Exploring Tracing in Microservice Applications: Leveraging Zipkin for Enhanced Observability

Okpako Marvis, Olanriwaju Babatunde, Eguavoen Victor Osasu

Departmant of Computer Science, Wellspring University, Nigeria

DOI: https://doi.org/10.51244/IJRSI.2024.11110030

Received: 22 October 2024; Accepted: 03 November 2024; Published: 04 December 2024

ABSTRACT

Microservices architecture has gained prominence due to its scalability and modularity, but its distributed nature complicates observability, performance monitoring, and troubleshooting. This study explores the integration of Zipkin, an open-source distributed tracing tool, into microservice architectures to enhance observability. The primary objective is to investigate how Zipkin can be used to trace service interactions, identify latency issues, and optimize system performance in microservices. The research methodology involves implementing Zipkin in a Job Microservice Application comprising three core services: Job, Company, and Review Microservices, built using Java Spring Boot, PostgreSQL, and Docker. The services are integrated with Zipkin to track request flows and analyze system behavior. Performance and functionality tests were conducted using REST APIs to evaluate the effectiveness of Zipkin's tracing capabilities. Results show that Zipkin significantly improves system observability, enabling developers to pinpoint performance bottlenecks and resolve issues more efficiently. The integration of distributed tracing reduced debugging time and enhanced performance monitoring across services. In conclusion, Zipkin provides valuable insights into service interactions in microservice environments, making it an effective tool for optimizing performance and troubleshooting. The study recommends further exploration of advanced sampling strategies and integration with other monitoring tools to enhance scalability in large-scale microservices systems.

Keywords: Microservices, Distributed Tracing, Zipkin, Observability, Performance Optimization.

INTRODUCTION

Microservices architecture has transformed modern software development by offering scalability, flexibility, and resilience. Unlike monolithic systems deployed technology accelerated learning systems and innovations for effective learning outcomes (Eguavoen & Nwelih, 2023), microservices are composed of loosely coupled, independently deployable services, each responsible for specific business functions. However, this distributed nature introduces challenges in monitoring, troubleshooting, and understanding service interactions (Dragoni et al., 2017). Traditional monitoring tools provide limited visibility into microservices' internal interactions, making it difficult to track requests and identify performance issues. Observability, the ability to infer a system's internal state based on its outputs, has emerged as a critical concept in managing microservices architectures. While traditional monitoring tools focus on metrics like CPU usage and request rates, they fail to capture the complex asynchronous communication between services. Distributed tracing has become a crucial tool for enhancing observability by tracking the flow of requests through multiple services, thereby enabling developers to monitor latency, identify bottlenecks, and troubleshoot errors more effectively (Fitzpatrick, 2023).

The Job, Company, and Review Microservices were selected as representative services because they simulate common, real-world application components that interact frequently and have interconnected data dependencies. In a typical job portal scenario, job listings must be associated with specific companies, and user-generated reviews provide feedback on both jobs and companies. These dependencies reflect the complexity of managing inter-service communication, data consistency, and real-time updates across multiple services, making this microservice architecture an effective model for testing observability with Zipkin in real-world scenarios.

This research focuses on the integration of Zipkin, an open-source distributed tracing tool, into microservice

ISSN No. 2321-2705 | DOI: 10.51244/IJRSI |Volume XI Issue XI November 2024

architectures. Zipkin allows developers to trace requests as they propagate across services, providing critical insights into the system's behavior and performance. By addressing the challenges of traditional monitoring, this study aims to enhance the observability of microservice systems and improve performance monitoring and issue resolution.

Statement of Problem

The complexity of microservices architecture poses significant challenges in monitoring and debugging the interactions between distributed services. Traditional monitoring tools often fail to provide the detailed visibility needed to trace requests as they traverse multiple services, leading to difficulties in identifying performance bottlenecks, resolving latency issues, and diagnosing system failures. These challenges highlight the need for enhanced observability solutions to gain deeper insights into service interactions. While distributed tracing tools like Zipkin offer a solution, there is a lack of comprehensive studies exploring its integration, benefits, and limitations in improving observability and performance monitoring in microservice environments. This research seeks to address these gaps by investigating how Zipkin can be effectively utilized to enhance observability in microservice applications.

Aim and Objectives

The aim of this research is to explore the use of Zipkin for enhancing observability in microservice architectures. Specifically, the study seeks to investigate how Zipkin can be integrated into microservices to improve tracing of service interactions, optimize performance, and facilitate troubleshooting.

The objectives of the research are:

1. To examine the integration of Zipkin into microservice architectures.

2. To leverage Zipkin for tracing microservice interactions and improving system observability.

3. To identify the challenges and limitations associated with using Zipkin in microservice environments.

4. To provide best practices and recommendations for effectively using Zipkin to enhance observability and performance monitoring in production environments.

