BUILDING A LOG STACK TRACE FRAMEWORK FOR MICROSERVICES ARCHITECTURE A SIMULATION STUDY

Renjith Ramachandran

Independent Researcher and Solutions Architect

ABSTRACT

Microservices architecture (MSA) has become increasingly popular in recent years due to its numerous benefits. Many companies are migrating their monolithic applications to an MSA, where each service operates independently, often with its own persistence layer, and interacts with both internal and external services. These services may also participate in an event-driven architecture, emitting and consuming events. However, as the number of services increases, it becomes more challenging to trace request flows across services for debugging. The sheer volume of logs generated by these systems can make data collection and analysis to identify the root cause resource-intensive and costly. While distributed tracing tools can track service calls, they often fall short in capturing custom logs across multiple services. Traditional logging approaches are inefficient in this context due to the complex interactions between services and limitations in log level configurations. This paper proposes a framework that can be integrated into each service within an MSA to collect log data and create a detailed stack trace of calls across services during events such as errors. This approach would help System Reliability Engineers and developers quickly debug issues and identify root causes. The framework supports structured logging, correlates logs with specific events and services, and enables efficient real-time monitoring of the system.

KEYWORDS

Distributed logging, Microservices Architecture, Event Driven Architecture, Framework.

1. INTRODUCTION

The term microservice was mentioned in 2014 by Martin Flower [2]. MAS is a distributed application where all its modules or elements are microservices and these can run independently [1] [2]. The complexity of Microservices architecture has increased over the years. New frameworks and architecture patterns were built to support microservices architecture. The concept of each service handling one specific functionality looked compelling initially. As the complexity of the business increased, microservices architecture became complex. Identifying an error through tracing also became complex [3].

Distributed tracing involves the process of monitoring, profiling and logging the execution path through a distributed application at runtime in response to a user's request. A user's request typically results in behavior that can span across multiple services in the application, resulting in a distributed trace, a detailed record of the execution path through the application [4].

David C. Wyld et al. (Eds): NWCOM, ASOFT, ITCCMA, CSITY, BDIoT, MLNLP, SIGPRO, AIFZ– 2024
pp. 57-66, 2024. - CS & IT - CSCP 2024 DOI: 10.5121/csit.2024.141905

Tracing tools are used to track calls between services for debugging errors and monitoring response times. Their popularity grew with the increased adoption of microservices architecture (MSA). These tools generate vast amounts of trace data from services for each request, making the collection and analysis of this data a significant challenge [5]. A critical aspect of tracing is identifying errors. When an error occurs, it's essential to capture not only the call stack but also the corresponding logs from each service. This helps the operational and development teams to debug issues more efficiently by examining the log stack trace. However, distributed tracing tools often provide log details only at the moment of the call, missing the full context of the invocation or method. To obtain the complete log details, the log levels in the application often need to be adjusted at runtime, which can lead to an overwhelming volume of log data, especially in high-traffic applications.

Several studies have been conducted to address this issue in complex distributed architectures. Identifying the root cause of failures in such environments is critically important, leading to multiple proposed solutions. Zhang et al. [8] reviewed various published works on failure diagnosis and found that most focus on root cause localization or failure classification rather than integrating error traceback with localization. Additionally, the paper, along with many researchers, suggests using machine learning and AI models to identify root causes from logs, which can significantly increase costs related to resources, infrastructure, and maintenance.

Pathak et al. propose a simpler solution with the Self-Adjusting Log Observability Tool [9]. Although the tool simplifies the process, adjusting log levels at runtime can have a ripple effect, particularly in high-traffic situations. Moreover, the tool employs a blast radius approach [9], which may not effectively trace back through all services in the chain.

The objective of this research is to address these limitations in tracing and error debugging, in microservices architecture (MSA) using a framework that is simple to implement and provides a log stack trace, rather than just a call stack trace, across all services. This approach would facilitate the quick identification and resolution of errors. The framework should automatically generate a log stack trace whenever an error occurs in any service within the chain of service invocations, eliminating the need for manual intervention. Furthermore, the framework should follow a consistent logging strategy, allowing both operations and development teams to effectively trace and resolve issues using the generated log data. This framework would significantly reduce the volume of logs generated, thereby accelerating the debugging process.

2. LOGGING

Software systems generate log files with information about the running program. Developers insert logging statements in to the which are then printed in to the log files and used at a later point for analysis [6]. Logging is crucial for traceability of the application. In MSA, each service generates logs. There are tools available which can aggregate logs from different services and give a unified view to the developers and reliability engineers.

