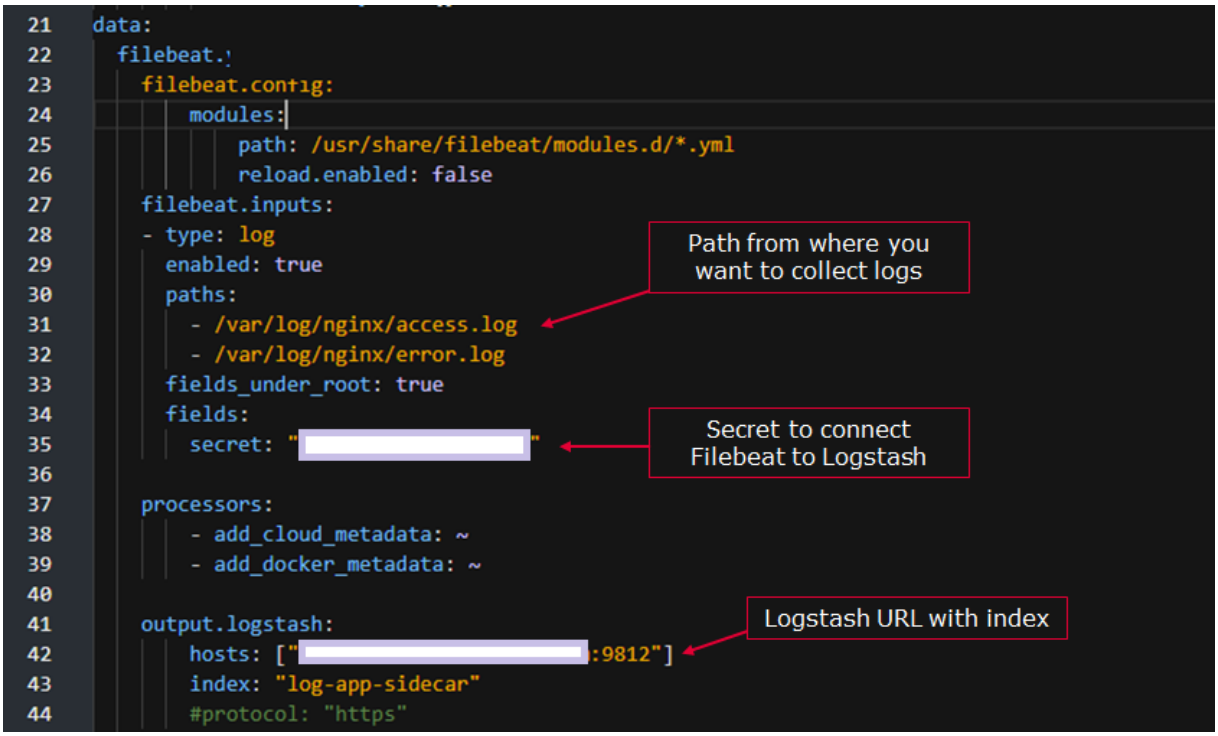

Figure 5.6: ConfigMap of Filebeat.

To fetch the application data using Filebeat, we have done configuration there. We are starting with setting up the ConfigMap for Filebeat. Figure 5.6 is the ConfigMap file of Filebeat, where we have to define various information to connect our Filebeat container to the ELK cluster. 'Paths' is a field where we have to specify which path our application is storing logs. 'secret' is inside 'field' where we have to define the password/secret to connect our Filebeat container to Logstash. And last important thing is the 'URL' of the Logstask instance with its port number and 'index' name by which we can filter our application logs. Once this ConfigMap setup is done, we can configure the Deployment-Config file. This Filebeat container can be added to the application as a sidecar container to fetch application data with less latency.

Figure 5.7 is DeploymentConfig, we also need to add configuration such as volumeMount information to provide persistence volume in the openshift environment and add that ConfigMap configuration to the Filebeat container. Once this configuration is done successfully, we can open the kibana dashboard and visualize our logs there, as shown in

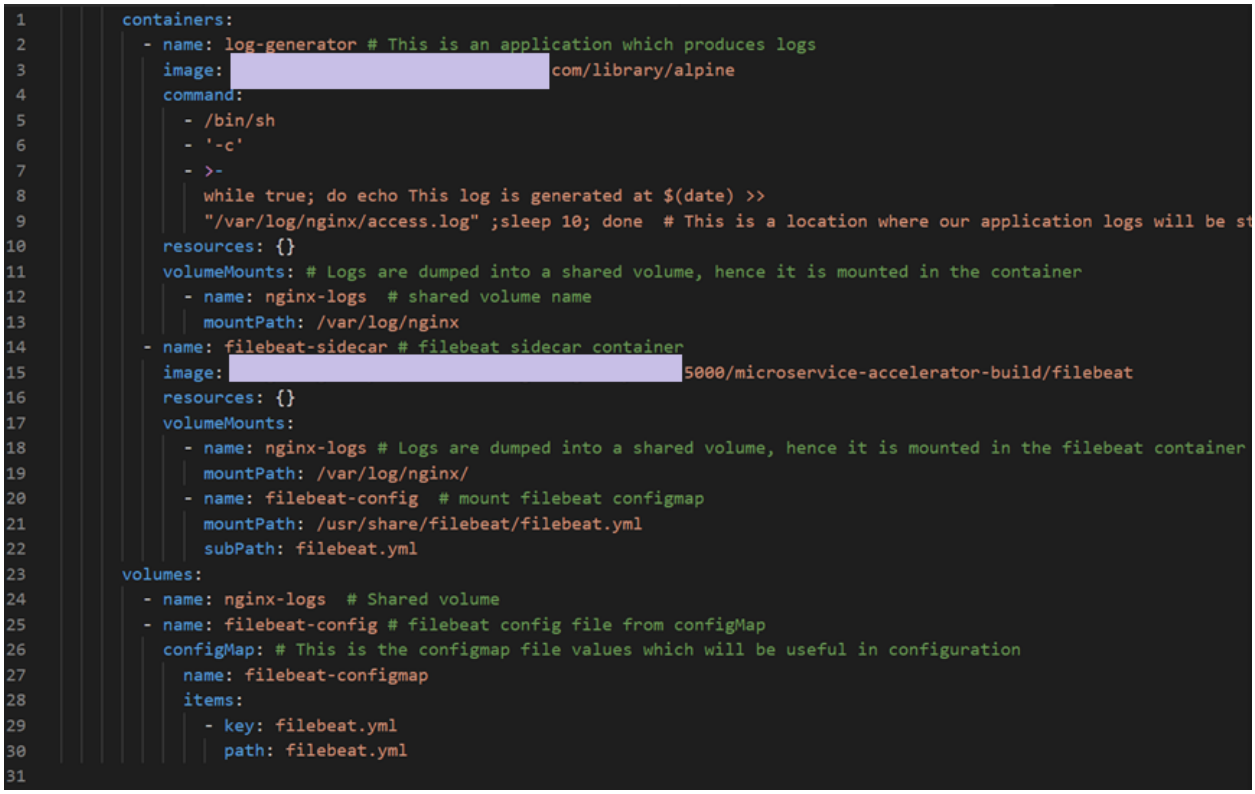

Figure 5.7: DeployementConfig of container.