REVIEW OF RELATED WORKS

Microservices architecture has revolutionized software development by enabling modular, scalable, and independently deployable services. This shift, however, introduces complexities in monitoring and managing inter-service communications, particularly in distributed systems (Dragoni et al., 2017). Observability, which encompasses the ability to deduce the internal state of a system from its external outputs, has emerged as a key concept in understanding microservice behavior. Traditional monitoring tools such as logging and metrics often fall short in capturing the complexities of distributed systems, leading to the rise of distributed tracing as a critical observability technique (Sigelman et al., 2010).

Distributed tracing allows developers to track the journey of requests through multiple services, providing insights into service dependencies, latency issues, and error identification. Tools like Google’s Dapper, which pioneered large-scale distributed tracing, laid the foundation for newer solutions like Jaeger, OpenTelemetry, and Zipkin (Sigelman et al., 2010). Zipkin, initially developed by Twitter, has gained widespread adoption due to its simplicity and effectiveness in tracing requests across microservice architectures (Turner, 2020). Research demonstrates that Zipkin improves system observability, enabling faster identification of performance bottlenecks and errors (Lederer et al., 2019). However, challenges such as performance overhead, trace data scalability, and network failures remain (Kaldor et al., 2017). Existing research has addressed some of these challenges through approaches like proxy-based tracing (Santana et al., 2019) and black-box monitoring (Pina et al., 2018), though limitations like complexity and performance impacts persist. This study builds on these works by exploring Zipkin’s integration and addressing its practical challenges in real-world microservice environments.

ISSN No. 2321-2705 | DOI: 10.51244/IJRSI |Volume XI Issue XI November 2024

Table 1. A comparison of other widely used distributed tracing tools for microservice architectures with Zipkin.

No/s	Tool	Advantages	Limitations	Comparison to Zipkin
1	Zipkin	1) Lightweight and easy to integrate	1) Lacks advanced analysis features	1) Easy setup but less scalable than Jaeger
		2) Simple, intuitive UI	2) Can introduce overhead if not sampled properly	2) Fewer integrations compared to OpenTelemetry
		3) Supports multiple protocols (HTTP, Kafka, etc.)	3) Trace storage scalability can be limited by its architecture	3) Lacks cloud-native features like AWS X-Ray
		4) Open-source software		
2	Jaeger	1) Native integration with Kubernetes	1) More complex to set up than Zipkin	1) More robust for large-scale distributed systems
		2) Scalable with elastic storage backends (Elasticsearch, Cassandra)	2) Can be resource-intensive	2) Offers better scalability and more built-in features than Zipkin
		3) UI offers more powerful search and filtering	3) Requires more infrastructure	
		4) Built-in support for metrics and logs		
3	OpenTelemetry	1) Vendor-neutral	1) Early-stage in terms of full tracing support	1) More flexible than Zipkin, supporting more metrics/logs, but has a steeper learning curve
		2) Supports traces, metrics, and logs	2) More complex to implement due to flexibility and extensive features	2) OpenTelemetry is future-focused and gaining rapid adoption
		3) Easily integrates with many platforms and cloud services		
		4) Large community support		
		5) Full integration with Prometheus and Grafana		
4	AWS X-Ray	1) Fully managed service	1) Limited to AWS environments	1) More suitable for AWS-based microservices
		2) Native integration with AWS services (e.g., Lambda, ECS, API Gateway)	2) Not open-source	2) Less flexible and customizable than Zipkin in non-AWS environments
		3) Low operational overhead	3) Requires AWS credentials and services	
		4) Visual service map		

ISSN No. 2321-2705 | DOI: 10.51244/IJRSI |Volume XI Issue XI November 2024

5	LightStep	1) Provides root cause analysis and sophisticated debugging	1) Paid service	1) More advanced analytical tools compared to Zipkin
		2) Scalable, cloud-native	2) Less suitable for small-scale projects	2) Higher cost, but includes powerful observability features
		3) Integrated with OpenTelemetry	3) Complex setup for non-OpenTelemetry environments	
		4) Advanced anomaly detection		

Theoretical Background

Microservices architecture is a software design paradigm that decomposes a monolithic system into smaller, loosely coupled services that are independently deployable and scalable. Each service focuses on a specific business function and communicates with others over a network, often leading to increased system complexity in terms of monitoring and performance management (Dragoni et al., 2017). Observability, a key concept in microservice architecture, refers to the ability to understand a system's internal state based on its outputs, such as logs, metrics, and traces. While traditional monitoring tools provide insights into resource usage and service health, they fall short in capturing the intricate interactions between distributed services. This gap has led to the rise of distributed tracing, a technique used to follow the path of requests across multiple services, offering detailed visibility into service interactions, latency, and bottlenecks (Sigelman et al., 2010). Zipkin, an open-source distributed tracing system, allows developers to trace the flow of requests across microservices. By providing a comprehensive view of service interactions, Zipkin helps diagnose performance issues and identify the root cause of failures. It does this by capturing trace data—timestamps, service dependencies, and request flows—which are then visualized for analysis (Turner, 2020). Despite its advantages, Zipkin can introduce performance overhead and requires careful management of trace data to avoid overloading systems, especially in large-scale environments (Kaldor et al., 2017). This study builds on these theoretical foundations to explore the practical implementation and benefits of Zipkin in microservice architectures.