2.1. Log Levels

There are different log levels available to limit the logging when the application goes to production.

Table 1. Log Levels [7]

Log Level	Comments
SEVERE	For fatal program errors
WARNING	For warning messages
INFO	For Informational messages during runtime
CONFIG	For Informational messages about config
FINE	Used for detailed info for debugging problems
FINER	Used for greater detailed info for debugging problems
FINEST	Used for greatest detailed info for debugging problems

3. LIMITATIONS OF THE CURRENT METHODOLOGIES

An increasing number of enterprises are adopting Microservices Architecture (MSA) due to its numerous benefits. However, as applications become more critical, any performance degradation or outages can lead to significant financial losses. In their study, Daniel et al. [10] identified "more complex monitoring" and "debugging difficulties" as two of the top 10 disadvantages associated with MSA.

Numerous studies have been conducted to address monitoring challenges, with a growing trend in using AIOps to tackle tracing problems. Common use cases for AIOps include anomaly detection, root cause analysis, and event correlation. However, due to the vast amount of data collected from various systems, machine learning and AI experts often need to invest considerable time in analysing the data to identify the root cause of issues [11][12]. Yunke further discusses the challenges of using AIOps in the thesis. AIOps is currently viewed as being in the phase of inflated expectations. Enterprises may face infrastructure-related challenges if they are unprepared to adapt to the requirements of AIOps. The thesis also elaborates on data quality challenges, emphasizing that data must be collected in a specific manner to satisfy AIOps requirements. Additionally, it addresses data manipulation challenges, noting that the large volume of data generated by distributed systems must undergo multiple transformations before it is suitable for AIOps [12].

The Traditional logging mechanism used in most of the microservices application in MSA has been useful until the complexity of the architecture and volume of the requests grew. They have the following limitations.

Limited Visibility: It will be difficult to get a unified view of the system in a complex distributed system [13].

Limited Error Context: Only the service where the error occurred logs detailed error information, leaving other services without context for troubleshooting.

Inconsistent Logging Levels: Each service operates at a different logging level, which can lead to inconsistent data across services and complicate debugging efforts.

Error Response Isolation: Other services in the chain only receive the error response from the invoking service, lacking detailed logs that is essential for root cause analysis.

Real-Time Log Level Adjustment: Modifying the log level in real-time during production to capture additional details can produce an overwhelming volume of log data, complicating management and analysis.

Potential for Performance Impact: Increased logging, especially at more detailed levels, can adversely affect the performance of services, leading to potential latency issues.

Complexity in Root Cause Analysis: Collecting and correlating logs from multiple services to identify the root cause of errors adds complexity to the debugging process.

Higher Resource Consumption: The generation of excessive logs can lead to increased storage requirements and resource consumption, potentially impacting system performance and costs.

Correlation of Logs: Due to the decoupled nature of services, tracking the flow of a single request or event across multiple services necessitates the use of correlation IDs or trace IDs to link logs from different services.

Considering the shortcomings of traditional logging mechanisms and modern approaches like AIOps, a new framework is required to overcome these challenges and complexities. This framework should bridge the existing gaps and provide a more efficient solution for monitoring and troubleshooting in modern systems. It should be easy to implement, collecting only the essential data needed for analysis while minimizing noise by excluding irrelevant logs.

4. LOG STACK TRACE FRAMEWORK

The goal of the Log Stack Trace Framework is to create a straightforward solution that overcomes the limitations of traditional logging methods and modern observability approaches. The framework aims to produce only relevant data for error tracing, eliminate the need for changing log levels at runtime, and provide a complete trace of logs across multiple systems when an error occurs. Logs from different systems will be collected based on specific events, and upon the occurrence of such an event, all related logs will be aggregated and sent to a centralized system for analysis.

Figure1 below illustrates a sample flow in a microservices architecture (MSA). In the diagram, "MS App" represents a microservice. The logging framework is integrated into the application. As shown, MS App 1 invokes MS App 2 and MS App 3 using HTTP API requests. MS App 3 interacts with a database to save or retrieve data. Meanwhile, MS App 2 calls MS App 4 via an HTTP API request, and MS App 4 subsequently calls MS App 5, which in turn communicates with an external system located outside the enterprise firewall.