Figure 5.8. We can observe various information available regarding logs such as timestamp, message, index, and location from where we are fetching logs. There is various other helpful information that we can filter as per our requirement.

ELK stack also provides the feature of APM [17]. This APM is capable enough to fetch and visualize trace information, as shown in Figure 5.9. In addition, APM also provides metrics information related to those traces, including parameters such as latency, throughput, time spent on the span, and total error. We can also correlate our logs and trace data in APM. Having logs, metrics, traces, and alerts in the same places makes ELK stack a great observability tool.

Some of the features of the ELK stack are: 1) It provides a centralized log aggregation mechanism. 2) It is an open-source tool. 3) Real-time data analysis and visualization is possible in ELK. 4) It supports various shippers for shipping log data to ELK. 5) Massive amount of data can be process in short interval. Along with the above-mentioned advantages, ELK has certain areas where it might be improved. Some of which are mentioned as follows: 1) Configuring self-hosted ELK is complex. 2) ELK utilizes high computing resources for easy operations.

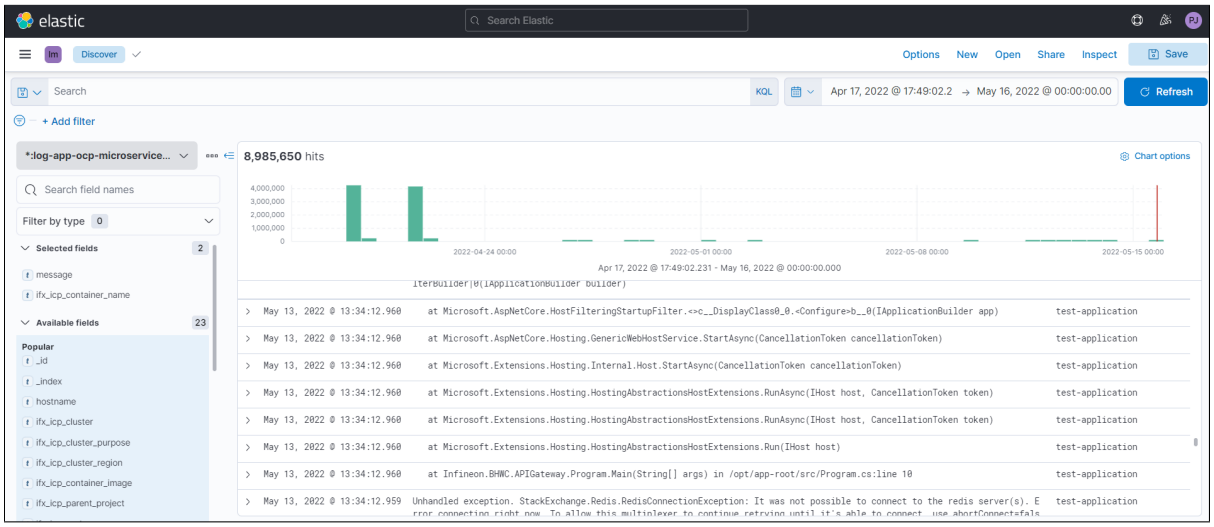

Figure 5.8: Kibana Logging Dashboard.

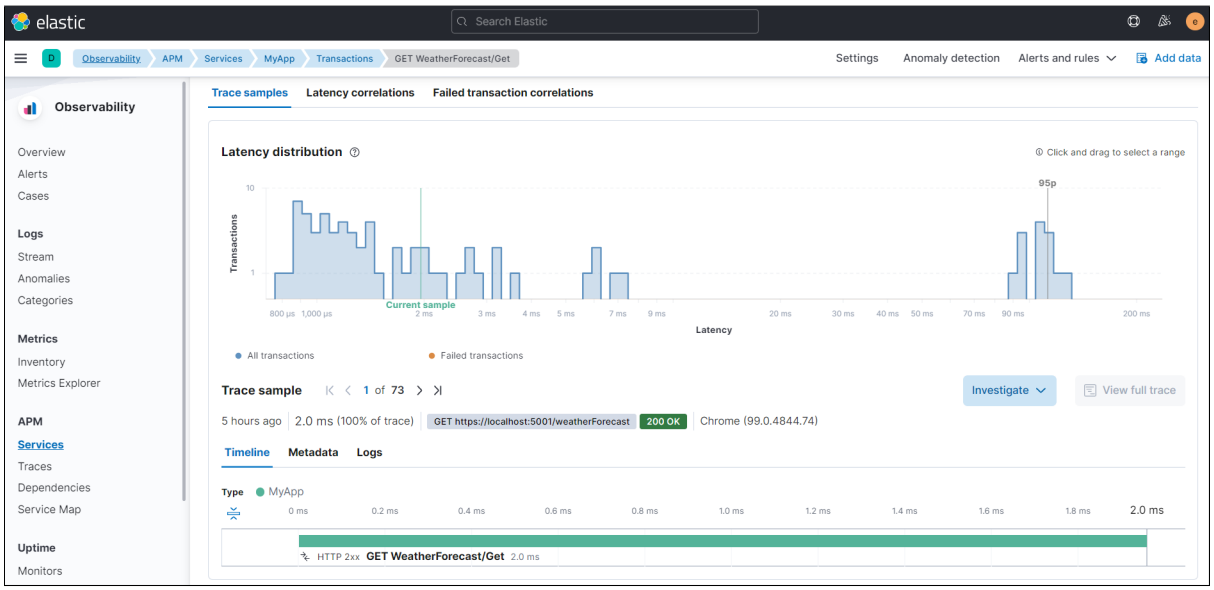

Figure 5.9: ELK Tracing Dashboard

5.2.2 Loki

Loki is a log aggregation system similar to Elasticsearch but it is more effortless to set up and work with more promising functionalities. Loki was initially started by Grafana Labs in 2018 [18]. It is motivated by Prometheus for log aggregation and is highly cost-effective and effortless to operate. Loki sets labels on each log stream instead of indexing logs. Visualization Loki logs can be done using the Grafana dashboard.

Loki has three components – Promtail, Loki, and Grafana. The promtail agent is responsible for locating the target, adding the labels to the incoming log streams, and pushing it to the Loki instance. After connecting Loki with Grafana, we can start visualizing those logs on the dashboard. Figure 5.10 shows deployment of Loki on the Grafana dashboard with sample microservice application data. It shows how logs can be visualized with the help of Grafana. We can also enable real-time log collection to get metrics and log correlation in the dashboard.

