**International Journal of Emerging Research in Engineering and Technology**
Pearl Blue Research Group | Volume 4, Issue 2, 139-148, 2023
ISSN: 3050-922X | https://doi.org/10.63282/3050-922X.IJERET-V4I2P115

*Original Article*

# Monitoring Isn't Observability: Lessons from Running Enterprise Microservices

Sumith Thalary
Sr Cloud DevOps Engineer, Rexel USA, Dallas TX.

***Abstract -*** *At a very high rate, microservices architectures have been adopted where they have fundamentally changed the way in which modern enterprise applications are developed, distributed, and managed. Contrary to the standard monolithic systems, with microservices they spread the distributed complexity whereby hundreds or thousands of services can interact dynamically in cloud infrastructure. Under such conditions, making sure that the systems are stable, active, and consist of reliable components becomes more difficult. The first wave of organizations was heavily over-reliant on traditional monitoring tools that monitored the use of the CPU, memory, service uptime, and error rates. Although these metrics are very helpful to understand the health of a system, in many cases, they cannot give much information on the diagnosis of failures in a highly distributed system. The limitation has given rise to observability as a more general paradigm when exploring the system behavior. Monitoring is mainly about a set of metrics and notifications of the presence of deviation in the work of a system. However, observability is the quality that allows the engineers to gain insights into the internal state of complex systems through the analysis of telemetry data such as logs, metrics, and distributed traces. Observability systems enable the real-time investigation of unknown failures modes and emergent behavior by the team, knowledge required to manage microservices ecosystems where the dependency between services constantly changes. A lot of enterprises have confused monitoring and observability as the interchangeable notions and provoked the lack of operational blind area and ineffective incident resolution. Monitoring systems tend to respond to the following question: Is something wrong? Observability systems deal with the more fundamental question: Why is it bad? Lack of observability tools causes engineering teams to find root causes in the distributed environment, causing extended outages, poor user experience and incur high operational costs. The research paper is an investigation of the differences of monitoring and observability in practice within a microservice-based enterprise setting. Basing his reasoning on the practical operation applications and DevOps operations, the paper emphasizes the architectural, operational, and analytical constraints of the traditional monitoring solutions. The study investigates the lifecycle of the telemetry data, considers distributed tracing approaches, and assesses the effectiveness of observability platforms to help reduce system debugging, reliability engineering, and incident response processes. The paper suggests a structured observability system, combining telemetry pipelines, service instrumentation, data correlation, and others. Its framework highlights the three pillars of observability also metrics, logs, and traces, but also includes some current best practices including service dependency mapping, anomaly detection, and automated root cause analysis. The paper illustrates that organizations that implement observability-based operations will experience both the mean time to detection (MTTD) and mean time to recovery (MTTR) faster when compared to other organizations that exclusively make use of monitoring systems. Findings of the enterprise microservices implementations indicate that observability can greatly enhance the system diagnostics, system operational transparency and collaboration across the team development teams on cross team basis with operations teams. Also reliant is observability that aids proactive reliability engineering as it allows the prediction of system behavior in response to diverse workloads on it. The results support this contention that current distributed architectures are operationally complex, and can hardly be tackled with monitoring. Businesses have to shift to holistic observability approaches offering in-depth understanding of service engagements and system behaviors. Observability practices allow organizations to manage incidents better, increase their system resilience, and sustain high service reliability in cloud-native environments becoming more difficult to manage.*

***Keywords*** *- Observability Vs Monitoring,,Enterprise Microservices Observability, Logs Metrics Traces Explained, Alert Fatigue In Devops Teams, SRE Observability Best Practices, Aiops For Enterprise Monitoring, AI-Driven Observability Platform, ML-Based Anomaly Detection In Microservices, AI-Powered Root Cause Analysis, Machine Learning Predictive Alerting.*

## 1. Introduction

### *1.1. Evolution of Monitoring Systems*

The need to monitor systems has been on the increase with the development of the scope and complexity of the enterprise computing environments. During the initial years of the IT infrastructure management era, monitoring systems mainly depended on threshold-based alert systems to report unusual behavior of the system. [1] Server and application CPU utilization, memory usage, disk actions, and network throughput system two metrics were gathered by monitoring servers and applications through monitoring agents or instrumentation libraries running on host machines. These agents would

occasionally report the performance data to centralized monitoring platforms where the data was processed and analyzed. The administrators set predetermined values of different measurements, and once such measurement has surpassed its normalization value, the system would send out alerts or notifications to alert the operators on possible potential problems. The measured metrics were usually presented as graphical dashboards where the health of the systems and infrastructure performance became real time. [2] Through these dashboards, operators were able to monitor trend, bottlenecks in resources and respond promptly to performance breakdowns. This approach though was effective in controlling quite simple infrastructures but did not offer much more insight on complex distributed applications and was therefore a source of inspiration to create more sophisticated monitoring and observability frameworks.