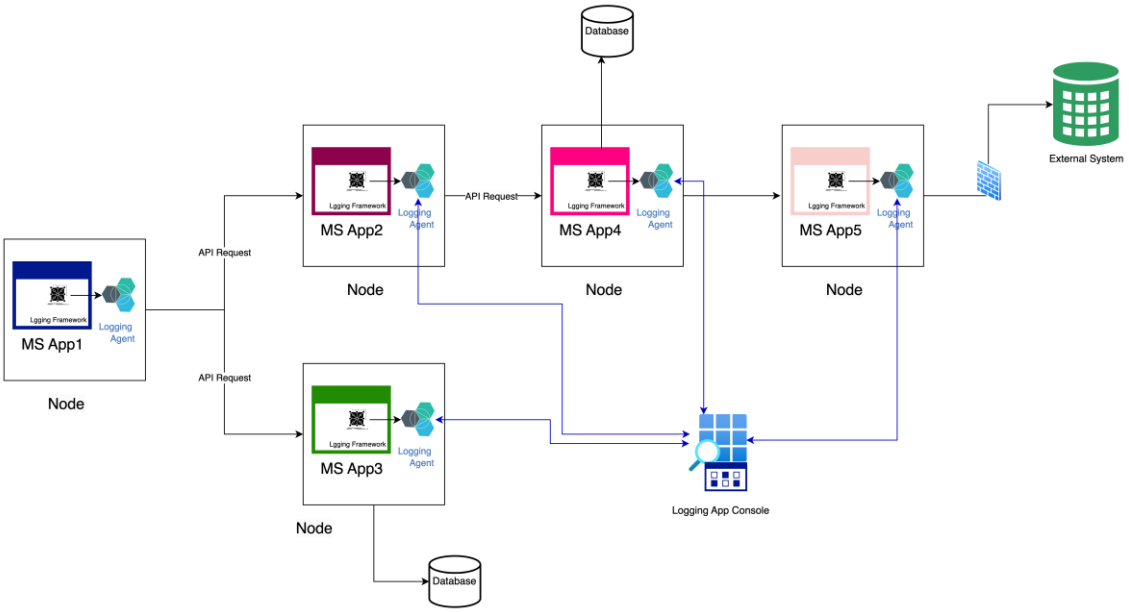


Figure 1. A Sample Application Flow in MSA

4.1. Components of the Framework

The Framework consists of the following components.

Table 2. Framework Components

Log Level	Comments	Indicator
Logging Framework	An embedded framework is installed within the MS App, providing functionalities for logging.	Lgging Framework
Logging Agent	Side Car Agent which collects logs from Apps and pushes to a Logging App for analysis	Logging Agent
Logging App Console	Dashboard console which will collect all the logs and provide search functionalities and reporting	Logging App Console

4.2. Internals of the Framework

Figure 2 illustrates the call trace where App1 invokes App3 using an HTTP API request. The Logging Framework internally stacks or buffers the log entries whenever `logger.log` statements are executed. In the event of a specific trigger, such as an error, the Logging Framework captures and logs the trace, as shown in Figure 3, both to a local log file and in JSON format to the Logging App Console (Figure 4). App3 calls invokeDB(), which results in an error. This triggers App3 to generate JSON data, send it to the logging agent, and log the information. The framework captures all logged data preceding the error, providing a complete call stack, which facilitates easier debugging.

As the control returns to App1, as shown in Figure 2, the error thrown by App3 is propagated back to App1. App1 then handles the error. While handling the error, the logging framework logs

the stacked data from previous calls to the local log file and sends the corresponding JSON data to the Logging App Console via the Logging Agent.

The error could be of multiple reasons. It could be database connectivity or query issues or errors from the interfacing app or network related issues. The call stack trace along the log stack trace can help reliability engineers to identify the issue.