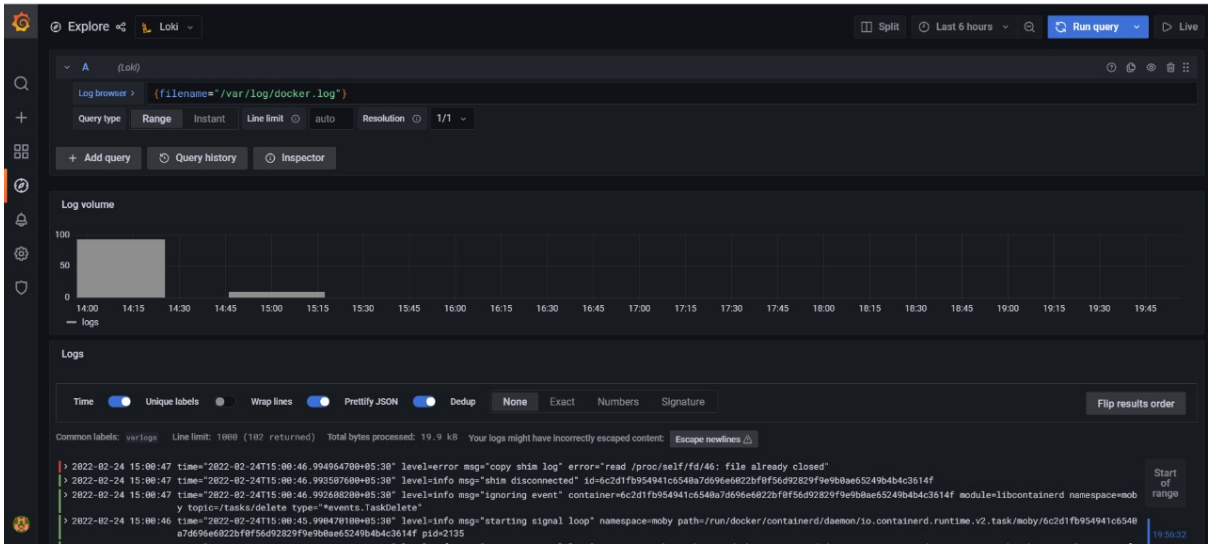

Figure 5.10: Loki Logging Dashboard.

Some of the features of the Loki are: 1) It is very cost-effective. 2) It provides higher scalability. 3) It is easy to plug with popular tools like Kubernetes and Grafana. However, Loki has certain areas where it might be imporved, which are as follows: 1) It is not easy to perform complex queries on Loki. 2) It does not provide a rich dashboard as same as ELK provides.

5.3 Tracing

Having tracing in the application provides you added advantage because logging will tell you that there is an error. Still, we need a tracing mechanism to understand where the problem occurred in the microservice.

5.3.1 Jaeger

Jaeger[19] is one of the great tracing tools with numerous functionalities. Jaeger will help us to understand the execution of requests in the system. In Figure 5.11, we have deployed Jaeger on the OpenShift cluster and tested it with a sample microservice application.

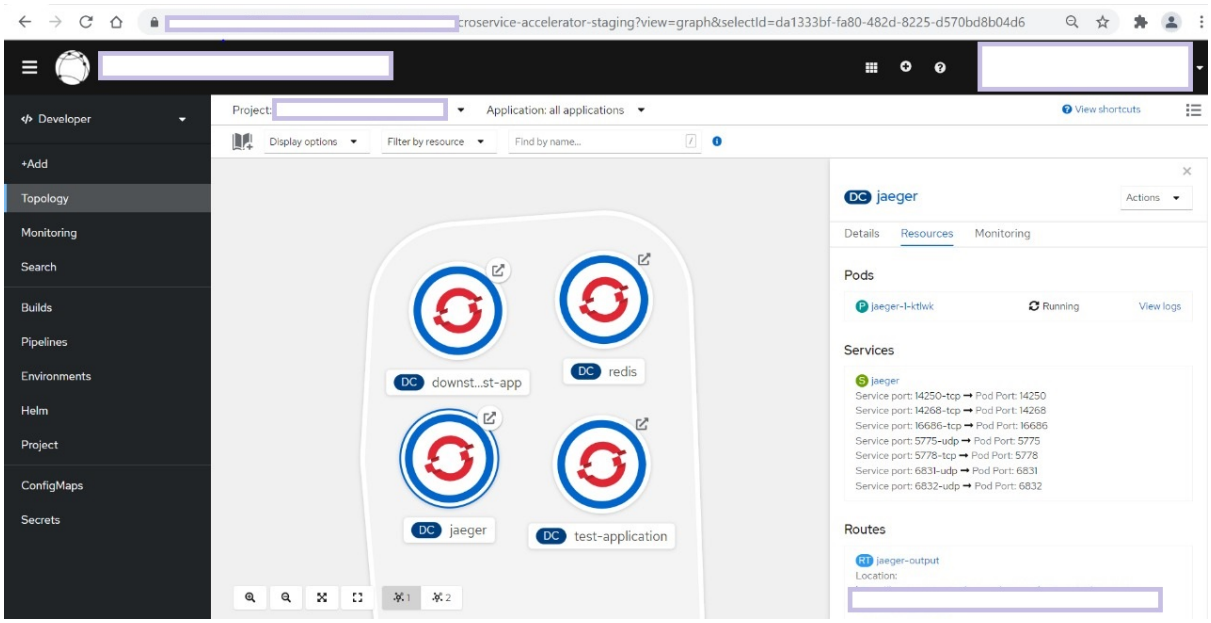

Figure 5.11: Jaeger and test application deployment on OpenShift.

For implementing Jaeger with the OpenShift environment, we have used Docker images of Jaeger. Jaeger Architecture includes Jaeger-Client, Jaeger-Agent, and Jaeger-Collector, Jaeger-Query, Jaeger-Console. Jaeger-Client includes language-specific implementations of the OpenTracing API for distributed tracing. Jaeger-Agent is a network daemon that listens for spans sent over User Datagram Protocol. Jaeger-Collector receives spans and places them in a queue for processing. Jaeger-Query is a service that retrieves traces from storage and hosts a UI to display them. Finally, Jaeger-Console is a user interface that visualization of distributed tracing data.

In Figure 5.12, we can see the homepage of the Jaeger tool. On the left side, we have various filters available. We can filter our traces, such as operation filter by which we can