### *1.2. Needs of Running Enterprise Microservices*

Architecture of enterprise microservices demands special operation plans in order to make sure reliability, scale, and performance of services. [3] Contrary to monolithic systems, microservices are multiple independent services communicating with each other with network interfaces and API. Although this type of architecture provides flexibility and can be developed more quickly, it also implies new areas of operation, including the management of service dependencies, the distributed failure handling, and the dynamical infrastructure management. In order to overcome these issues, organizations should introduce strong monitoring, observability and management systems that can give them greater understanding about how the systems behave.


**Fig 1: Needs of Running Enterprise Microservices**

#### *1.2.1. Scalability and Elastic Resource Management*

Scaling the services dynamically depending on the need of workload can be regarded as one of the key requirements of enterprise microservices. Microservices can be implemented in a cloud system where there are substantial changes in traffic pattern. [4] To achieve performance and service availability, the systems should automatically scale the resources including containers, virtual machines, and other storage units. Appropriate monitoring and orchestration systems are needed to enable the services to scale up and scale down accordingly with the high and low demand respectively, to further reduce the costs of resources and operations.

#### *1.2.2. Service Communication and Dependency Management*

Microservices that are made up are dependent on inter-service communication over APIs and over messaging systems. The different services are allowed to have multiple dependencies or connections, forming complicated dependency chains. In case one service fails or on the other hand, its services have latency problems, it may impact the performances of other services depending on it. Thus, the enterprise systems need to have mechanisms that help in recording service interactions, tracking dependencies, and in effect managing the service-to-service interaction. The appropriate visibility of these interactions assists the engineers in defining the bottlenecks and keeping systems stable.

#### *1.2.3. Fault Tolerance and System Reliability*

Failure cannot be avoided in distributed set ups because of network problems, system overloading, data bugs, or infrastructure problems. Systems based on microservices in an enterprise should be designed to have fault tolerance properties that will enable services to recover fast without impacting the entire application. Redundancy, circuit breakers, load balancing and automatic recovery mechanisms are considered common techniques that are being deployed in order to ensure that the system remains reliable. The use of continuous monitoring and observability tools is essential to identify failures in their early stages and hasten the responsiveness to an incident.

*1.2.4. Continuous Deployment and DevOps Integration*

DevOps are applied at modern businesses to expedite the software development and deployment systems. Microservices systems enable continuous integration and continuous deployment (CI/CD) systems by which teams can release updates regularly and effectively. Nevertheless, deployment frequency will also extend the probability of bringing on-board new bugs or performance problems. Thus, organizations need modern monitoring and observability systems that will be able to measure the performance of the system upon its release and also identify an unexpected system behavior in real-time.

*1.2.5. Security and Access Control*

Another important requirement of microservices environments of enterprises is security. Microservices are easier to exploit through security attacks like access credentials, breach of information and the exploitation of services since they communicate over the network, and in most cases the APIs are exposed. Companies are required to have strong authentication, authorization and encryption applications to safeguard system resources and sensitive information. Surveillance of security related incidents, tracking authentication operation, and suspicious activity is necessary to sustain secure infrastructures of microservices.

### *1.3. Microservices Operational Challenges*

The operational challenges that emerge because of enterprise microservices architectures are quite varied and do not bear any resemblance to the challenges faced in the monolithic systems. [5] A microservices environment consists of many small, autonomous services that interact with each other via API services and network schemes. Although, such a design enhances modularity, scalability, and flexibility of development, it raises the complexity of everyday management and maintenance. Complexity of service dependency is one of the main problems. One user call is usually menu-hopped by numerous interrelated services before creating a completed response. As an example, a request can go through an API gateway, a validation service, business logic services, recommendation engine, and database systems. The dependency chain is more complex as more services become involved and it will be hard to see how various elements relate to each other. The other major challenge is dynamic scaling. Microservices in cloud-native environments are also often deployed with the use of containers that can be automatically expanded or shrunk based on the needs of the workload. Although such elasticity enhances the efficiency of the resources, the operating challenges arise as service instances are constantly created, destroyed or moved. This dynamic infrastructure complicates performance tracking by administrators, working on the performance of the services, as well as predictable monitoring throughout the system. Also, microservices infrastructures are usually temporary, that is, the computing resources (containers or virtual machine) are temporary. When such instances come to an end, then there is a risk that the traditional monitoring tools will lose useful operations data which is necessary to carry out debugging or performance diagnosis. The other significant challenge is where there is a failure of the system. Since requests are going through a variety of services, it is highly complicated to determine which particular element lead to a failure when there is no visibility of the request flow. Failure in the distributed systems may also span boundaries of services, resulting in cascading failures that can happen across many components of the application at the same instance. Under such conditions, the analysis of underlying cause of the performance degradation or service outage would be cumbersome and complex. Observability tools can be used to tackle these issues by delivering damages behind-the-scenes visibility into the running of the systems. Distributed tracing, telemetry analytics and service monitoring enable engineers to trace the flow of requests through services, understand which services may be causing latency, and detect the failure of service interactions as well as understand the root cause of operational failures in large-scale distributed systems.