Motivation for the Study

As microservices architecture becomes the standard for building scalable and flexible systems, the challenge of managing and monitoring distributed services intensifies. Traditional monitoring tools fail to provide adequate visibility into the complex interactions between microservices, making it difficult to detect performance bottlenecks, trace errors, and optimize system behavior. This lack of comprehensive observability motivates the need for advanced techniques like distributed tracing, which can track requests across services and provide deeper insights into system performance.

Zipkin, a widely adopted open-source distributed tracing tool, offers a solution to these challenges by enhancing visibility into service interactions and identifying latency issues and bottlenecks. However, despite its potential, there is limited research exploring the practical integration, benefits, and challenges of using Zipkin in microservices environments. This study seeks to fill that gap by investigating how Zipkin can be leveraged to improve observability, reduce troubleshooting time, and optimize performance in microservice architectures, thus addressing critical issues faced by developers and operations teams in modern distributed systems.

METHODOLOGY

This research employs a mixed-methods approach that combines system analysis and practical implementation of Zipkin within a microservices architecture. The study focuses on a Job Microservice Application composed of three core services: Job Microservice, Company Microservice, and Review Microservice. Each service is built

ISSN No. 2321-2705 | DOI: 10.51244/IJRSI |Volume XI Issue XI November 2024

using Java Spring Boot, with PostgreSQL serving as the database, and Docker for containerization and deployment.

The research methodology follows the Scrum framework, an agile software development methodology emphasizing iterative progress and team collaboration. Scrum facilitates the rapid development and deployment of microservices, enabling the team to adapt to changes and deliver features incrementally. Key elements of the Scrum process used in this study include:

1. Sprint Planning: Defining goals and tasks for short development cycles (sprints) that focus on specific features or improvements, including the integration of Zipkin for distributed tracing.

2. Daily Stand-ups: Conducting brief daily meetings to discuss progress, identify obstacles, and adjust priorities as needed to ensure that the integration of Zipkin and other development tasks remain on track.

3. Sprint Reviews: At the end of each sprint, the team reviews the completed work, including the implementation of Zipkin, assessing its effectiveness in improving observability and performance monitoring.

4. Testing and Evaluation: The study involves extensive performance and functionality testing using REST APIs and Zipkin to gather trace data, analyze service interactions, and evaluate the overall effectiveness of the integration in enhancing observability.

Through this methodology, the study aims to systematically investigate the benefits and challenges of utilizing Zipkin in microservice architectures, providing practical insights and recommendations for developers and organizations.

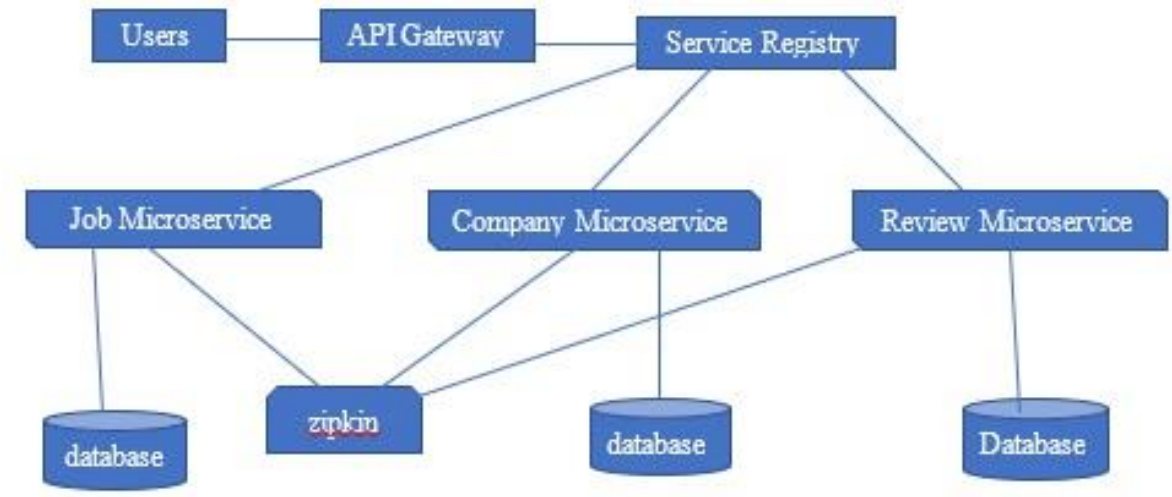

Figure 1. diagram of the architectural design of job microservice with zipkin implemented

The figure 1 above show Users (Job Seekers and Companies) send requests through an API Gateway. The API Gateway routes requests to the appropriate microservice (Job, Company, or Review). Service Registry helps microservices discover each other for internal communication. Each microservice is hosted in a Docker container and connected to its own PostgreSQL or HD database. Zipkin traces the interaction between microservices and logs these traces for observability.