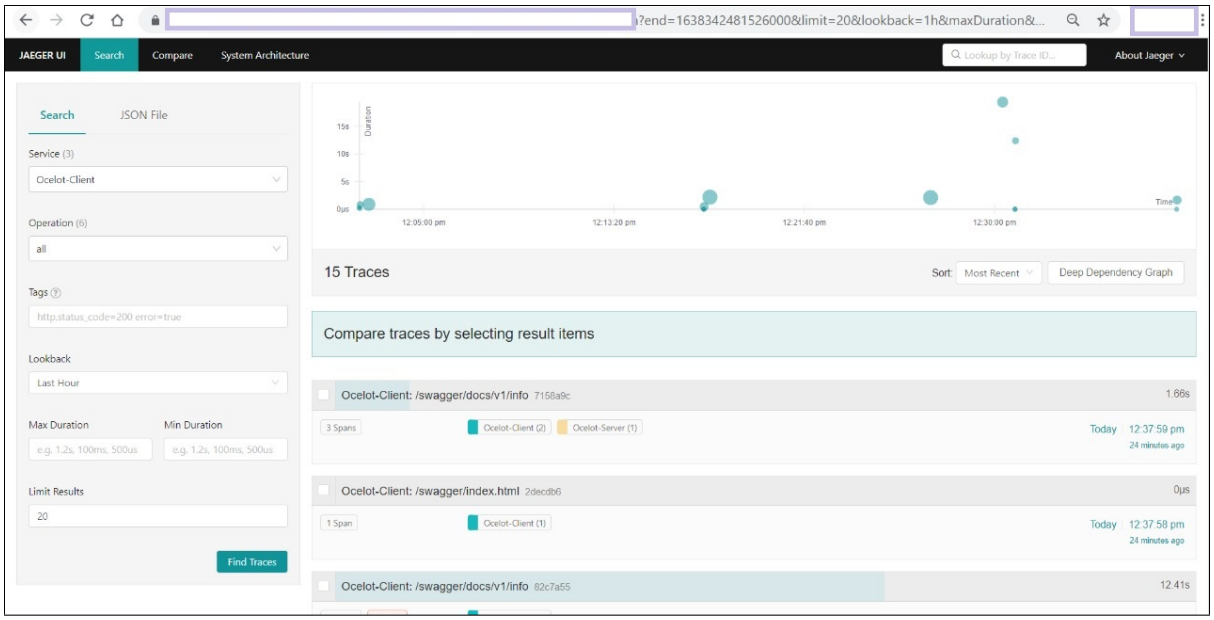

Figure 5.12: Jaeger homepage view.

filter specific operations for which we have created traces and other Tags by which we can filter particular status_code or error. We can also filter the traces by time. On the right side, we have a listing of requests requested by the client.

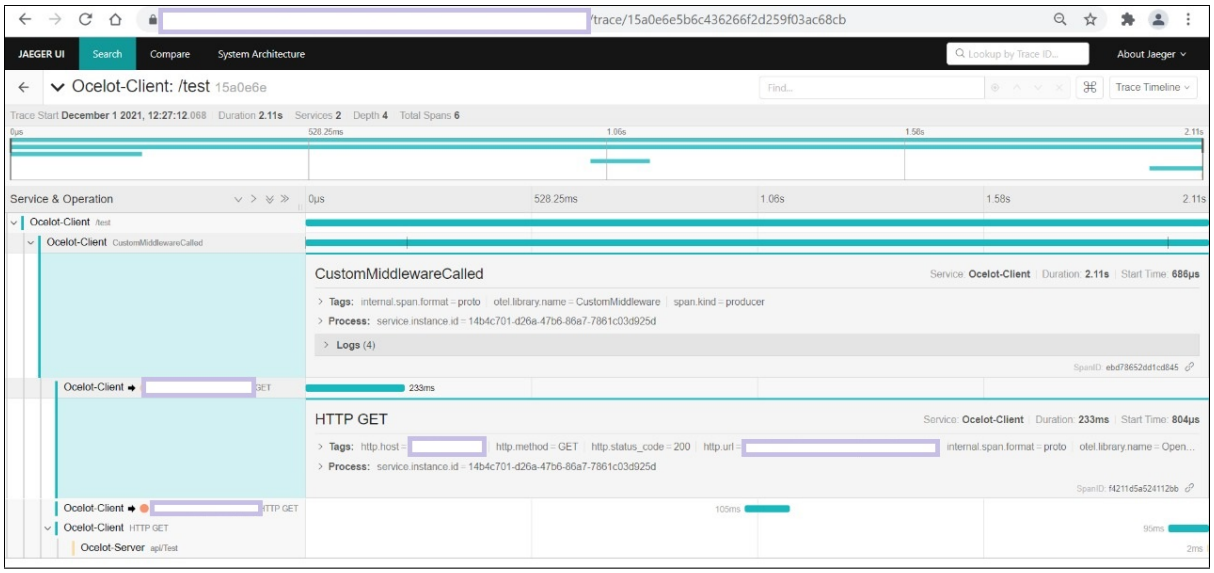

Figure 5.13: Jaeger view of single request.

And once we click on any of the requests, it will expand the view and will us more valuable and detailed information execution flow of that request. In Figure5.13, we can see the entire execution flow of the '/test' request. It contains information such as when this trace is created, how many services it requested, what depth it is called, and how long it takes to complete the request. And also, information on each span is available

in sequence view. This tracing tool will be beneficial for microservice architecture as hundreds and thousands of services communicate with each other, so if a single service failed, we can find it with the help of Jaeger in a much more efficient way. Jaeger also provides Directed Acyclic Graph(DAG)is to visualize how services are communicating.

Some of the features of Jaeger: 1) The Jaeger backend has no single point of failure and scales with the business needs. 2) It can handle billions of spans in a day. 3) Jaeger has instrumentation libraries designed to support the OpenTracing standard. 4) Jaeger supports open-source NoSQL databases as trace storage backends, Cassandra 3.4+ and Elasticsearch 5.x+.

Scope of improvement in Jaeger: 1) Jaeger can have a unified UI for metrics and Traces for more good observability. 2) There are no alert options available as of now in Jaeger. 3) There is no role-based access control available for better team management. 4) Filtering components are limited in Jaeger like we cannot run aggregates filtered on traces.

5.3.2 SigNoz

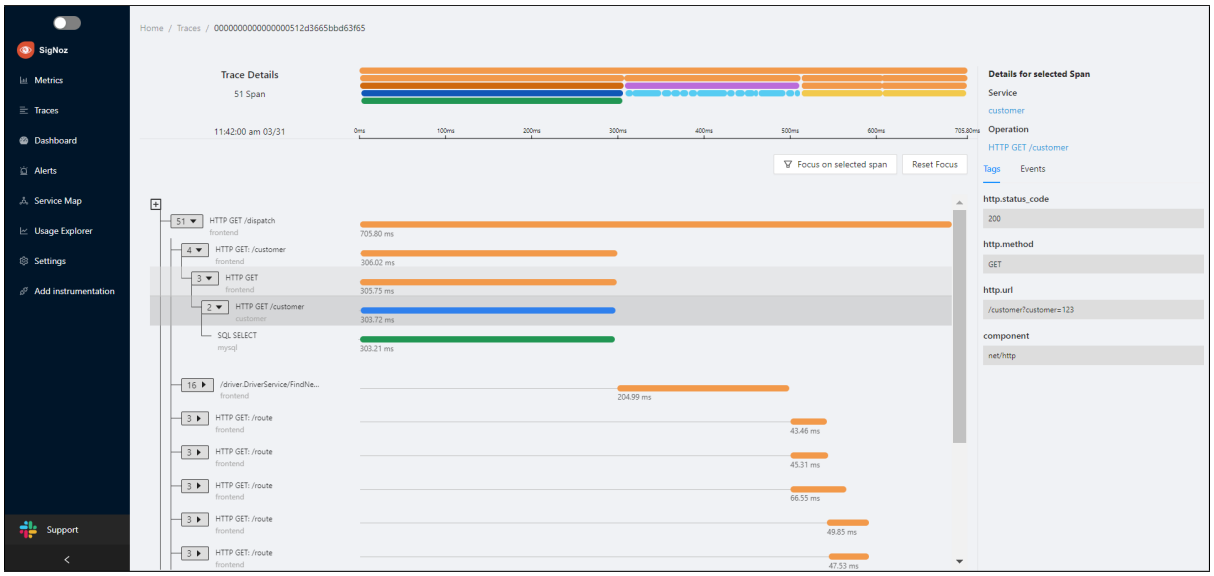

Figure 5.14: Distributed Tracing using SigNoz

SigNoz is an open-source tool for metric and tracing data visualization [20]. It was founded in 2020 to deliver better observability corresponded to existing tools. In Figure 5.14, we have a diagram of trace visualization in SigNoz, and this image illustrates how convenient it is to understand request execution using SigNoz. Furthermore, as SigNoz also provides metrics visualization, we can monitor application performance, as seen in

Figure 5.15. One of the great features of SigNoz is that it allows for a correlation between metrics and tracing data.

