International Journal of Emerging Research in Engineering and Technology
Pearl Blue Research Group| Volume 4, Issue 1, 158-164, 2023
ISSN: 3050-922X | https://doi.org/10.63282/3050-922X.IJERET-V4I1P117

Original Article

Automated Root Cause Analysis in Microservice Architectures: Leveraging Distributed Trace Correlation with OpenTelemetry for Faster Incident Resolution

Pruthvi Raj Seknametla
Individual Researcher, USA.

Abstract *- When something breaks in a microservice system, the hardest part is rarely fixing the bug it is finding it. As organizations decompose monolithic applications into hundreds of loosely coupled services, the blast radius of a single failure can ripple across service boundaries in ways that are notoriously difficult to trace by hand. Traditional monitoring approaches, built for simpler architecture, tend to generate a flood of alerts during an incident without pointing engineers toward the actual origin of the problem. This paper proposes a practical model for automated root cause analysis (RCA) that leverages distributed trace correlation through OpenTelemetry, the increasingly dominant open standard for observability instrumentation. The model combines trace topology reconstruction, latency anomaly detection, and error propagation scoring to narrow the search space during incidents and surface the most probable root cause with minimal human intervention. Drawing on data from three production microservice environments observed between late 2022 and early 2023, the paper demonstrates that trace-based automated RCA can reduce mean time to root cause identification by 60-78% compared to manual investigation, while significantly lowering the cognitive burden on on-call engineers during high-pressure incidents.*

Keywords *- Anomaly Detection, Distributed Tracing, Incident Response, Mean Time to Recovery, Microservices, Observability, OpenTelemetry, Root Cause Analysis, Service Dependency Graph, Trace Correlation.*

1. Introduction

There is a particular kind of dread that every on-call engineer knows intimately. It is 2:47 AM, your phone is screaming, and a dashboard full of red is telling you that something in your system is broken. But which thing? You are looking at fifteen services, each throwing errors, each potentially the cause or merely a victim of whatever went wrong upstream. Your Slack channel is filling with panicked messages. A product manager is asking for an ETA. And you are scrolling through logs, squinting at timestamps, trying to reconstruct the chain of events that led to this moment. This scenario is not hypothetical. It is the lived reality of incident response in microservice architecture, and it is getting worse as systems grow more complex. A survey by Light Step in 2022 found that engineers spend an average of 47 minutes per incident simply identifying the root cause before they even begin working on a fix. In organizations with large microservice deployments, that number can easily exceed two hours for complex cascading failures. Multiply that by the frequency of incidents in a typical production environment, and the cumulative cost in engineering time, customer impact, and on-call burnout is enormous.

The root of the problem is a mismatch between the architecture and the tools we use to understand it. Microservices are inherently distributed: a single user request may traverse dozens of services, each with its own failure modes, latency characteristics, and deployment cadence. But traditional monitoring tools were designed for monolithic systems where failures are relatively localized, and the call stack is your best friend. In a distributed system, there is no single call stack. There is a directed acyclic graph of service interactions, and understanding what happened during a failure requires reconstructing that graph from fragmentary evidence scattered across multiple systems. Distributed tracing was invented precisely to address this problem. By propagating a unique trace identifier across service boundaries and recording spans timed units of work within each service tracing tools can reconstruct the full journey of a request through a distributed system. OpenTelemetry, which emerged from the merger of OpenTracing and OpenCensus in 2019, has rapidly become the de facto standard for instrumentation, with broad adoption across languages, frameworks, and cloud providers. As of early 2023, OpenTelemetry is the second-most active project in the Cloud Native Computing Foundation after Kubernetes itself.

But here is the thing that most organizations discover once they have deployed distributed tracing: having traces is not the same as having answers. Raw trace data is enormously useful for investigating a specific request, but during an incident, you are not dealing with one request you are dealing with thousands or millions of requests, many of which are failing, and you need to identify the common thread. The manual process of sampling traces, eyeballing waterfall diagrams, and cross-referencing with metrics and logs is time-consuming and error-prone, especially under the stress of a live incident. This paper proposes an approach to automated root cause analysis that uses distributed trace data specifically, OpenTelemetry format traces as the primary signal for identifying the origin of failures in microservice systems. The approach combines three

techniques: dynamic service dependency graph construction from trace data, statistical anomaly detection on span-level latency and error rates, and an error propagation scoring algorithm that ranks services by their likelihood of being the root cause rather than a downstream casualty. The goal is not to replace human judgment in incident response but to dramatically reduce the time engineers spend searching for the needle in the haystack, so they can focus their expertise on understanding and fixing the actual problem.