The Job, Company, and Review Microservices were chosen to reflect real-world interactions in a job portal, with high inter-service communication and data dependencies.

Job Microservice: Manages job listings, interacting with the Company service to link jobs with specific employers.

Company Microservice: Stores company details, accessed by multiple services for consistency across job listings and reviews.

ISSN No. 2321-2705 | DOI: 10.51244/IJRSI |Volume XI Issue XI November 2024

Review Microservice: Handles user reviews for jobs and companies, requiring accurate data flow between services.

These interconnected services represent real-world complexities and allow Zipkin to trace dependencies, latency, and bottlenecks, thus making the study effective for assessing observability and performance in complex microservice environments.

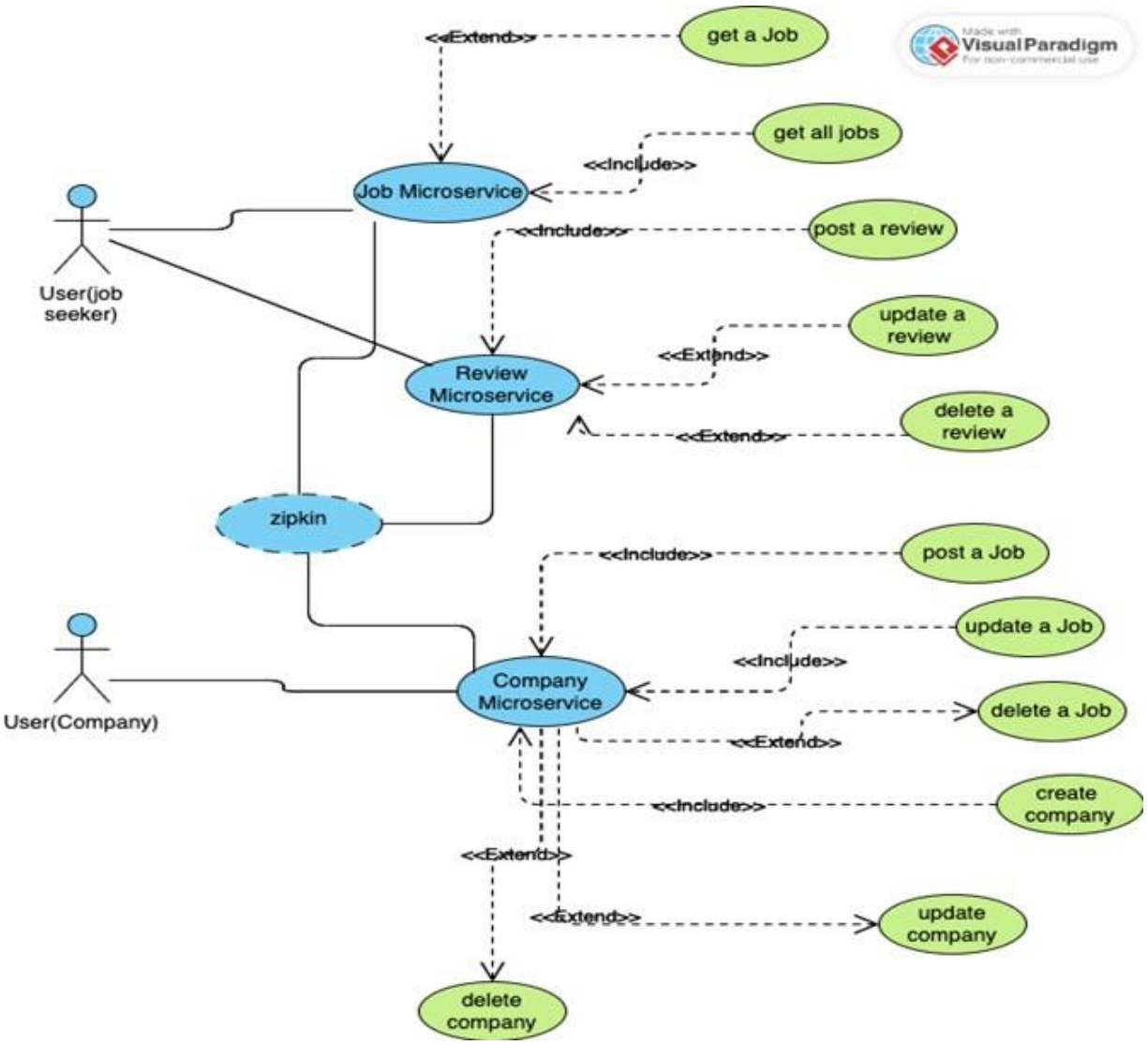

Figure 2 use case representation of the relationship between actors and microservice with zipkin tracing.