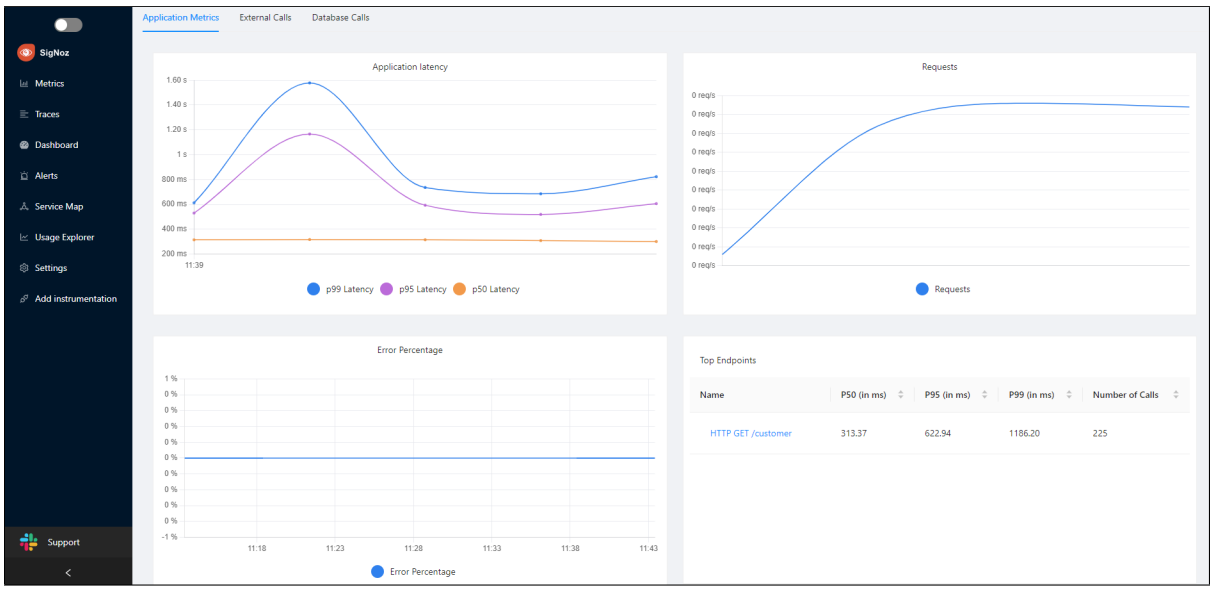

Figure 5.15: Monitoring Application performance using SigNoz.

SigNoz architecture includes OpenTelemetry Collector, ClickHouse, Query Service, and Frontend. OpenTelemetry Collector will receive multiple format data input from various applications and forward it to ClickHouse for storage. Query service fetches data from ClickHouse and processes it before passing it to the front end. And the last component is the frontend, which provides a unified UI for logging and metrics data, including service-map and alert functionality.

Some of the features of SigNoz: 1) Provide correlation between metrics and tracing data. 2) Advance filtering option available for data filtering. 3) It also includes a feature of alerting and service mapping. Scope of improvement in SigNoz: 1) Tool is very young in the industry. 2) It does not provide support of logs as of now (it is there in their future road map)

5.3.3 ZipKin

Zipkin is an open-source distributed tracing system to collect trace information [21]. Zipkin assists us to locate precisely where a request to the application has failed or spent a long time. We can instrument our application tracing using the Zipkin client library. Figure 5.16 shows various request information on the homepage of the Zipkin tool.

Zipkin architecture includes collector, storage, query-service, and web UI. The col-

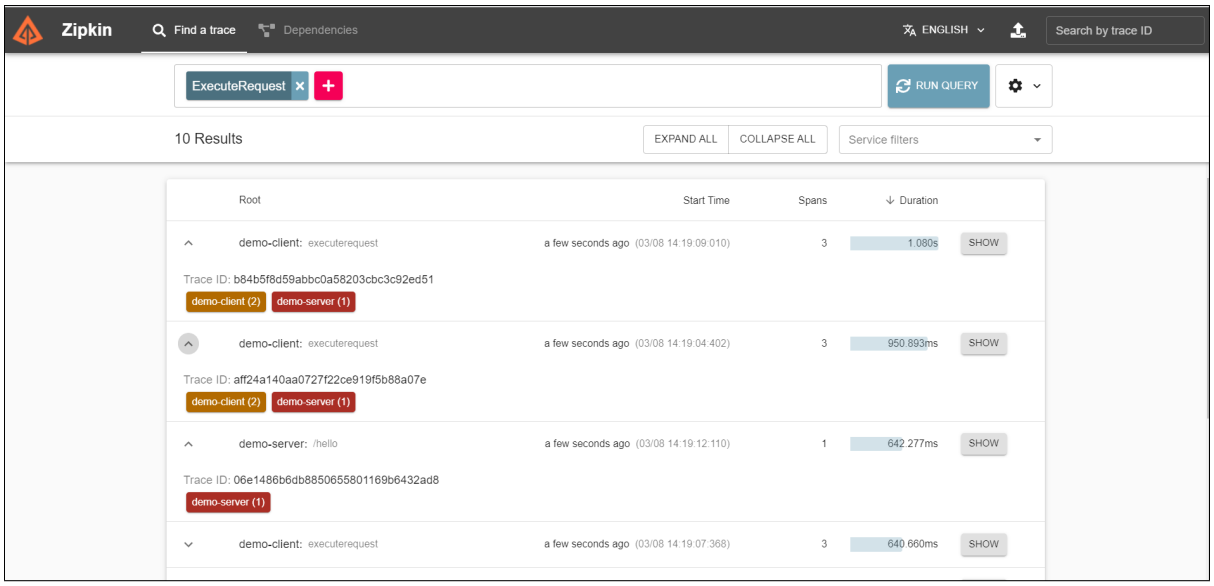

Figure 5.16: Distributed Tracing using Zipkin.

lector is used for validating, storing and indexing incoming trace data. As storage, it supports Cassandra, ElasticSearch, and MySQL. Query-service provides API for finding and retrieving traces to web UI. And using web UI, we can visualize our trace on the frontend

Some of the features of Zipkin: 1) Zipkin supports multiple storage backends. 2) It has large community support 3) Zipkin provide service mapping functionality 4) It is a mature project. Scope of improvement in Zipkin: 1) It lacks support for some client libraries.

5.4 Metrics

Metrics play a crucial role in tracking application performance. It will give us helpful application insight such as CPU usage, Services rate, and many more. We can utilize these insights to set up a better infrastructure for our application.