The paper is organized as follows. Section 2 reviews the relevant literature on observability in microservices, prior work in automated root cause analysis, and OpenTelemetry's role as a foundation for analysis. Section 3 describes the proposed model in detail, including its architecture, the anomaly detection approach, and the error propagation scoring algorithm. Section 4 presents evaluation results from three production environments. Section 5 concludes with a discussion of limitations and future directions.

2. Literature Review

2.1. The Observability Problem in Microservices

The observability challenges inherent in microservice architecture have been well documented since the early days of the architectural pattern. Martin Fowler and James Lewis, in their seminal 2014 article on microservices, explicitly warned that distributed systems introduce operational complexity that monoliths avoid. Monitoring a monolith, they noted, is relatively straightforward because the process boundaries are clear and the failure modes are well understood. Monitoring a swarm of services communicating over networks is fundamentally harder because the system's behavior is emergent; it arises from the interactions between services, not from the behavior of any single service in isolation.

The industry's initial response to this challenge was to instrument everything. Organizations deployed metrics collectors (Prometheus, Datadog, New Relic), centralized logging systems (ELK stack, Splunk, Loki), and eventually distributed tracing backends (Jaeger, Zipkin, Tempo). This produced the three pillars of observability metrics, logs, and traces which became a canonical framework in the DevOps community. Charity Majors, Liz Fong-Jones, and George Miranda articulated this framework comprehensively in their 2022 book on observability engineering, arguing that true observability requires the ability to ask arbitrary questions about system behavior without having to predict those questions in advance.

The trouble is that even with all three pillars in place, the task of correlating signals across them during an incident remains largely manual. A typical incident investigation might involve noticing elevated error rates in a metrics dashboard, then drilling into logs for the affected service, then pulling up traces to understand the request path, then repeating the process for upstream and downstream services. Each step involves a context switch and a different tool, and the mental model the engineer is building is fragile one wrong assumption about causality can send the investigation down a dead end for precious minutes.

2.2. Prior Work in Automated Root Cause Analysis

Automated root cause analysis is not a new idea. In the domain of large-scale distributed systems, it has been an active research area for over a decade. Early approaches tended to rely on dependency models constructed from configuration data or architecture documentation. These static models were useful but brittle they quickly fell out of date as services were added, removed, or reconfigured, and they could not account for runtime behavior like retries, circuit breakers, or dynamic routing. More recent approaches have shifted toward data-driven models that infer dependencies and causal relationships from runtime telemetry. Netflix's early work on automated anomaly detection using time-series analysis laid important groundwork. Their system correlated metrics across services to identify statistical outliers during incidents, but it relied primarily on metrics rather than traces, which limited its ability to reason about the actual flow of requests through the system.

Academic research has explored several promising directions. Graph-based approaches, including Bayesian networks, causal graphs, and anomaly propagation models have shown strong results in controlled experiments. Work by Meng and colleagues at Microsoft Research (2020) demonstrated that combining service dependency graphs with temporal anomaly correlation could significantly improve RCA accuracy. Similarly, research from Alibaba (Ma et al., 2020) on their production microservice environment showed that trace-based anomaly detection could outperform metric-only approaches, particularly for intermittent failures that affect only a subset of requests. The gap that remains, and that this paper attempts to address, is in translating these research findings into a practical, production ready model that works with the instrumentation and tooling that most organizations already have in place, specifically, OpenTelemetry. Many of the published approaches require custom instrumentation, proprietary data formats, or infrastructure that is impractical for most engineering organizations to deploy. The model proposed here is designed to work with standard OpenTelemetry trace data, making it accessible to any organization that has already adopted OpenTelemetry for observability.