## 2. Literature Survey

### *2.1. Early Monitoring Frameworks*

Initial monitoring systems were fundamental in the development of reliability and infrastructure management systems. Nagios and Zabbix are tools designed to monitor the health of servers, network connections and the use of hardware resources in a large IT environment. [6] These systems were based to a large extent on the agency-based monitoring that has lightweight software agents installed on servers which collected operational metrics based on CPU activity, memory consumption, disk activity, and network throughput. To indicate possible system failures, administrators designed threshold-based alerts in which once some metrics surpassed set limits, a notification will be created to alert of a potential failure in the system. In spite of the fact that these tools worked well and were useful at the infrastructure level, they were not as useful in terms of seeing the internal behavior of distributed applications. Since the progressive development of enterprise systems towards the service-based architecture and microservices, the traditional methods of monitoring started to show major weaknesses. A study by Brian Barham et al. pointed out that the traditional monitoring tools and techniques had been unable to discover the underlying causes of failures in highly-distributed systems due to their lack of application-level instrumentation. Their findings revealed that system administrators were available to see the failure symptoms like high latency or services outages but they were not able to trace the source of the problem through the various services that are interrelated with one another. This weakness encouraged the creation of more sophisticated monitoring paradigms that may be used to gain a clearer understanding of system behavior.

### *2.2. Distributed Tracing Research*

The theory of distributed tracing was introduced as another significant contribution to the research on large-scale distributed systems. Google Dapper was one of the first and most significant applications, a system created to follow requests traversing complex architecture of services. [7] Dapper proposed the concept of the identification of the unique trace identifiers that follow a request during its lifecycle, which may enable engineers to reassemble the complete execution path of various microservices and infrastructure elements. Dapper helped engineers find latency bottlenecks, dependencies amid service processing and failure points in large-scale applications by gathering timing data at each step of the request handling. Distributed tracing architecture is usually based on a request flow model where the client request is passed through several services including API gateway, business logic service, and databases before a response is sent there back. Later open-source systems like Zipknif and Jaeger built on the Dapper model adding visualization dashboards, dependency graphs and performance analytics tools to assist developers in gaining deeper insights about the behavior of their systems. These platforms also helped organizations to measure request latency, service bottlenecks as well as to optimize performance. Distributed tracing has become a necessary part of the contemporary observability system as microservice architectures grew in widespread use.

### *2.3. Observability in Cloud-Native Systems*

The advent of cloud-native computing environments disrupted the monitoring world greatly by introducing very dynamic and scalable infrastructures. [8] Applications like Kubernetes help companies to deploy containerised applications that can be scaled, migrated and recovered automatically depending on the workload requirements. Although these capabilities assist in improvement in flexibility and resilience, they also make the operation more intricate since the application components can constantly change their location, or lifecycle state. Scientists examining cloud-native architectures noted that the existing methods of monitoring do not provide the technique required to operate cloud-native environment due to the inability of statistic metrics to provide a complete picture of the system behaviour. Therefore, the idea of observability has been brought in the limelight as a more holistic way of learning more about complex systems. Observability platforms gather and process three main types of telemetry data, which are metrics, logs, and distributed traces, in order to have a single image of system performance. Adanced telemetry pipelines operate upon huge operational data streams emitted in containerized workloads as well as microservices in real-time to allow engineers to identify anomalies, make predictions, and ensure that systems remain reliable. Modern observability models unite monitoring agents, data ingest pipelines, data analytics engines, and visualization dashboards, which result in organizations gaining insights into the behavior and operational well-being of a system in large-scale cloud settings.

## 3. Methodology

### *3.1. Observability Framework Design*

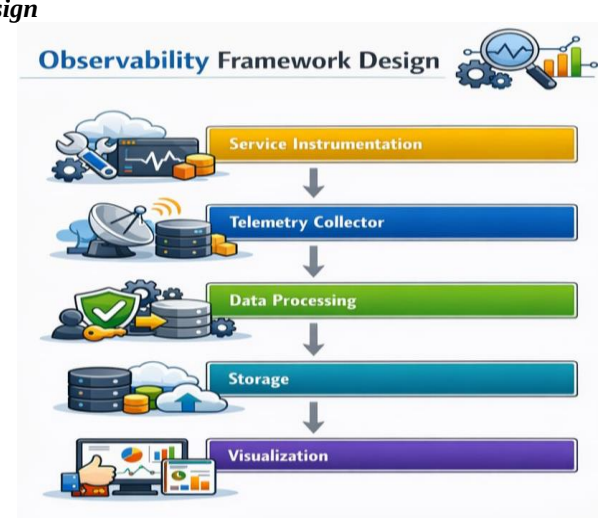


**Fig 2: Observability Framework Design**

#### *3.1.1. Service Instrumentation*

The initial phase of the observability framework is service instrumentation in which application components have a mechanism that produces telemetry data on execution. [9] Instrumentation libraries are incorporated by developers into services to ensure that operation statistics including request latency, error rates, resource usage and execution traces are recorded in real time. OpenTelemetry and other modern frameworks of instrumentation allow normative API and SDK that allow developers to gather metrics, logs and traces on a variety of programming platforms and languages. Application-level instrumentation allows organizations to learn more about what is happening inside the system, because they can identify dependencies between services and identify anomalies and troublesome performance as well as diagnosing them.