5.4.1 Prometheus

Prometheus is an open-source tool that is used to view metrics information. As a part of this project, we have used service-monitoring [22] of the OpenShift environment, which is based on the Prometheus rule only. Service monitoring provides a very interactive UI design where we can visualize metrics information and events of the application. Figure5.17 shows metrics information related to Pod CPU usage.

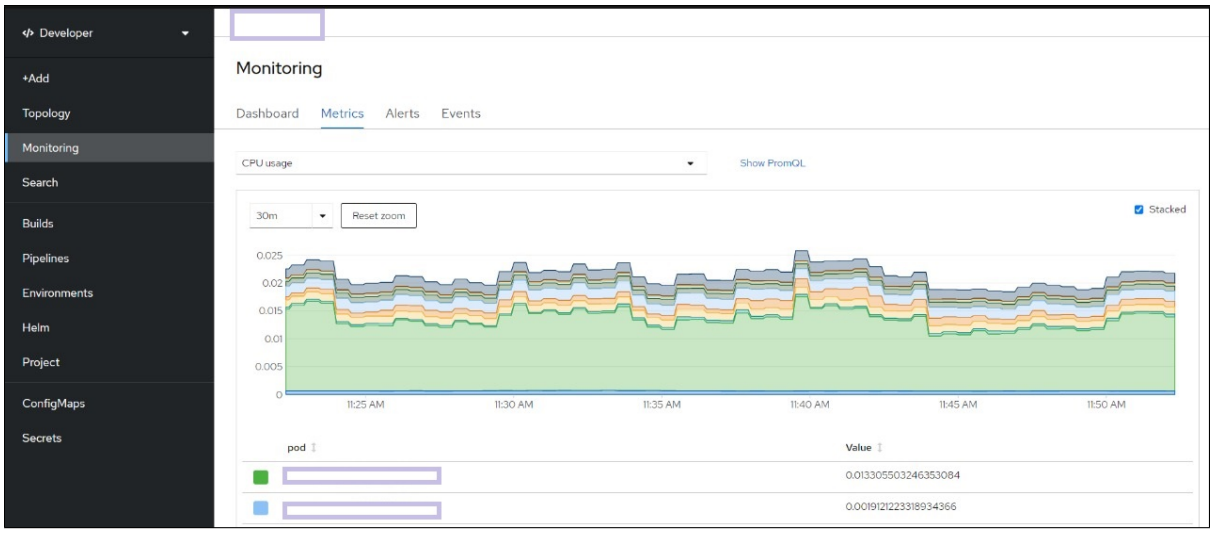

Figure 5.17: Service Monitoring of OpenShift.

Service monitoring also provides feature for visualization custom metrics data. Insight metrics tab, we have a field for searching our custom metrics information. In Figure5.18, we can see the custom metrics that we have created for this project with visualization.

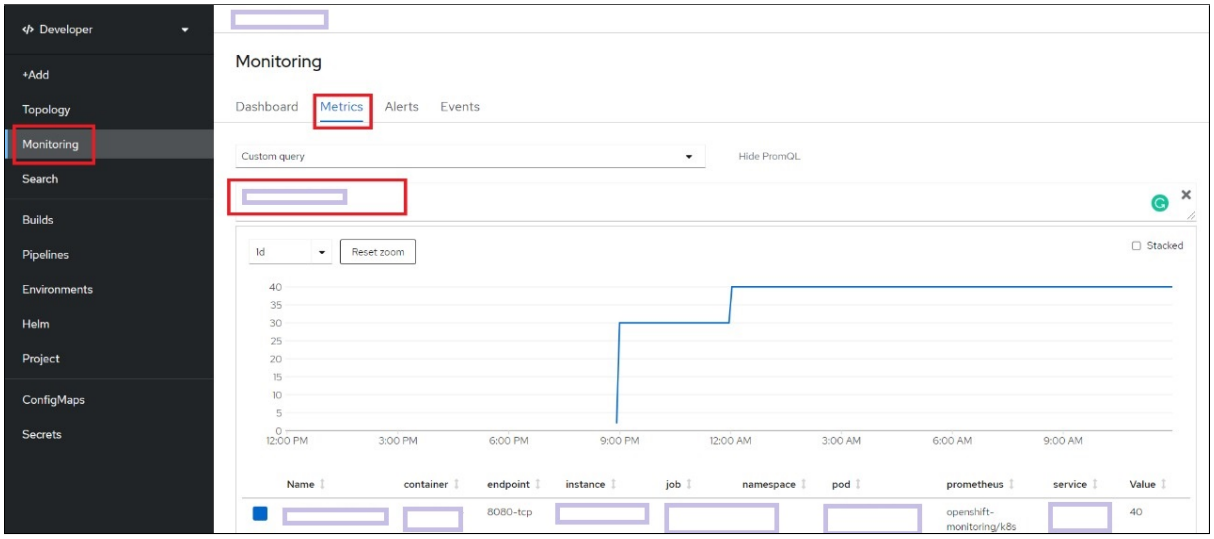

Figure 5.18: Visualizing custom metrics data of application in Service Monitoring.

Grafana [23] is a great data visualization tool. Having grafana with prometheus is similar to having ice on the cake. Figure 5.19 demonstrates application metrics visualization in the grafana dashboard. The reason why we want grafana along with prometheus is that prometheus provides very limited visualization capability; that's why we need grafana for out of the box visualization.

Grafana, on the other hand, has various other integration options for logging, metrics, and trace data. We can use grafana for better observability in the system.

Figure 5.19: Application performance monitoring in Grafana

Some of the features of Prometheus and Grafana: 1) Prometheus have effective query language for fetching and analyzing metrics data. 2) Grafana supports various integration options. 3) Grafana provide excellent data visualization feature. 4) We can use grafana for alerting on fly application failure information. 5) Grafana supports data grouping for more useful data visualization.

Scope of improvement in Prometheus and Grafana: 1) Prometheus is lacking in providing good visualization UI. 2) Prometheus doesn't have a long term storage option. 3) Grafana does not store data. 4) Visualization libraries are limited in grafana.

5.5 Alerting

Alerts play a crucial role in the application. Using alerts, we can get on-fly information regarding system failure. Openshift will trigger alerts when any metrics information hits a certain threshold. We have the option to configure to get messages via Webex or mail.

Figure 5.20 is alerting configuration where we can configure our alerts. We have options to configure multiple alerts into single groups and multiple groups as per our requirements. Inside the 'rules' field, we can define our rule by which Openshift can trigger an alert. We have the option to configure the severity, what message we want to see, the expression on which the alert used to be triggered, and the alert name. Here in the example Figure 5.20, we define an expression where if the rate of sampleapp_ticks_total increases more than 0.0002 in less than a minute, it will generate an alert. We can customize this expression based on our requirements.

```yaml
apiVersion: monitoring.coreos.com/v1
kind: PrometheusRule
metadata:
  name: sailor-alerts
  namespace: my-namespace
spec:
  groups:
    - name: sailor-alerts
      rules:
        - alert: TooManyFailedRequests
          expr: rate(sampleapp_ticks_total[1m]) > 0.0002
          labels:
            severity: critical
```

Figure 5.20: Alerting configuration

Figure 5.21 is the message that we have received on the Webex group, which includes various useful information such as site name, cluster name, project name, the current status of the alert, severity, what time it is generated, and what is a type of error. These information will be very useful to debug application in much faster way. We have done these setup for our project in Openshift environment.