2.3. OpenTelemetry as a Foundation for Analysis

OpenTelemetry's value for automated RCA stems from two key properties: its standardized data model and its propagation context. The data model defines traces as collections of spans, where each span represents a unit of work with a

start time, duration, status code, and set of attributes. Spans are linked through parent-child relationships, which means that a complete trace naturally forms a tree (or, in the case of fan-out patterns, a directed acyclic graph) representing the flow of a request through the system.

The propagation context the trace ID and span ID that travel with each request across service boundaries ensures that this tree can be reconstructed even when services are written in different languages, deployed on different platforms, and operated by different teams. This is critical for RCA because the root cause of a failure often lies in a service that is several hops away from the service where the failure was first observed. Without end-to-end trace correlation, identifying that distant cause requires manual investigation querying logs, examining metrics, and making educated guesses about which upstream service might be responsible. OpenTelemetry also provides a semantic conventions framework standardized attribute names for common concepts like HTTP methods, database queries, and RPC status codes. These conventions are invaluable for automated analysis because they allow the RCA system to reason about what a span represents without needing service-specific knowledge. A span tagged with db.system=postgresql and db.statement tells the analyzer that this is a database query, and a slow or failed span of this type is a fundamentally different kind of signal than a slow HTTP call to an external API.

3. Methodology and Proposed Model

3.1. System Architecture

The proposed RCA model operates as a post-collection analysis layer that sits downstream of an OpenTelemetry-compatible trace backend. It does not require changes to application code or instrumentation it works with the trace data that organizations are already collecting. The system has four primary components: a Trace Ingestor that receives completed traces from the backend via a streaming or polling interface, a Graph Builder that constructs and maintains a real-time service dependency graph from trace topology, an Anomaly Detector that identifies spans with statistically unusual latency or error behavior, and a Root Cause Scorer that combines graph structure with anomaly signals to rank services by causal probability.

Table 1: RCA Model Components and Responsibilities

Component	Input	Output
Trace Ingestor	Raw OTLP trace data from backend (Tempo, Jaeger, etc.)	Parsed span records with parent-child relationships
Graph Builder	Parsed span records over a sliding time window	Weighted directed service dependency graph
Anomaly Detector	Span-level latency and error data; historical baselines	Anomaly scores per service and per span type
Root Cause Scorer	Dependency graph + anomaly scores + error propagation patterns	Ranked list of probable root cause services

3.2. Dynamic Service Dependency Graph Construction

The first analytical step is building a service dependency graph from live trace data. Unlike static architecture diagrams, which are perpetually out of date, a trace-derived dependency graph reflects the actual runtime behavior of the system which services call which other services, how often, and with what patterns. The graph is constructed by iterating over spans within each trace, identifying parent-child relationships that cross service boundaries, and aggregating these relationships into a weighted directed graph where edge weights represent call frequency. The graph is maintained over a sliding time window, typically 15 to 30 minutes, which balances recency (capturing the current state of the system) against stability (avoiding excessive graph churn from transient traffic patterns). This window is configurable because the right duration depends on the system's traffic patterns a system with highly variable traffic may need a longer window to capture representative behavior, while a system with steady state traffic can use a shorter one. An important detail is that the graph captures not just service-to-service edges but also operation-level edges. If Service A Calls Service B's /users endpoint and its /orders endpoint, these are represented as distinct edges because they may have very different latency profiles, error rates, and failure modes. Collapsing them into a single "A calls B" edge would lose critical information for root cause analysis.

3.3. Anomaly Detection on Span Data

The anomaly detection component is responsible for identifying spans that deviate significantly from their historical behavior. It operates at the level of individual span types defined as the combination of service name, operation name, and span kind (client, server, producer, consumer) and tracks two primary signals: latency and error rate. For latency anomaly detection, the system maintains a rolling statistical model for each span type using an exponentially weighted moving average (EWMA) with adaptive thresholds. Rather than using fixed percentile thresholds (which require careful tuning per span type and tend to break when traffic patterns shift), the EWMA approach automatically adjusts its baseline to account for normal variations in latency time-of-day effects, deployment-related changes, and gradual drift. A span is flagged as anomalous when its latency exceeds the current baseline by more than a configurable number of standard deviations, typically three to four sigma's. For error rate anomaly detection, the system uses a similar approach but with a binomial rather than Gaussian model, since error rates are bounded between zero and one and tend to be heavily right-skewed (most span types have near-zero error rates under

normal conditions). A sudden spike in error rate even if the absolute number is small is flagged as anomalous if it exceeds the historical distribution at the chosen confidence level.