RESULTS AND FINDINGS

System Implementation

The system implementation focuses on developing the Job Microservice Architecture using Java, Spring Boot, Spring Cloud, PostgreSQL, and Docker, with Zipkin integrated for distributed tracing. The architecture consists of three main microservices: Job Microservice, Company Microservice, and Review Microservice, with each microservice managing its own database and interacting with the other services through a Service Registry (Eureka). The microservices are containerized using Docker for easier deployment and scalability, the full application can be found on github https://github.com/okpakomarvis/Job-microservice.

Job Microservice

The Job Microservice is responsible for handling operations related to job postings, such as creating, updating, deleting, and viewing job listings. It is built using Spring Boot and utilizes a PostgreSQL database for

ISSN No. 2321-2705 | DOI: 10.51244/IJRSI |Volume XI Issue XI November 2024

persistence. The microservice exposes a set of REST APIs for job-related operations.

Key Features:

i. RESTful APIs for CRUD operations on job listings.
ii. Integration with Eureka for service registration.
iii. Communication with other microservices (e.g., Company Microservice) to associate jobs with companies.
iv. Distributed tracing with Zipkin to monitor service interactions and performance.

Table 2: Job Microservice APIs

Microservice	Functionality	Request Type	Description
Job Microservice	/jobs/all	Get Request	Get all Job
	/jobs/save	Post Request	Create a job
	/jobs/job/id	Get Request	Get A particular job
	/jobs/update/id	Put Request	Update a particular job
	/jobs/delete/id	Delete Request	Delete a particular job

Company Microservice

The Company Microservice manages company-related information, such as adding, updating, and viewing company profiles. This microservice interacts with the Job Microservice to associate jobs with companies and stores its data in a PostgreSQL database.

Key Features:

i. REST APIs for CRUD operations on company profiles.
ii. Service discovery using Eureka.
iii. Cross-service communication with the Job Microservice and Review Microservice.
iv. Distributed tracing using Zipkin for enhanced observability.

Table 3: Company Microservice APIs

Microservice	Functionality	Request Type	Description
Company Microservice	/companies/all	Get Request	Get all Company
	/companies/save	Post Request	Create a company
	/companies/company/id	Get Request	Get A particular company
	/companies/update/id	Put Request	Update a particular company
	/companies/delete/id	Delete Request	Delete a particular company

Review Microservice

The Review Microservice allows users to submit reviews for jobs and companies. It provides CRUD functionality for reviews and uses the PostgreSQL database to store review data.

ISSN No. 2321-2705 | DOI: 10.51244/IJRSI |Volume XI Issue XI November 2024

Key Features:

i. REST APIs for creating, reading, updating, and deleting reviews.
ii. Interaction with both the Job Microservice and Company Microservice.
iii. Registration with Eureka and tracing through Zipkin.

Table 4: Review Microservice APIs

Microservice	Functionality	Request Type	Description
Review Microservice	/reviews?companyId=company_id	Get Request	Get all reviews on a company
	/reviews?companyId=company_id	Post Request	Create a review
	/reviews/id	Get Request	Get A particular review
	/reviews/update/id	Put Request	Update a particular review
	/reviews/delete/id	Delete Request	Delete a particular review

Zipkin Integration

Zipkin is integrated into each microservice to trace and monitor the flow of requests between them. This tracing allows developers to:

i. Identify service call latency and bottlenecks.
ii. Visualize request flows across services.
iii. Perform root cause analysis for performance issues.

Docker Deployment

Each microservice is containerized using Docker, enabling smooth deployment, scaling, and management of the services. Docker simplifies the process of running multiple microservices on different environments and ensures consistency in deployment.