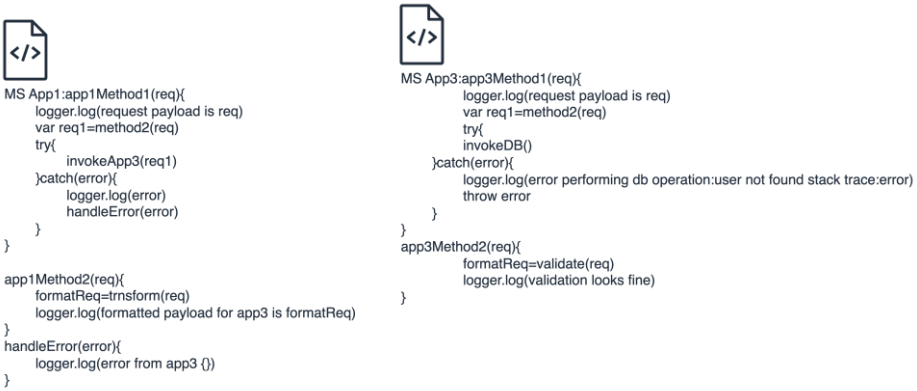
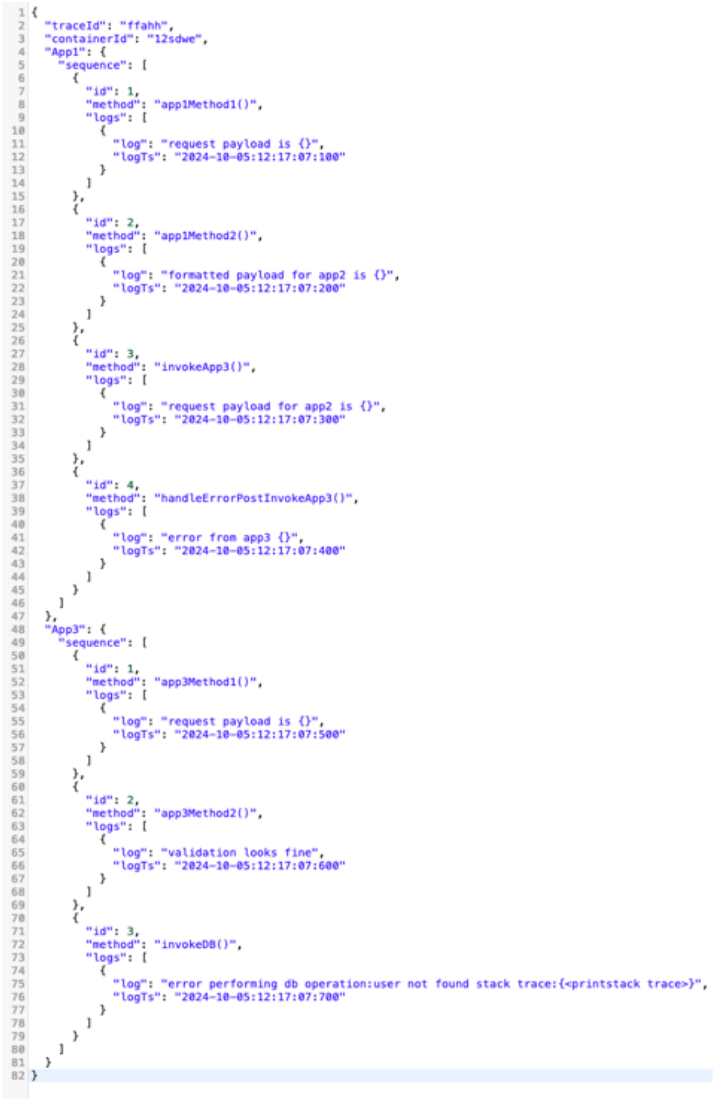


Figure 2. MS App1 invoking MS App3

4.3. Log Structure

```
traceId:containerId:app:sequence:method:log:logTs
ffahh.12sdwe:App3:3:invokeDB:error performing db operation:user not found stack trace:{<printstack trace>}:2024-10-05:12:17:07:700
ffahh.12sdwe:App3:2:app3Method2:validation looks fine:2024-10-05:12:17:07:600
ffahh.12sdwe:App3:1: app3Method1 payload is {}:2024-10-05:12:17:07:500
ffahh.12sdwe:App1:4: handleError:error from app3 {}:2024-10-05:12:17:07:800
ffahh.12sdwe:App1:3: invokeApp3:request payload for app3 is {}:2024-10-05:12:17:07:300
ffahh.12sdwe:App1:2:app1Method2():formatted payload for app3 is {}:2024-10-05:12:17:07:200
ffahh.12sdwe:App1:1: request payload is {}:2024-10-05:12:17:07:100
```