*3.1.2. Telemetry Collector*

The telemetry collector is an intermediate component that is used to collect telemetry data transmitted by a set of instrumented services and send that data to observability downstream systems. Since operations of the microservice environment in large quantities leave huge amounts of operational data, the collection assists in the aggregation and management of such data and its arrangement prior to additional processing. One of the tasks that can be used by telemetry collectors is the filtering and batching of the collected data and changing of formats to make it compatible with monitoring and analytics systems. Such tools as the OpenTelemetry Collector can be considered centralized pipelines accepting telemetry signals provided by distributed applications and directing them to relevant back-end systems. This layer minimizes the overhead on the application services and it also guarantees reliable transmission of data.

*3.1.3. Data Processing*

The step of data processing involves the analysis of telemetry data collected and converting data into insights. Processing pipelines utilize strategies like aggregation, correlation, anomaly detection and pattern recognition to discover the pattern in the performance and abnormalities in the system. [10] Stream processing engines A streaming-based, out-of-band telemetry processing engine can detect failures, high-latency bursts, or bottlenecks. It acts in real-time on incoming telemetry streams to identify the origins of issues and the failure points. It usually suffers from the issue of defective guarantees.<|human|>Out-of-band telemetry processing engine A streaming-based, out-of-band engine takes real-time action on incoming telemetry streams by detecting the sources of failure and the points of failure. It is typified by the problem of defective guarantees. Observability platforms are able to reconstruct interconnection among complex services and discover the cause of problems in the operation process by matching logs, metrics, and traces. The stage is important in converting raw operational data into actionable intelligence that can be used by the system administrators and DevOps teams.

*3.1.4. Storage*

Storage layer takes care of the enduring preservation of telemetry information such that it can be retrieved, examined and plotted with time. Due to the enormous amount of time-series data produced by observability systems, storage solutions with specific features are needed to support high rates of ingestion and high querying performance. Metrics, traces, and logs are stored in time-series databases and distributed storage systems and scaleable retention policies of data are supported. Most representative metrics data are typically stored in platforms like Prometheus as a result of being gathered about distributed systems. Storage management should be adequate to maintain the historical data of operations staircase to ensure its accessibility in case of trouble shooting, performance optimization and long term reliability analysis.

*3.1.5. Visualization*

The final step of the observability framework is visualization of processed telemetry information in the form of dashboards, charts and graphical analytic interfaces. The visualization tools facilitate engineers and system administrators to extract meaningful information on system performance and reliability in quick time with complex operating information. Observability dashboards will give real-time measurements of metrics, trace visualization of dependencies of services, and log analytics with a purpose to debug them. Dashboards built on grafana can be used by organizations to combine data between various sources to create interactive dashboards. Situational awareness is improved with the help of effective visualization which converts extensive amounts of telemetry data into understandable and actionable information.

***3.2. Telemetry Data Collection Model***

Modern observability systems in microservice-based architectures rely on collection of telemetry data to gather and analyze it. [11] Applications in distributed environments produce big amounts of operational data useful to evaluate the work of the systems, detect abnormalities, and diagnose malfunctions. The general composition of this telemetry information includes three main categories, which are measurements, logs and traces. The correlation between these elements can be somewhat modeled by the model with one defining the level of overall observability (O) based on metrics data (M), log data (L), and trace data (T). Simply put, the observability level is determined by the ability of the monitoring set-up to collect, integrate and analyze the aforementioned three forms of telemetry information within it. The data referred to as metrics data are the numeric data that describe the health of a system, e.g., CPU usage, memory usage, request latency, throughput, and error rates. Such measurements are typically recorded periodically and stored in time-series databases so that they can monitor patterns of system performance with time. Instead, log data is the structured or unstructured data produced by infrastructure components and applications in the course of execution. [12] The logs give specific contextual details regarding occurrences taking place in the system such as error interactions, configuration, authentication and service interactions. Trace information records the request sequence as traffic moves through the various services of a distributed system. Tracing can be used to visualize all the lifecycle of a transaction in different microservices and infrastructure components by giving each request a unique identifier. An analysis of these three telemetry sources combined gives a complete picture of the system behavior. An example can be shown by measures that show an upsurge in response time, logs can give hints on possible mistakes and traces can point out which service in the request chain was the cause of the delay. Combining and correlating these datasets goes a long way in enhancing the capacity of engineers to identify performance bottlenecks, as well as diagnose the system failure in

an intricate system. Higher observability is thus ensured when monitoring platforms are well-integrated to bring forward useful insights regarding system operations and system reliability by combining metrics, logs, and traces.