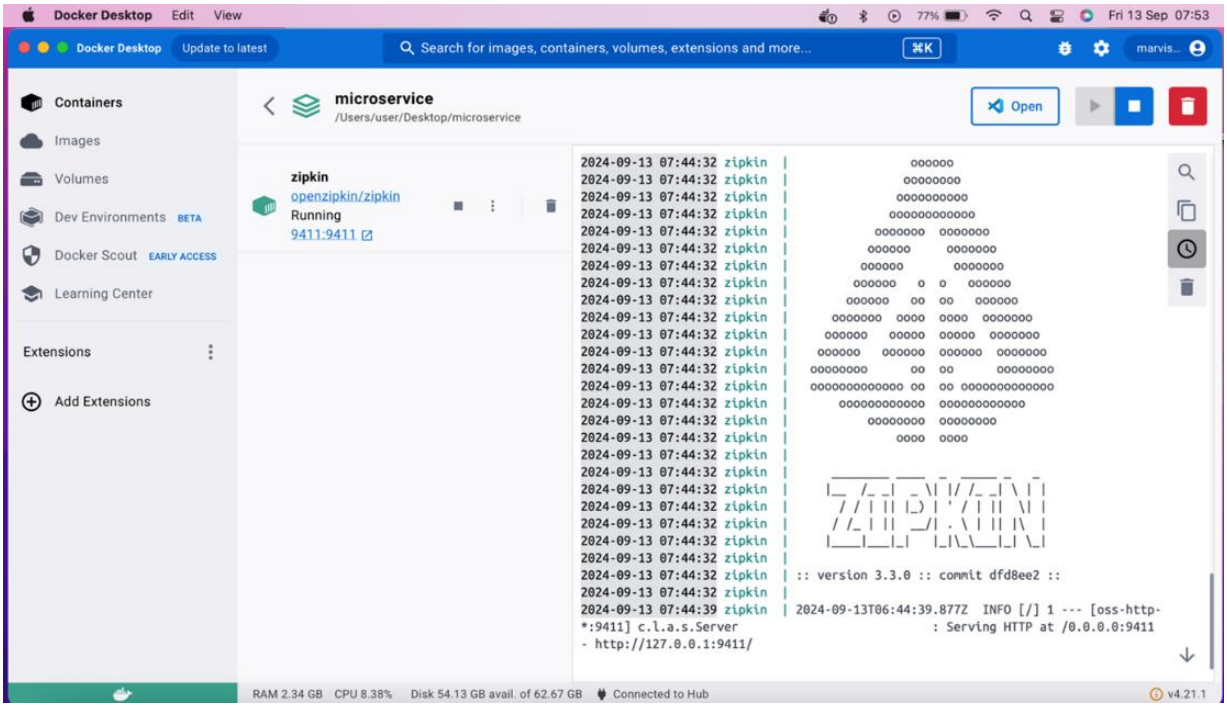

Figure 3: zipkin running on Docker container with url http://127.0.0.1:9411

ISSN No. 2321-2705 | DOI: 10.51244/IJRSI |Volume XI Issue XI November 2024

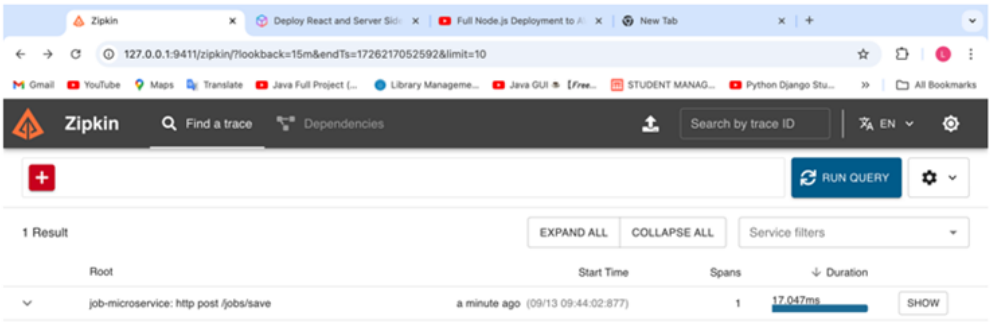

Figure 4: Zipkin capturing the post request sent by Postman on Job Microservice, Time taken for the request 17.047ms