Figure 3. Log Trace Structure



```
{
  "traceId": "ffahh",
  "containerId": "12sdwe",
  "App1": {
    "sequence": [
      {
        "id": 1,
        "method": "app1Method1()",
        "logs": [
          {
            "log": "request payload is {}",
            "logTs": "2024-10-05:12:17:07:100"
          }
        ]
      },
      {
        "id": 2,
        "method": "app1Method2()",
        "logs": [
          {
            "log": "formatted payload for app2 is {}",
            "logTs": "2024-10-05:12:17:07:200"
          }
        ]
      },
      {
        "id": 3,
        "method": "invokeApp3()",
        "logs": [
          {
            "log": "request payload for app2 is {}",
            "logTs": "2024-10-05:12:17:07:300"
          }
        ]
      },
      {
        "id": 4,
        "method": "handleErrorPostInvokeApp3()",
        "logs": [
          {
            "log": "error from app3 {}",
            "logTs": "2024-10-05:12:17:07:400"
          }
        ]
      }
    ]
  },
  "App3": {
    "sequence": [
      {
        "id": 1,
        "method": "app3Method1()",
        "logs": [
          {
            "log": "request payload is {}",
            "logTs": "2024-10-05:12:17:07:500"
          }
        ]
      },
      {
        "id": 2,
        "method": "app3Method2()",
        "logs": [
          {
            "log": "validation looks fine",
            "logTs": "2024-10-05:12:17:07:600"
          }
        ]
      },
      {
        "id": 3,
        "method": "invokeDB()",
        "logs": [
          {
            "log": "error performing db operation:user not found stack trace:{<printstack trace>}",
            "logTs": "2024-10-05:12:17:07:700"
          }
        ]
      }
    ]
  }
}
```


Figure 4. Log Trace Structure in JSON format

Figure 5 shows Microservice Apps at scale. Each of the new containers are identified using the container id in the log file.

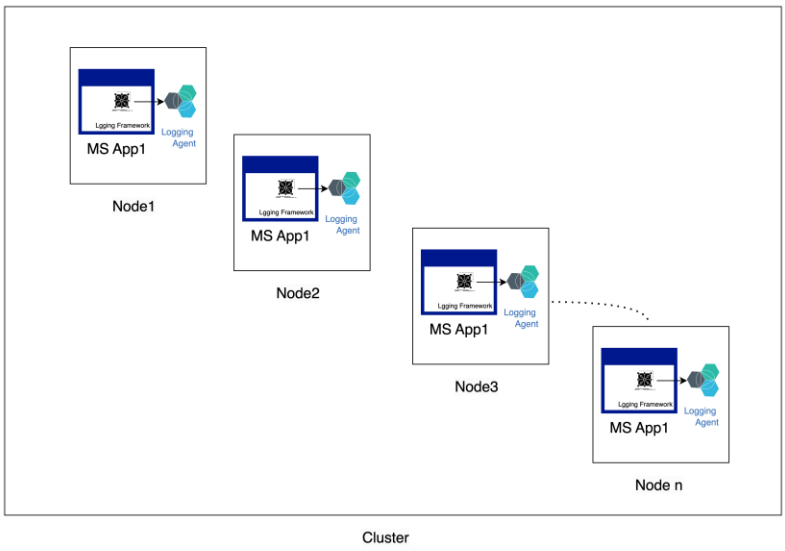


Figure 5. Cluster with more nodes

4.4. Advantages of the Framework

Easy Integration: The framework is available as a library that can be easily incorporated into microservice applications. It can also be set up without the need for a logging agent or an app console. The framework generates log entries in a predefined format that can be grouped and analysed effectively.

Backward Compatibility: The framework can function as a traditional logger, supporting various log levels for flexibility.

Logs Only Relevant Information: It produces only relevant log entries, minimizing unnecessary data and simplifying the process of issue analysis.

No Complex Infrastructure Requirements: The framework does not rely on complex data analysis systems, machine learning models, or AIOps infrastructure for error analysis.

Automatic Log Collection During Errors without log level updates: When an event such as an error occurs, the framework automatically collects all necessary log information from different systems, regardless of the current log level settings.

Effective Error Localization: Developers and reliability engineers can efficiently pinpoint the source of errors and trace the logs back to the respective services.

Reduces False Alerts: As the framework is capturing logs during errors and there is a way to correlate the errors from different systems using id, the number of false alerts reduces.

4.5. Future Scope

Further study is necessary to ensure that the framework can accommodate all potential scenarios. Additional testing is also required to extend the framework to support event-driven architecture. Once the research is complete and the results are favourable, the framework can be published to a centralized repository, such as GitHub, for developers to access and integrate into their

microservices. Two frameworks are needed: one for the core logging framework and another for the logging agent. The framework can operate without the logging agent as a standard logger.