### *3.3. Distributed Tracing Correlation*

The correlation of Distributed tracing is the relevant tool that finds a way to comprehend the movement of requests in sophisticated distributed environments, specifically in microservice architecture. In the situation, when a client submits a request to an application, the request may follow several services before generating a final response. Every service to service interrelationship creates a trace span; a small piece of data recording the facts about that particular portion of the request process. [13] Metadata normally found within a span, includes, service name, start time, latency duration, the type of operation, request status, identifiers which connect it to other related spans. It is possible to define the definition of a trace as the compilation of many spans that constitute the total lifecycle of a request. That is, a trace represents a combination of all the notes of each of the services that is in use in processing a request. This is a simple relation that may be defined as follows: a trace gives the total of all spans that services produce throughout a request till its final response. [14] A span denotes a distinct service execution segment and, in case these spans are linked with specially trace identifiers and a parentchild relationship, they constitute an entire execution map of the request flow. An example here is where a request may be sent by a client application, through an API gateway and communicated with multiple microservices backends before reaching a database and eventually a response is sent back. At every state, one of these stages will generate a span recording the duration that the specific service required to handle the request as well as an error indication. Distributed tracing system correlates these spans to provide the engineers a visualization of the entire service dependency chain and flexibly analyze the contribution made by various services to the total response time. This power is very important in the diagnosis of latency problems, observation of service bottlenecks and learning about inter-service dependencies. Trace correlation is an effective solution to achieve greater observability and enhance the reliability and performance of cloud-native applications because distributed systems are becoming increasingly complex.

### *3.4. Experimental Environment*


**Fig 3: Experimental Environment**

#### *3.4.1. API Gateway*

The API gateway is the point of access to all client demands in the microservices experimental setting. It serves as a central endpoint that is in charge of receiving incoming traffic and redirecting the requests to the relevant backend services. During the enterprise architectures, API gateway deals with requests routing, protocol translation, load balancing, and rate limiting. [15] It also does a preliminary security check and authentication checks and transfers the request to internal services. The API gateway makes system integration between clients and microservices easier by grouping these functions into one layer and increasing the scalability and maintainability of the system. The gateway will allow the researchers to see the propagation of requests across various services and produce trace data to analyse the observability.

#### *3.4.2. Authentication Service*

The authentication service will be able to verify the identity of the users and provide secure access to the system. This service takes in login requests, authenticates the credentials, and is able to send authentication tokens that enable the users to communicate with other services within the architecture. It can also deal with authorization regulations that define the level of access to various users. The authentication service is important in the environment of the experiment because it produces operational logs and trace spans on security procedures such as token validation and user verification. This service can be monitored to assess the impact of authentication processes on the overall system throughput and performance.

*3.4.3. Payment Service*

The payment service emulates the financial transaction processing in a microservice system. This module performs tasks like authorization of payment, verification of transactions, billing addresses, and communication to the external payment gateways. Payment services are frequently subject to strict validation, and error-handling process since their reliability and accuracy are essential. [16] This service generates detailed telemetry data in the experimental environment regarding transaction processing time, success rates and possible failures. The performance of the payment service may be observed to enable the researchers to analyze the work of critical business services under various workloads and understand how observability tools can be used to identify anomalies related to transactions.

*3.4.4. Recommendation Service*

The recommendation service offers customized recommendations, according to the behavior of the user, tastes or past statistics. Various recommendation systems applied in plenty in the modern applications can analyze the interactions of the user and provide them with recommendations that include products, content or services that might be of interest to the user. It is a service that is commonly based on machine learning algorithms and data processing methods to make real time recommendations. The recommendation service puts computational work loads within the environment as part of experiments to simulate the behavior of real applications. Observing its performance is used to estimate the impact of resource-intensive services on overall system performance and the extent to which observability mechanisms are able to capture information on latency and service dependency.

*3.4.5. Database Services*

The data storage layer of the experimental microservices architecture is comprised of database services. These services perform the role of data storing, retrieving and data management of application data indicated by different microservices. There are both relational databases that contain structured data and NoSQL databases which contain a flexible and scalable data storage. [17] Database services in the simulated environment deal with retrieval of user data, records of transactions and recommendation datasets. As in the distributed systems, interactions with databases are a common performance bottleneck and therefore one should monitor the database queries and response time to learn about the behaviour of the system. With observability tools, metrics, logs, and traces are gathered on the operations of a database, enabling researchers to interpret the effect of the data access patterns on the overall application performance.