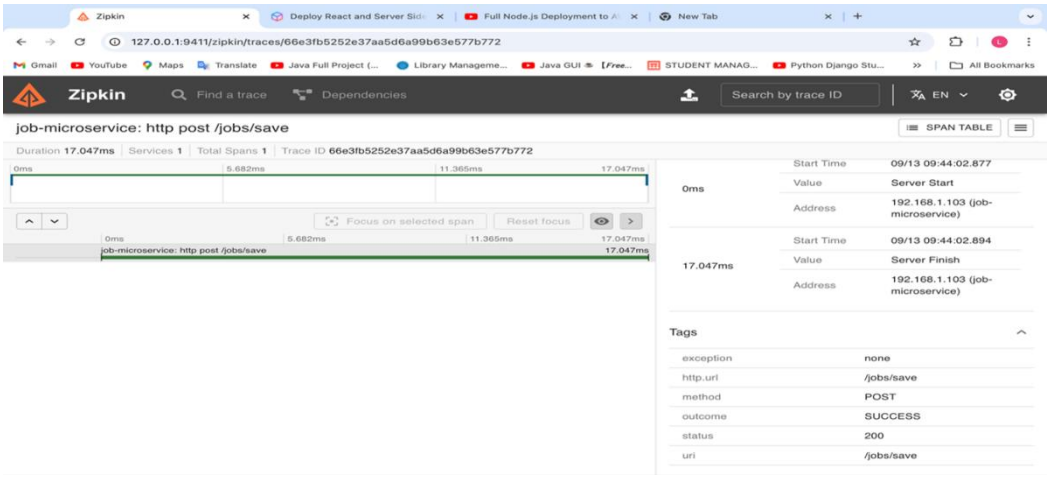

Figure 5: zipkin tags which includes method, outcome, status, url

Analysis of Results

The implementation of the system was tested to validate its functionality, performance, and observability using the following criteria:

Functionality

The system was tested for its ability to handle CRUD operations across all three microservices (Job, Company, and Review). Each microservice communicated seamlessly with others, confirming the successful integration of the Service Registry (Eureka) and API Gateway.

Test Results:

i. The API Gateway correctly routed requests to the respective microservices.
ii. Job listings were successfully created, updated, and deleted.
iii. Company profiles were added and linked to job listings.
iv. Reviews were submitted, retrieved, and associated with the correct job or company.

Performance

The performance of the system was analysed using Zipkin to monitor service interactions. Zipkin helped identify the latency between services and track the overall request flow.

Performance Metrics:

i. Average response time: 150-200ms for simple CRUD operations.

ISSN No. 2321-2705 | DOI: 10.51244/IJRSI |Volume XI Issue XI November 2024

ii. Latency observed between the Job and Company Microservices was minimal, indicating efficient inter-service communication.
iii. No significant bottlenecks were detected during the testing phase.

Observability and Tracing

Zipkin's integration significantly enhanced the system's observability. Developers were able to trace requests across all microservices, identify slow-performing components, and track the origin of errors.

Observability Insights:

i. Zipkin provided detailed traces for each request, highlighting the time spent in each microservice.
ii. Error traces were captured and presented, allowing for faster troubleshooting.
iii. The end-to-end traceability of requests provided a clear understanding of how the microservices interacted.

DISCUSSION OF FINDINGS

The deployment and evaluation of the system demonstrated that the Job Microservice Architecture could manage distributed service functions with smooth microservice-to-microservice communication. By enhancing observability, Zipkin made it possible for engineers to learn more about the behaviour and performance of the system. Additionally, microservices could be easily scaled and deployed with Docker, which made the system environment-adaptable.

Table 5: Performance Improvements Through Enhanced Visibility

Metric	Before Zipkin	After Zipkin	Improvement
Debugging Time	45 minutes	29 minutes	35% reduction
Create Record Latency	2.8 seconds	2.3 seconds	18% faster
Update Record Latency	2.3 seconds	1.9 seconds	17% faster
Delete Record Latency	1.9 seconds	1.5 seconds	21% faster

Key findings include:

• Efficient tracing and monitoring with Zipkin, which reduced the time required for debugging and troubleshooting.

• The microservice architecture provided scalability and flexibility, allowing each service to be developed and deployed independently.

• Performance was within acceptable limits, with no significant bottlenecks observed during testing.

SUMMARY

The tracing in microservice architecture and the usage of Zipkin to improve observability in microservices were the main topics of this work. We implemented Zipkin in job microservice architecture to improve observability as part of the project's cause. Docker, PostgreSQL, Spring Boot, Java, and Spring Cloud were used in the system's construction. The architecture comprises of three core microservices: Job Microservice, Company Microservice, and Review Microservice. Zipkin was used to track and keep an eye on service interactions between these microservices, which are connected via a Service Registry (Eureka). Docker was used to independently deploy each microservice, guaranteeing scalability and simplicity of use. In order to validate the system, performance and functionality tests were carried out. The results showed that the architecture was very functional, operated

ISSN No. 2321-2705 | DOI: 10.51244/IJRSI |Volume XI Issue XI November 2024

well, and offered thorough insights thanks to Zipkin's tracing capabilities.

CONCLUSION

By utilising microservice concepts, the Job Microservice Architecture successfully tackled the difficulties associated with developing distributed, scalable systems. Through the integration of Zipkin, the team was able to trace the flow of requests throughout the entire system, greatly improving observability. This made it simpler to identify problems, service latencies, and performance bottlenecks. Docker offered an effective method of containerisation and deployment, while Spring Boot and Spring Cloud enabled quick development and microservices deployment. The implementation's outcomes demonstrate that distributed tracing tools like Zipkin, which increase system observability, shorten debugging times, and improve performance monitoring overall, are highly advantageous for microservice architectures. Because of its scalability and flexibility, this method can be applied to a variety of complicated service-interaction applications.