5. CONCLUSIONS

As microservices architecture gains traction due to its agility, scalability, and faster release cycles, it is crucial to develop frameworks that address some of its shortcomings. Architectural patterns and frameworks have been created to facilitate the adoption of microservices. However, the distributed nature of microservices introduces several challenges, including the difficulty of tracing logs across different systems. While numerous studies have explored this topic, most lean towards utilizing advanced technologies like AI or ML to address the issue. While the benefits of these technologies are undeniable, there is an urgent need for a simple framework that can be easily integrated with existing applications without incurring excessive resource, cost, and infrastructure overhead. This paper proposes an extensible framework designed to resolve issues associated with traditional approaches while minimizing overhead. The framework aims to alleviate one of the key pain points of microservices architecture, enabling companies to build applications more efficiently.

REFERENCES

[1] N. Dragoni, S. Giallorenzo, A. L. Lafuente, M. Mazzara, F. Montesi, R. Mustafin, and L. Safina, ‘‘Microservices: Yesterday, today, and tomorrow,’’ in Present and Ulterior Software Engineering. Cham, Switzerland: Springer, 2017, pp. 195–216.

[2] J. Lewis and M. Fowler, ‘‘Microservices: A definition of this new architectural term,’’ MartinFowler.com, 2014, vol. 25, nos. 14–26, p. 12.

[3] Velepucha, V., & Flores, P. (2023). A survey on microservices architecture: Principles, patterns and migration challenges. IEEE Access, 11, p.88339–88358. https://doi.org/10.1109/access.2023.3305687

[4] Jacob, S., Qiao, Y., & Lee, B. (2021). Detecting cyber security attacks against a microservices application using distributed tracing. Proceedings of the 7th International Conference on Information Systems Security and Privacy, https://doi.org/10.5220/0010308905880595 pp. 589-595

[5] Parker, A., Spoonhower, D., Mace, J., & Sigelman, B. (2020). Distributed Tracing in practice. O’Reilly Media, Inc. pp1-5

[6] Gholamian, S., & Ward, P. A. S. (2022). A Comprehensive Survey of Logging in Software: From Logging Statements Automation to Log Mining and Analysis. arXiv [Cs.SE]. Retrieved from http://arxiv.org/abs/2110.12489 pp. 1-5

[7] ONJava.com: An introduction to the java logging api. (n.d.). https://www.inf.ed.ac.uk/teaching/courses/ec/miniatures/logging-up.pdf, pp. 3-4

[8] Zhang, S., Xia, S., Fan, W., Shi, B., Xiong, X., Zhong, Z., ... & Pei, D. (2024). Failure Diagnosis in Microservice Systems: A Comprehensive Survey and Analysis. arXiv preprint arXiv:2407.01710.

[9] Pathak, D., Verma, M., Chakraborty, A., & Kumar, H. (2024, July). Self Adjusting Log Observability for Cloud Native Applications. In 2024 IEEE 17th International Conference on Cloud Computing (CLOUD) (pp. 482-493). IEEE.

[10] dos Santos Krug, D., Chanin, R., & Sales, A. Exploring the Pros and Cons of Monolithic Applications versus Microservices.

[11] L. Yu, Z. Zheng, Z. Lan, T. Jones, J. M. Brandt and A. C. Gentile, "Filtering log data: Finding the needles in the Haystack," IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2012), Boston, MA, USA, 2012, pp. 1-12, doi: 10.1109/DSN.2012.6263948.

[12] Y. Hua, “A systems approach to effective aiops implementation,” Ph.D. dissertation, Massachusetts Institute of Technology, 2021.

[13] P. Thantharate, "IntelligentMonitor: Empowering DevOps Environments with Advanced Monitoring and Observability," 2023 International Conference on Information Technology (ICIT), Amman, Jordan, 2023, pp. 800-805, doi: 10.1109/ICIT58056.2023.10226123.

AUTHOR

Renjith Ramachandran received his Bachelor's Degree in Electronics and Communications Technology from India and his Master's Degree in Computer Science from the US. He spent 12 years as a consultant, taking on various roles from Software Engineer to Architect, and working with clients in industries such as Telecom, Banking, and Insurance. He currently serves as a Solutions Architect, with research interests that focus on software architectures, emerging technologies, and the development of innovative tools and frameworks.