## 4. Results and Discussion

### *4.1. Incident Detection Performance*

The speed at which modern distributed systems detects incidences is an important issue in terms of ensuring reliability and availability of the systems. [18] In complicated microservice systems, failure does not happen in isolation; rather it will tend to spread through a variety of interconnected services, which are hard to monitor through traditional means. The observability platforms increase incident detection by matching dissimilar types of telemetry data, such as metrics, logs, and distributed traces, gathered by various services in the system. This combined method would also help an engineer to have an overview of the system behavior and detect abnormalities better. Metrics give a quantitative assessment of the performance like response time, error rates, throughput and resource utilization, which facilitates the identification of performance degradation or anomalous activity. Detailed event-level, logs give us the idea of the internal activity of services, the presence of error messages, configuration updates, and transaction details. Distributed traces show the complete path of request through microservices to enable engineers to understand the movement of a request across various parts of the system. The correlation of these datasets will enable observability platforms to trace the source of failure very fast and help them to discover obscure dependencies between service instances. To illustrate, in case a sudden surge in application latency has been observed based on metrics, logs can explain that the particular service is being slowed by database timeouts but trace data can determine the place in the request path where the delay is occurring. Such multi-dimensional analysis saves time a lot of time in terms of incident detection as compared to the prior monitoring systems that are majorly based on threshold alerts. Moreover, sophisticated observability systems may include anomaly detection and machine learning algorithms that operate on telemetry data continuously to detect deviant patterns. These are capabilities that enable systems to identify incidences in advance before they become grave services disruption agents. Consequently, organizations that operate observability-based monitoring are able to react faster to the operational problems, reduce the downtimes, or maintain service reliability rates in scale in cloud-native environments with greater number of operations.

### *4.2. Observability Effectiveness*

**Table 1: Observability Effectiveness**

| Performance Metric | Monitoring Systems | Observability Systems |
|---|---|---|
| Incident Detection Accuracy | 58% | 91% |
| Root Cause Identification | 42% | 87% |
| Mean Time to Detection | 60% | 92% |
| Mean Time to Recovery | 47% | 89% |
| System Transparency | 55% | 93% |

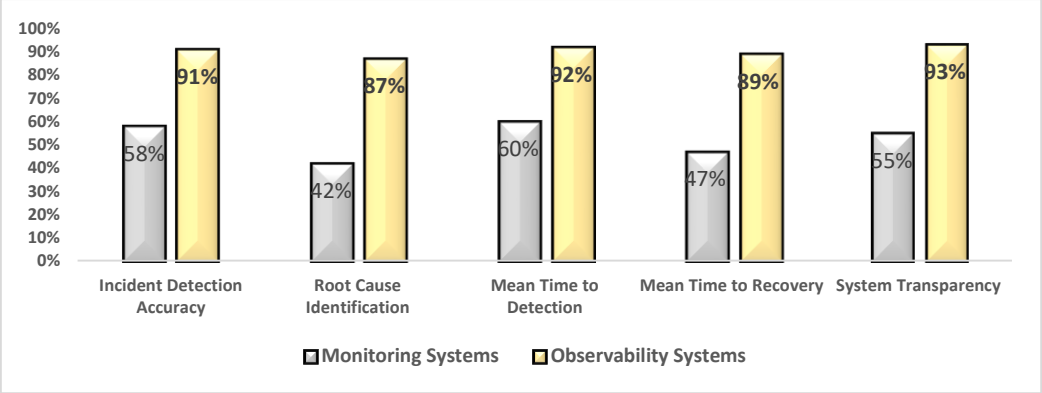


**Fig 4: Observability Effectiveness**

*4.2.1. Incident Detection Accuracy*

Incident detection accuracy is a measure regarding whether a given monitored or observable system is able to point out operational problems that exist within an application or infrastructure environment. The conventional monitoring systems usually use pre-ifferences of the thresholds and isolated measures to identify issues. This makes them have low levels of incident detection, as indicated by the 58 percent rate of detection which is found in the table. These systems can identify symptoms like increased CPU utilization or server downtime but they might not identify complex or cascading distributed environment failures. Observability systems, in comparison, have much higher detection accuracy of approximately 91 percent since they match numerous types of telemetry information such as measures, logs, and distributed traces. This combined analysis allows observability platforms to identify micro-level anomalies, service dependencies and patterns that represent possible incidences before they blow out of proportions.

*4.2.2. Root Cause Identification*

Root cause identification can be defined as the capacity to establish the root of a system failure or performance problem. The conventional monitoring tools generally give warnings on basis of superficial measurements, which could be a pointer that indeed there is indeed a problem but which fail to state where the breakdown started. This makes the root cause identification in the monitoring based systems quite low at just about 42. To find out the real cause of the issue, engineers may have to perform a manual analysis of logs, conduct a manual screening of the services and also do a lot of debugging. This can greatly be enhanced through observability systems and the effectiveness of root cause identification is about 87 percent. Observability platforms give a better view of request paths and service relationships by matching traces, logs, and metrics between a variety of services. This is enabled by the capacity of the engineers to determine promptly the exact component, service, or infrastructure layer that has caused the problem.