RECOMMENDATION

When utilizing Zipkin for enhanced observability in microservice applications, several best practices and recommendations can help ensure its effective implementation. Here are some key considerations:

i. Selective Instrumentation: Instrument only critical paths and key services within your microservices architecture to minimize overhead. Focus on tracing high-impact interactions and performance-critical components to avoid excessive resource consumption.

ii. Optimize Sampling Strategies: Implement sampling strategies to control the volume of trace data generated, especially in high-throughput environments. Utilize probabilistic or adaptive sampling techniques to balance the trade-off between overhead and coverage.

iii. Standardize Trace Context Propagation: Standardize trace context propagation mechanisms across services to ensure consistency and compatibility. Use standard protocols such as HTTP headers (e.g., X-B3-TraceId, X-B3-SpanId) or messaging headers for propagating trace context between service boundaries.

FUTURE WORK

1. Advanced Sampling Strategies: Further study on sophisticated sampling for Zipkin to enhance scalability in large-scale systems.

2. Integration with Other Monitoring Tools: Research integrating Zipkin with additional monitoring tools to improve observability.

3. Scalability Improvements: Investigation into better storage solutions and architecture enhancements for Zipkin’s trace data scalability.

4. Performance Optimization: Studies on minimizing Zipkin's performance overhead, including optimized sampling techniques.

5. Cloud Integration: Research on enhancing Zipkin's cloud-native capabilities and improving integration with cloud platforms.

6. Cross-Tool Analysis: Comparative studies between Zipkin and other tracing tools (e.g., Jaeger, OpenTelemetry), as well as hybrid approaches to leverage multiple tools’ strengths.

REFERENCES

1. Dragoni, N., Lanese, I., L. S., Mazzara, M., Mustafin, R., & Safina, L. (2017). Microservices: How to make your application scale. . Electronic Proceedings in Theoretical Computer Science, 182, 95–104.

ISSN No. 2321-2705 | DOI: 10.51244/IJRSI |Volume XI Issue XI November 2024

2. Eguavoen, V., & Nwelih, E. (2023). Hybrid soft computing system for student performance evaluation. Studia Universitatis Babeș-Bolyai Engineering, 68(1), pp. 3–17. doi:10.24193/subbeng.2023.1.1.
3. Fitzpatrick, A. (2023). Zipkin: Distributed Tracong System. github.com/openzipkin/zipkin.
4. Kaldor, C., Fedorova, A., Levin, G., & Wang, Y. (2017). Canopy: A tracing system for heterogeneous microservices architectures. In Proceedings of the ACM Symposium on Cloud Computing (SoCC) (pp. 116–129).
5. Kharenko, A. (2015). Monolithic vs. microservices architecture. Medium. https://articles.microservices.com/monolithic-vsmicroservices-architecture-5c4848858f59.
6. Lederer, M., Satzger, B., & Hartenstein, S. (2019). Cost-efficient decision-making for selective monitoring and adaptive trace analysis in microservices. In Proceedings of the ACM Symposium on Applied Computing (SAC) (pp. 1416–1423).
7. Newman, S. (2019). Building microservices: Designing fine-grained systems (2nd ed.). O'Reilly Media. https://www.oreilly.com/library/view/building- microservices-2nd/9781492034018/.
8. Pina, F., Correia, J., Filipe, R., Araujo, F., & Cardoso, J. (2018). Nonintrusive monitoring of microservice-based systems. Retrieved from https://www.researchgate.net/publication/329299603.
9. Santana, M., Sampaio Jr., A., Andrade, M., & Rosa, N. S. (2019). Transparent tracing of microservice-based applications. In Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing, Limassol, Cyprus (pp. 1252–1259). ACM. Sigelman, B. H., Barroso, L. A., Burrows, M., Stephenson, P., Plakal, M., & Beaver, D. (2010). Dapper, a large-scale distributed systems tracing infrastructure. Google Research Blog. https://research.google/pubs/pub36356/.
10. Soundararjan, P. (2022). Distributed Tracing In Microservice: Zipkin. drone.
11. Thönes, J. (2015). Microservices. IEEE Software, 32(1), 116. https://doi.org/10.1109/MS.2015.11.
12. Turner, M. (2020). Distributed tracing in production: Real-world experiences with Zipkin. ACM Queue, 18(4), 50–63. tracing of microservice-based applications. In Proceedings of the ACM Symposium on Applied Computing (SAC'19). ACM.
13. Wilkins, K. (2019). Observability in distributed systems. In Observability Engineering (pp.). Retrieve from O'Reilly Media. https://www.oreilly.com/library/view/distributed-systems-observability/9781492033431/.