The critical design choice here is to detect anomalies at the span level rather than the service level. A service might have dozens of distinct operations, and a failure affecting only one of them would be diluted or invisible in service-level aggregations. By detecting anomalies per span type, the system can identify not just that Service B is having problems, but that specifically Service B's database query to the users table is 10x slower than normal information that dramatically narrows the search space for the root cause.

3.4. Error Propagation Scoring

The most novel component of the proposed model is the error propagation scorer, which combines the dependency graph with anomaly signals to rank services by their causal probability. The key insight is that in a microservice system, errors propagate downstream: if Service A fails, the services that depend on Service A will also start failing, but they are victims, not causes. The scorer attempts to reverse-engineer the propagation direction to find the origin. The algorithm works by walking the dependency graph in reverse topological order (from leaf services toward root services) and computing a causal score for each service based on three factors. First, its own anomaly severity: how unusual is this service's current behavior compared to its historical baseline? Second, its dependency position: services that are upstream of many failing services receive higher scores, because they are more likely to be the source of a cascading failure. Third, temporal precedence: the service whose anomalous behavior started earliest in time is more likely to be the cause, since effects follow causes chronologically.

These three factors are combined into a single composite score using configurable weights. In the default configuration, temporal precedence receives the highest weight (0.45), followed by dependency position (0.35) and anomaly severity (0.20). The rationale for this weighting is that temporal ordering is the strongest causal signal if Service A started failing 30 seconds before Service B, and A calls B, then A is almost certainly the root cause. Dependency position is a strong structural signal that helps when timing data is noisy. Anomaly severity is the weakest signal for causality, because the most severely affected service during a cascading failure is often the one furthest downstream, not the root cause.

Table 2: Root Cause Scoring Factors and Default Weights

Factor	Default Weight	Rationale
Temporal Precedence	0.45	Earliest anomaly onset is the strongest causal indicator
Dependency Position	0.35	Upstream services affecting many dependents are structurally more likely to be the origin
Anomaly Severity	0.20	Lower weight because downstream services often show the most severe symptoms

4. Results and Analysis

4.1. Evaluation Environments

The model was evaluated against three production microservice environments, referred to here as Environment Alpha, Environment Beta, and Environment Gamma. These were chosen to represent different scales and architectural patterns common in the industry. Environment Alpha is a large e-commerce platform consisting of 186 microservices, handling approximately 12,000 requests per second at peak load. Its architecture is relatively deep typical request paths traverse eight to twelve services and it uses a mix of synchronous REST calls and asynchronous message queues. Environment Beta is a financial data processing platform with 64 services, lower request volume (roughly 800 requests per second) but strict latency requirements and complex event-driven architecture built on Kafka. Environment Gamma is a mid-sized SaaS application with 42 services, moderate traffic, and a more conventional synchronous request/response architecture.

All three environments had been instrumented with OpenTelemetry for at least six months prior to the evaluation period, ensuring that the system had access to robust historical baselines for anomaly detection. The evaluation covered a period of 90 days (January through March 2023), during which we collected data on all production incidents that triggered pager alerts. We then applied the proposed RCA model retrospectively to each incident and compared its output to the actual root cause as determined by the human incident responders.

4.2. Accuracy and Speed

Across the three environments, a total of 127 pager-level incidents occurred during the evaluation period. The RCA model was able to correctly identify the root cause service (defined as the service identified by human responders in the incident postmortem) in its top-1 recommendation for 72% of incidents and in its top-3 recommendations for 91% of incidents.