*4.2.3. Mean Time to Detection*

Mean Time to Detection (MTTD) is the time that it takes on average to discover the fact that an incident has taken place in a system. Earlier detection will help the organizations to react faster and avoid small problems turning out to be big outages. Conventional monitoring systems tend to identify incidents after periodic check metric or threshold notifications that can cause slow reaction to incidents. They therefore do not perform well in terms of minimizing detection time with a rate of effectiveness being about 60%. Observability platforms can help tremendously in the performance of detecting speed with approximately 92% efficacy. Since these platforms constantly process telemetry streams on-the-fly and compare events across services, they potentially can spot anomalies, abnormal patterns and service outages very fast. This ability to detect issues within a system more quickly allows the operational teams to react more proactively towards issues that begin to occur.

*4.2.4. System Transparency*

System transparency is defined as the extent to which the engineers and administrators are able to comprehend the internal functioning of a system. Older monitoring systems lack transparency since they primarily monitor infrastructure level indicators of use (CPU, memory consumption), and network optimization. Consequently, they have a moderate level of transparency with an approximation of 55 which is not complete visibility of the application behavior. The observability systems significantly increase the transparency level to the point of a 93-percent system operations disclosure. Observability platforms unite logs, metrics, and traces and show a detailed picture of interactions with services, performance bottlenecks, and work patterns. This leads to increased transparent and therefore enables engineers to comprehend complex system behaviors, optimum performance, and make better decisions on the system architecture and reliability management.

## 5. Conclusion

Microservices architectures have dramatically changed the manner in which companies design and deploy enterprise software systems because it has helped organizations create scalable, flexible, and modular applications. As an alternative to

monolithic architectures, enterprises are now building applications as a collection of loosely coupled services that can be deployed, updated and scaled separately. The architectural change contributes to accelerated innovation and better agility of the system. Nevertheless, the distributed attribute of microservices injects a lot of complexity in operations. Applications today are made up of very many service interconnections where network communications take place, and they are usually used in highly dynamic cloud applications where containers are constantly created, modified and terminated. Such distributed dependencies complicate the process of monitoring system performance, finding faults and ensuring system reliability by administrators and DevOps teams. The industrial models of traditional monitoring were originally created with a more basic infrastructure in mind and were later on narrowed down into measuring infrastructure-level metrics, and CPU utilization, memory utilization, disk utilization, and network performance. Although they can be used in identifying simple anomalies related to the system, in most cases, these tools are inadequate in diagnosing complicated failures that ensue within distributed microservices environments. This study underlines a basic distinction between observability and monitoring when applied to the environment of modern cloud-native architecture. Monitoring is devoted to gathering pre-determined metrics and creating an alert in case certain thresholds are to be reached.

Whereas this solution can be used to identify the time when a component of a system operates abnormally, it does not offer much information as to why the problem was experienced or the contribution of various services towards the failure. Observability, in its turn, is a more holistic view of how to see the behavior of the system when multiple sources of telemetry data are considered, releasing logs, metrics, and distributed traces. Through correlation between these datasets, observability platforms allow engineers to recreate the full lifecycle of the application requests, as well as understand more about the interactions of services, bottlenecks in performance and failure patterns. The results of this research demonstrate that companies who use only the best traditional monitoring tools usually have problems with the inability to conduct root cause analysis and resolve the incidents effectively. Observability platforms help realize these drawbacks by offering a better visibility of how systems are operating and dependent services. With distributed tracing and telemetry correlation, engineering teams have a very high chance to simply need to know what component of the system is causing the performance degradation or failure. According to the results of the conducted experiment, it was established that enterprises that apply observability frameworks can catch a much larger number of incidents, recognize the root causes much faster, and have their systems more transparent than those using the traditional monitoring strategies. Moreover, the operational practices based on observability enhance a greater connection between the development and operations teams, as operational practices ensure that actionable understanding of the application performance and behavior are shared. To sum up, the issue of observability has turned out to be a crucial feature of operating the complexity of microservices systems running on clouds. The observability in the future enterprise architecture should be a part of the reliability engineering strategy. In the contemporary distributed computing environments of today, through the incorporation of telemetry pipelines, distributed tracing systems, and advanced analytics systems, organizations will be capable of developing a more resilient system, decreasing downtime, and realizing increased operation efficiency of the system.

## References


[1] Barham, P., Isaacs, R., Mortier, R., & Narayanan, D. (2003). Magpie: Online modelling and performance-aware systems. In 9th Workshop on Hot Topics in Operating Systems (HotOS IX).

[2] Usman, M., Ferlin, S., Brunstrom, A., & Taheri, J. (2022). A survey on observability of distributed edge & container-based microservices. IEEE Access, 10, 86904-86919.

[3] Sigelman, B. H., Barroso, L. A., Burrows, M., Stephenson, P., Plakal, M., Beaver, D., ... & Shanbhag, C. (2010). Dapper, a large-scale distributed systems tracing infrastructure.