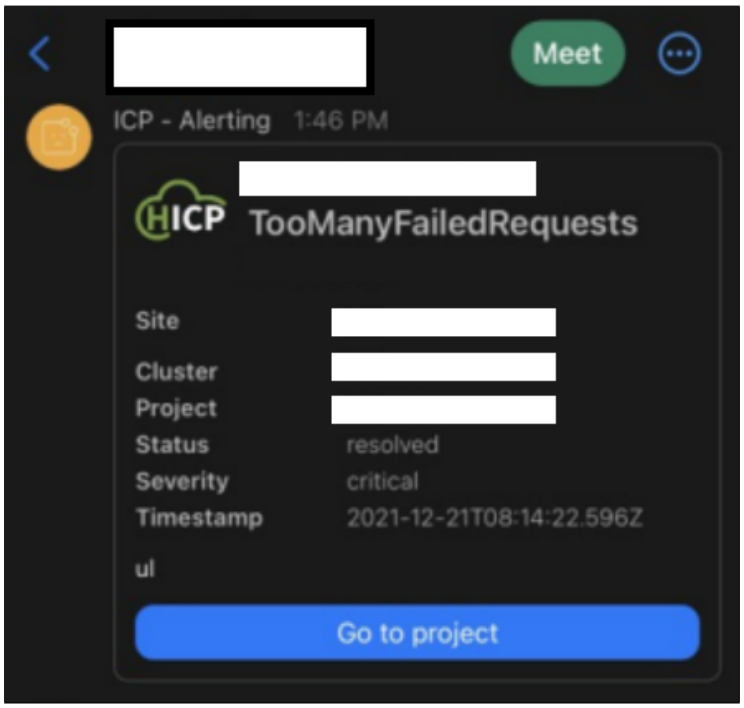

Figure 5.21: Alert on Webex

5.6 Dynamic Routing

In the ocelot package, we have the option to add service information. Still, it is tightly coupled with the configuration file because if we update single information, we need to re-deploy the configuration again. For that reason, we need dynamic routing in a microservice architecture. One of the challenging things in a microservice architecture is to enable dynamic routing so that the application self-register as and when it is up and automatically de-register if it is down. For this purpose, we have used Consul [24] developed by HashiCrop for enabling dynamic routing. Consul provides distributed key-value storage for application configuration, and it provides service mesh for secured service segmentation in any cloud environment. [25].

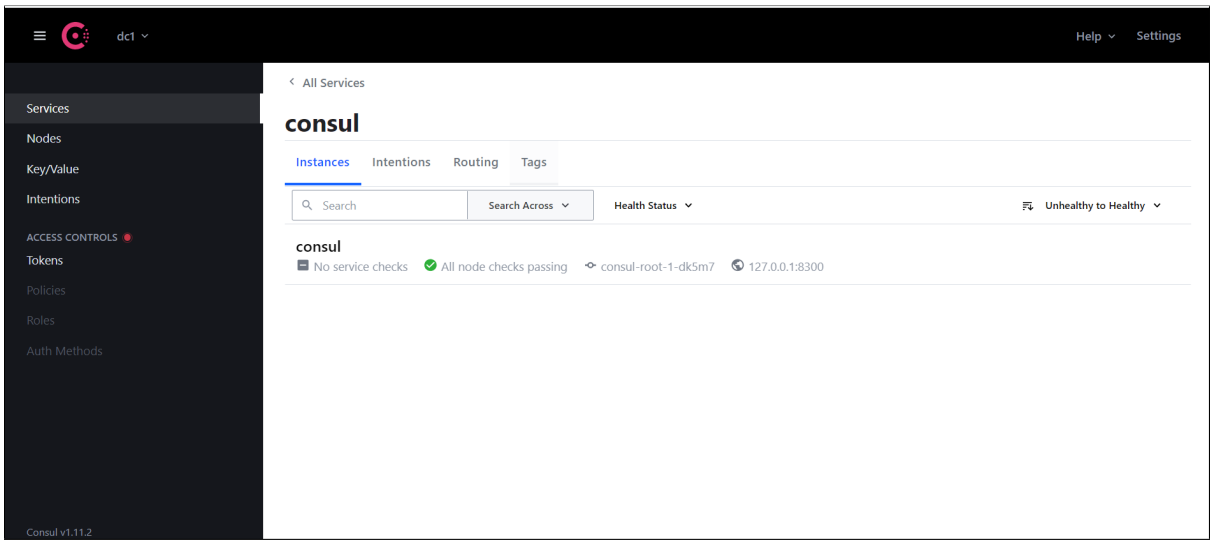

Figure 5.22: Consul dashboard for dynamic configuration

Figure 5.22 is the home page of the Consul, where we will have the lists of various services in the application. As we increase the instance count, it will dynamically bind that service to the Consul. It will automatically de-register services if they are not up and re-register as and when they are up. Another reason why service discovery (Consul) plays an important role is because we can have multiple instances of a single application running into the same cluster in cloud applications. It will work as a load balancer in the architecture. The health check is also provided by Consul so that we don't need any external monitoring servers for a health check. Consul is a very crucial tool in cloud-based applications.

Chapter 6

Future Work and Conclusion

As data increases, we need to increase resources as per needs, and typical monolithic architecture cannot handle scalability at this enormous scale. The main disadvantage of monolithic is that we cannot achieve rapid application development at a gigantic scale.

In this paper, we have discussed microservice architecture and challenges within architecture. We worked on an observability design pattern that addressed issues related to observability in microservices. We have investigated and implemented various open-source tools such as Prometheus, Grafana, SigNoz, Jaeger, Zipkin, Loki, and ELK stack. Our study will help others to understand the capabilities of these tools. So in this project, we have the functionality of API Gateway extension (Rate-Limiting, Caching, and IP whitelisting/blacklisting), Monitoring and Observability (Logging, Tracing, Metrics, Notification Alert, and Standard Visualization) as a part of infrastructure as a code.

Here we want to conclude that there would never be a scenario where we can have one fixed tool for observability in a microservice architecture. Many tools come and can replace existing tools based on market performance. So we should always look for an updated tool in a microservice architecture.

Bibliography

[1] M. Fazio, A. Celesti, R. Ranjan, C. Liu, L. Chen, and M. Villari, "Open issues in scheduling microservices in the cloud," *IEEE Cloud Computing*, vol. 3, no. 5, pp. 81–88, 2016.

[2] "Digital 2021 april global statshot report, https://datareportal.com/reports/digital-2021-april-global-statshot," 2022.

[3] A. Balalaie, A. Heydarnoori, and P. Jamshidi, "Microservices architecture enables devops: Migration to a cloud-native architecture," *Ieee Software*, vol. 33, no. 3, pp. 42–52, 2016.

[4] P. Rodgers, "Service-oriented development on netkernel-patterns processes & products to reduce system complexity web services edge 2005 east: Cs-3"," *CloudComputingExpo*, 2005.