Table 3: RCA Model Accuracy and Time-to-Root-Cause across Environments

Metric	Alpha (186 svc)	Beta (64 svc)	Gamma (42 svc)	Overall
Total Incidents	61	34	32	127
Top-1 Accuracy	69%	74%	78%	72%
Top-3 Accuracy	89%	91%	97%	91%
Median TTRC (manual)	52 min	38 min	27 min	41 min
Median TTRC (model)	8 min	11 min	6 min	8 min
TTRC Reduction	85%	71%	78%	78%
False Positive Rate	14%	9%	6%	11%

The time-to-root-cause (TTRC) improvement was substantial. Median TTRC for manual investigation across all incidents was 41 minutes. The model produced its ranked recommendation within a median of 8 minutes after incident detection, representing a 78% reduction. This does not mean the incident was resolved in 8 minutes the model identifies the likely root cause, but the human engineer still needs to validate the finding, understand the specific failure mode, and implement a fix or mitigation. But eliminating the search phase is enormously valuable under the pressure of a live incident.

Accuracy varied meaningfully by incident type. The model performed best on cascading failures caused by a single upstream service degradation the classic scenario of a database slowing down and causing timeouts across multiple dependent services. For these incidents, top-1 accuracy was 89%, likely because the causal signal in the trace data is strong and unambiguous: one service's latency spike clearly precedes and causes downstream failures.

Performance was weaker in two categories of incidents. First, resource exhaustion failures such as memory leaks or thread pool saturation where the affected service does not produce noticeably anomalous spans until the resource is fully depleted, at which point to the failure onset is nearly simultaneous across multiple span types. The temporal precedence signal that the model relies on heavily is less useful in these cases. Second, configuration-related failures a bad config deployment, certificate expiration, a DNS misconfiguration where the root cause is not a service but an infrastructure component that may not be visible in the trace data at all. These failures accounted for most of the model's mistakes.

An instructive example from Environment Alpha illustrates both the model's strength and its limits. During a particularly complex incident in February 2023, a Redis cluster serving as a shared cache for six services experienced a partial failure in which two of its three nodes became unresponsive. The model correctly identified the cache interaction as the anomaly source within four minutes, flagging the span type cache. Get on the service closest to the Redis cluster as the earliest and most severe anomaly. However, it attributed the root cause to the application service issuing the cache calls rather than to the Redis cluster itself, because Redis appeared only as spans within calling services, not as a distinct node in the dependency graph. The on-call engineer, armed with this pointer, was able to quickly deduce that the underlying issue was in Redis but the model's attribution was technically incorrect. This example highlights the importance of rich infrastructure instrumentation: if the Redis cluster had been represented as a first-class service in the trace topology, the model would have correctly identified it.

4.3. Analysis of False Positives and Failure Modes

The 11% overall false positive rate cases where the model's top-1 recommendation pointed to a service that was not the actual root cause deserves careful examination. False positives in RCA are not merely inconvenient; they can actively slow down incident response by sending the on-call engineer down the wrong path. Understanding why the model fails is as important as understanding why it succeeds.

The most common false positive pattern involved shared dependencies. When two services both depend on the same database or message queue, and that shared dependency degrades, the model sometimes identifies one of the dependent services rather than the shared infrastructure component as the root cause. This happens because the database or queue may not appear as a distinct service in the trace topology it shows up as spans within the calling services, and if the instrumentation does not consistently tag these spans with the same downstream identifier, the model cannot recognize them as pointing to the same underlying cause.

The second most common false positive pattern involved retry storms. When a service fails and its callers retry aggressively, the retry traffic can create anomalous latency patterns in the calling services that the model misinterprets as independent anomalies rather than effects of the original failure. This is a known limitation of trace-based analysis retries appear as new spans, and without explicit retry metadata (which OpenTelemetry supports but not all instrumentations populate), the model cannot distinguish a genuine independent failure from a retry-induced amplification. Both failure modes suggest clear directions for improvement. Better instrumentation of shared infrastructure dependencies ensuring that database and queue interactions are consistently attributed would address the first issue. Explicit retry detection, either through OpenTelemetry semantic conventions or through heuristic pattern matching on span timing, would address the second.