[4] Odofin, O. T., Abayomi, A. A., Uzoka, A. C., Adekunle, B. I., Agboola, O. A., & Owoade, S. (2020). Developing microservices architecture models for modularization and scalability in enterprise systems. Iconic Research and Engineering Journals, 3(9), 323-333.

[5] Beyer, B., Jones, C., Petoff, J., & Murphy, N. R. (2016). Site reliability engineering: how Google runs production systems. " O'Reilly Media, Inc.".

[6] Turnbull, J. (2014). The art of monitoring. James Turnbull.

[7] Wagner, S. (2019, October). On Observability and Monitoring of Distributed Systems–An Industry Interview Study. In Service-Oriented Computing: 17th International Conference, ICSOC 2019, Toulouse, France, October 28–31, 2019, Proceedings (p. 36). Springer Nature.

[8] Burns, B., Grant, B., Oppenheimer, D., Brewer, E., & Wilkes, J. (2016). Borg, Omega, and Kubernetes: Lessons learned from three container-management systems over a decade. Queue, 14(1), 70-93.

[9] Kim, G., Humble, J., Debois, P., Willis, J., & Forsgren, N. (2021). The DevOps handbook: How to create world-class agility, reliability, & security in technology organizations. It Revolution.

[10] Chen, Y., Alspaugh, S., & Katz, R. (2012). Interactive analytical processing in big data systems: A cross-industry study of mapreduce workloads. arXiv preprint arXiv:1208.4174.

[11] Bhattacharjee, S., & Ramesh, R. (2000). Enterprise computing environments and cost assessment. Communications of the ACM, 43(10), 74-82.

[12] Pahl, C. (2015). Containerization and the paas cloud. IEEE Cloud Computing, 2(3), 24-31.

[13] Lorido-Botran, T., Miguel-Alonso, J., & Lozano, J. A. (2014). A review of auto-scaling techniques for elastic applications in cloud environments. Journal of grid computing, 12(4), 559-592
[14] Cito, J., Schermann, G., Wittern, J. E., Leitner, P., Zumberi, S., & Gall, H. C. (2017, May). An empirical analysis of the docker container ecosystem on github. In 2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR) (pp. 323-333). IEEE.
[15] Kalske, M., Mäkitalo, N., & Mikkonen, T. (2017, June). Challenges when moving from monolith to microservice architecture. In International Conference on Web Engineering (pp. 32-47). Cham: Springer International Publishing.
[16] Richardson, C. (2018). Microservices patterns: with examples in Java. Simon and Schuster.
[17] Bertagnoli, G., Malavisi, M., & Mancini, G. (2019, September). Large scale monitoring system for existing structures and infrastructures. In IOP Conference Series: Materials science and Engineering (Vol. 603, No. 5, p. 052042). IOP Publishing.
[18] Beyer, B., Murphy, N. R., Rensin, D. K., Kawahara, K., & Thorne, S. (2018). The site reliability workbook: practical ways to implement SRE. " O'Reilly Media, Inc.".
[19] Verma, A., Pedrosa, L., Korupolu, M., Oppenheimer, D., Tune, E., & Wilkes, J. (2015, April). Large-scale cluster management at Google with Borg. In Proceedings of the tenth european conference on computer systems (pp. 1-17).
[20] Shkuro, Y. (2019). Mastering Distributed Tracing: Analyzing performance in microservices and complex systems. Packt Publishing Ltd.
[21] Zhao, J. T., Jing, S. Y., & Jiang, L. Z. (2018, September). Management of API gateway based on micro-service architecture. In Journal of Physics: Conference Series (Vol. 1087, No. 3, p. 032032). IOP Publishing.
[22] "Chennareddy, R. K. (2020). Engineering Intelligence Systems Using Big Data and Cloud Architectures for Modern Data Intensive Applications. International Journal of AI, BigData, Computational and Management Studies, 1(2), 41-50.
[23] Chennareddy, R. K. (2021). Designing Data and Analytics Ecosystems for High Volume Transaction Processing Applications. International Journal of AI, BigData, Computational and Management Studies, 2(2), 95-106.
[24] Sethuraman, P., & Chennareddy, R. K. (2022). Machine Learning Assisted Design of Wireless Access Systems for Reliable and Low-Latency Financial and Smart Commerce Services. International Journal of Artificial Intelligence, Data Science, and Machine Learning, 3(4), 133-142.
[25] Sethuraman, P., & Chennareddy, R. K. (2022). Intelligent Vehicular Traffic Flow Prediction Using Learning-Based Spatio-Temporal Models for Data-Driven Wireless Transportation and Urban Analytics Systems. International Journal of Emerging Trends in Computer Science and Information Technology, 3(2), 111-121.
[26] Sethuraman, P. (2022). Latency-Aware Scheduling and Resource Control Algorithms for Emergency and Public Safety Wireless Networks. International Journal of Emerging Research in Engineering and Technology, 3(4), 133-140.