[5] R. Chen, S. Li, and Z. Li, "From monolith to microservices: A dataflow-driven approach," in *2017 24th Asia-Pacific Software Engineering Conference (APSEC)*, pp. 466–475, IEEE, 2017.

[6] O. Al-Debagy and P. Martinek, "A comparative review of microservices and monolithic architectures," in *2018 IEEE 18th International Symposium on Computational Intelligence and Informatics (CINTI)*, pp. 000149–000154, IEEE, 2018.

[7] N. Dragoni, S. Giallorenzo, A. L. Lafuente, M. Mazzara, F. Montesi, R. Mustafin, and L. Safina, "Microservices: yesterday, today, and tomorrow," *Present and ulterior software engineering*, pp. 195–216, 2017.

[8] M. Villamizar, O. Garces, L. Ochoa, H. Castro, L. Salamanca, M. Verano, R. Casallas, S. Gil, C. Valencia, A. Zambrano, *et al.*, "Infrastructure cost comparison of run-

ning web applications in the cloud using aws lambda and monolithic and microservice architectures," in *2016 16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, pp. 179–182, IEEE, 2016.

[9] M. P. de Sá, "Emergent microservices in emergent ecosystems," in *2020 IEEE/ACM 13th International Conference on Utility and Cloud Computing (UCC)*, pp. 449–450, IEEE, 2020.

[10] A. Ciuffoletti, "Automated deployment of a microservice-based monitoring infrastructure," *Procedia Computer Science*, vol. 68, pp. 163–172, 2015.

[11] B. Mayer and R. Weinreich, "A dashboard for microservice monitoring and management," in *2017 IEEE International Conference on Software Architecture Workshops (ICSAW)*, pp. 66–69, IEEE, 2017.

[12] N. Marie-Magdelaine, T. Ahmed, and G. Astruc-Amato, "Demonstration of an observability framework for cloud native microservices," in *2019 IFIP/IEEE Symposium on Integrated Network and Service Management (IM)*, pp. 722–724, IEEE, 2019.

[13] "Ocelot, https://ocelot.readthedocs.io/en/latest/introduction/gettingstarted.html," 2022.

[14] "Redis, https://redis.io/," 2022.

[15] "Filebeat overview, https://www.elastic.co/guide/en/beats/filebeat/current/filebeat-overview.html," 2022.

[16] "Elastic stack: Elasticsearch, kibana, beats &; logstash," 2022.

[17] "Elastic apm, https://www.elastic.co/observability/application-performance-monitoring," 2022.

[18] "Grafana loki, https://grafana.com/oss/loki," 2022.

[19] "Open source, end-to-end distributed tracing by jaeger, https://www.jaegertracing.io/," 2022.

[20] "Signoz, 'https://signoz.io/'," 2022.

[21] "Zipkin, https://zipkin.io/," 2022.

[22] "Service monitoring, prometheus, https://docs.openshift.com/container-platform/4.7/monitoring/managing-metrics.html," 2022.

[23] "Grafana, https://grafana.com/," 2022.

[24] "Consul, service discovery, https://www.consul.io/," 2022.

[25] "Consul, wiki, https://en.wikipedia.org/wiki/consul$_(software)$,$''$ 2022.

Appendix A

- https://docs.microsoft.com/en-us/azure/architecture/guide/architecture-styles/microservices
- https://docs.openshift.com/
- https://redis.io/
- https://www.jaegertracing.io/
- https://prometheus.io/
- https://www.elastic.co/what-is/elk-stack
- https://www.elastic.co/beats/filebeat
- https://medium.com/swlh/building-net-core-api-gateway-with-ocelot-6302c2b3ffc5
- https://ocelot.readthedocs.io/en/latest/features/configuration.html
- https://redis.io/topics/config
- https://www.geeksforgeeks.org/difference-between-iaas-paas-and-saas/
- https://www.blogofpi.com/restrict-ip-address-asp-net-core-web-api/
- https://cloud.netapp.com/blog/cvo-blg-5-red-hat-openshift-benefits-you-didnt-know-about
- https://logz.io/blog/filebeat-vs-logstash/
- https://docs.openshift.com/container-platform/4.6/virt/logging_events_monitoring/virt-openshift-cluster-monitoring.html
- https://medium.com/opentelemetry/deploying-the-opentelemetry-collector-on-kubernetes-2256eca569c9

Jay - Aggregation and Observability in Microservice Accelerator

ORIGINALITY REPORT

8%	6%	5%	1%
SIMILARITY INDEX	INTERNET SOURCES	PUBLICATIONS	STUDENT PAPERS

PRIMARY SOURCES

1	s3-eu-west-1.amazonaws.com Internet Source	1%
2	export.arxiv.org Internet Source	1%
3	dzone.com Internet Source	1%
4	Muhammad Waseem, Peng Liang, Mojtaba Shahin, Amleto Di Salle, Gastón Márquez. "Design, monitoring, and testing of microservices systems: The practitioners' perspective", Journal of Systems and Software, 2021 Publication	<1%
5	Nebrass Lamouchi. "Chapter 14 Flying All Over the Sky with Quarkus and Kubernetes", Springer Science and Business Media LLC, 2021 Publication	<1%
6	slides.com Internet Source	<1%

7	Submitted to Napier University Student Paper	<1 %
8	www.jaegertracing.io Internet Source	<1 %
9	www.economyinformatics.ase.ro Internet Source	<1 %
10	thenextweb.com Internet Source	<1 %
11	ieeexplore.ieee.org Internet Source	<1 %
12	dokumen.pub Internet Source	<1 %
13	Submitted to Colorado State University Fort Collins Student Paper	<1 %
14	Submitted to Free University of Bolzano Student Paper	<1 %
15	Submitted to Macquarie University Student Paper	<1 %
16	scholarsbank.uoregon.edu Internet Source	<1 %
17	aaltodoc.aalto.fi Internet Source	<1 %
18	ns2.thinkmind.org	

	Internet Source	<1%
19	tel.archives-ouvertes.fr Internet Source	<1%
20	"Monitoring High Throughput Distributed System using Statistical Data Analysis", International Journal of Recent Technology and Engineering, 2020 Publication	<1%
21	Mario Villamizar, Oscar Garces, Lina Ochoa, Harold Castro et al. "Infrastructure Cost Comparison of Running Web Applications in the Cloud Using AWS Lambda and Monolithic and Microservice Architectures", 2016 16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid), 2016 Publication	<1%
22	helda.helsinki.fi Internet Source	<1%
23	www.elastic.co Internet Source	<1%
24	"Cloud Computing and Services Science", Springer Science and Business Media LLC, 2020 Publication	<1%

25	Omar Al-Debagy, Peter Martinek. "A Comparative Review of Microservices and Monolithic Architectures", 2018 IEEE 18th International Symposium on Computational Intelligence and Informatics (CINTI), 2018 Publication	<1%
26	Shuaiyu Wang, Yinsheng Li. "A Creditworthy Resources Sharing Platform Based on Microservice", 5th International Conference on Crowd Science and Engineering, 2021 Publication	<1%

Exclude quotes	On	Exclude matches	Off
Exclude bibliography	On		