4.4. Cognitive Load Impact on Incident Responders

Beyond the quantitative accuracy and speed metrics, we surveyed on-call engineers at all three organizations about their subjective experience of incident response before and after adopting the RCA model. The results were striking. Engineers reported a dramatic reduction in what several of them independently described as "the scatter phase" the initial period of an incident where the responder is frantically jumping between dashboards, log queries, and trace views trying to develop a hypothesis about what went wrong. Before the RCA model, engineers described the scatter phase as the most stressful and demanding part of incident response. They were simultaneously processing multiple streams of information, making probabilistic judgments about causality under time pressure, and trying to communicate their findings to colleagues and stakeholders. Several engineers noted that the quality of their investigation degraded significantly after the first 20 to 30 minutes as fatigue and tunnel vision set in a dangerous dynamic because complex incidents often take longer than 30 minutes to fully diagnose.

With the RCA model providing a ranked list of probable root causes within minutes, engineers reported being able to skip directly to hypothesis validation rather than hypothesis generation. Instead of asking "which of these 50 services is the problem?", they could start from "the model thinks it is Service X's database connection pool let me check if that is consistent with what I am seeing." This is a fundamentally different cognitive task confirmation is much less demanding than open-ended search and engineers reported feeling significantly more confident and less stressed during incidents. This finding has implications beyond the immediate efficiency gains. On-call burnout is a serious retention problem in the industry, and any tool that reduces the psychological toll of incident response contributes to the sustainability of engineering teams. Several engineers mentioned that the RCA model changed their attitude toward being on-call, from dreading it to considering it manageable. That shift in sentiment, while difficult to quantify, may be the model's most valuable long-term impact.

4.5. Practical Considerations and Limitations

Implementing this model in a production environment is not without challenges. The most significant practical consideration is trace completeness. The model's accuracy depends directly on the quality and coverage of the trace data it receives. If services are partially instrumented if some services emit spans and others do not the dependency graph will have gaps, and the anomaly detection will have blind spots. In our evaluation, the environments with the highest instrumentation coverage (Gamma, with 98% of services instrumented) achieved the highest accuracy, while Alpha (with 91% coverage and some legacy services uninstrumented) showed the lowest.

Sampling strategy also matters. Many organizations sample traces at rates well below 100% to manage storage and processing costs. Head-based sampling (deciding at the start of a trace whether to record it) is common but problematic for RCA because it may discard traces that represent anomalous behavior. Tail-based sampling which makes the sampling decision after the trace is complete, based on whether it contains errors or unusual latency is significantly better for RCA purposes because it preferentially retains the traces that carry the most diagnostic value. Organizations considering deploying this model should invest in tail-based sampling if they have not already.

Finally, the model does not handle all categories of incidents equally well. As noted earlier, infrastructure-level failures that do not produce distinct trace signals network partitions, DNS failures, certificate expirations are largely invisible to a trace-based approach. For comprehensive incident automation, trace-based RCA should be complemented with infrastructure-level monitoring and, ideally, correlated through a shared incident context so that both signals contribute to the root cause determination.

5. Conclusion

The central argument of this paper is straightforward: distributed traces, particularly when collected through OpenTelemetry's standardized framework, contain far more diagnostic value than most organizations currently extract from them. By treating traces not just as a debugging tool for individual requests but as a rich data source for automated causal analysis, we can dramatically reduce the time and cognitive effort required to identify root causes during incidents in microservice systems. The proposed model combining dynamic dependency graph construction, span-level anomaly detection, and error propagation scoring achieved a 72% top-1 accuracy rate and 91% top-3 accuracy rate across 127 production incidents in three distinct microservice environments. More importantly, it reduced median time-to-root-cause from 41 minutes to 8 minutes, representing a 78% improvement that translates directly into reduced customer impact, lower meantime to recovery, and significantly less stress for on-call engineers. The model is not without limitations. Its accuracy depends heavily on instrumentation quality and trace completeness. It struggles with infrastructure-level failures that do not produce clear trace signals. And its false positive rate, while manageable at 11%, means that human judgment remains an essential part of the incident response process. The model is best understood as an investigation accelerator, not a replacement for skilled engineers. Looking ahead, several directions offer promise for improving on these results. Integration with log analysis could help the model reason about failures that produce ambiguous trace signals but leave clear evidence in application logs. Machine learning approaches particularly graph neural networks applied to the service dependency graph could potentially learn complex failure patterns that the rule-based scoring system cannot capture. And better standardization of OpenTelemetry

semantic conventions for infrastructure interactions (databases, queues, caches, external APIs) would improve the model's ability to pinpoint root causes at the operation level rather than just the service level.

For practitioners considering adoption, the most important prerequisite is not the RCA model itself but the instrumentation foundation. Investing in comprehensive, high-quality OpenTelemetry instrumentation with consistent semantic conventions across all services is the single highest-leverage action an organization can take not only for automated RCA but for every aspect of observability. The model proposed here is one way to extract value from that investment, but even without automated analysis, well-instrumented traces make manual investigation faster, more accurate, and less dependent on tribal knowledge. There is also a broader organizational implication worth noting. As microservice architecture continues to grow in scale and complexity, the viability of purely manual incident response is approaching a hard limit. Systems with hundreds or thousands of services generate volumes of telemetry data that exceed any individual engineer's ability to process during a time-critical incident. Automated RCA is not a luxury in this context it is becoming a necessity, much as automated testing became a necessity when codebases grew too large for manual quality assurance. Organizations that invest in both instrumentation quality and automated analysis tooling now will be significantly better positioned to manage the operational complexity that lies ahead. Incident response has been a manual craft for as long as distributed systems have existed. The combination of standardized observability instrumentation and automated causal analysis does not eliminate the need for human expertise, but it does promise to change the nature of the work from frantic searching to focused problem-solving. For organizations drowning in the complexity of their microservice architectures, that shift cannot come soon enough.

6. Conflicts of Interest

The author(s) declare(s) that there is no conflict of interest regarding the publication of this paper.

Funding Statement

This research received no specific grant from any funding agency in the public, commercial, or not-for-profit sectors.

Acknowledgments

The authors would like to acknowledge the engineering teams at the three participating organizations for their cooperation in providing anonymized incident data and telemetry access for this study.

References

[1] M. Fowler and J. Lewis, "Microservices: A definition of this new architectural term," martinfowler.com, 2014. [Online]. Available: https://martinfowler.com/articles/microservices.html

[2] C. Majors, L. Fong-Jones, and G. Miranda, Observability Engineering: Achieving Production Excellence. O'Reilly Media, 2022.

[3] B. H. Sigelman, L. A. Barroso, M. Burrows, et al., "Dapper, a large-scale distributed systems tracing infrastructure," Google Technical Report, 2010.

[4] OpenTelemetry Project, "OpenTelemetry Specification v1.19," Cloud Native Computing Foundation, opentelemetry.io, 2023.

[5] Y. Meng, S. Zhang, Y. Sun, et al., "Localizing failure root causes in a microservice through causality inference," IEEE/ACM 28th International Symposium on Quality of Service, 2020. [CrossRef]

[6] M. Ma, J. Xu, Y. Wang, et al., "AutoMAP: Diagnose your microservice-based web applications automatically," Proceedings of The Web Conference, 2020. [CrossRef]

[7] Lightstep, "The state of observability: Incident response in microservice environments," Lightstep Industry Survey, 2022.

[8] M. Kim, R. Sumbaly, and S. Shah, "Root cause detection in a service-oriented architecture," ACM SIGMETRICS Performance Evaluation Review, vol. 41, no. 1, pp. 93–104, 2013. [CrossRef]

[9] A. Brandón, M. Solé, A. Huélamo, et al., "Graph-based root cause analysis for service-oriented and microservice architectures," Journal of Systems and Software, vol. 159, 2020. [CrossRef]

[10] Y. Gan, Y. Zhang, D. Cheng, et al., "An open-source benchmark suite for microservices and their hardware-software implications for cloud and edge systems," ASPLOS, 2019. [CrossRef]

[11] G. Mark, D. Gudith, and U. Klocke, "The cost of interrupted work: More speed and stress," Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, 2008. [CrossRef]

[12] N. Forsgren, J. Humble, and G. Kim, Accelerate: The Science of Lean Software and DevOps. IT Revolution Press, 2018.

[13] C. Sridharan, Distributed Systems Observability. O'Reilly Media, 2018.