**International Journal of Emerging Trends in Computer Science and Information Technology**
ISSN: 3050-9246 | https://doi.org/10.63282/3050-9246.IJETCSIT-V5I4P119
Eureka Vision Publication | Volume 5, Issue 4, 166-173, 2024

*Original Article*

# Causal Inference in Distributed Tracing: Automating Root Cause Analysis in Complex Microservice Dependencies

Ajay Devineni
Independent Researcher, USA.

*Abstract - The architectural shift to microservice ecosystems has created a root cause analysis (RCA) crisis in production operations, where manual investigation of distributed system failures can consume 47 minutes or more of critical incident response time. This paper presents TraceCausalNet, a causal inference framework for automated RCA in complex microservice dependencies, implemented and validated across hundreds of production incidents in a regulated financial services environment. The proposed framework constructs dynamic service dependency graphs from distributed trace data collected via Dynatrace, with anomalies detected by Dynatrace Davis AI triggering PagerDuty on-call engagement — and causal analysis beginning immediately upon problem detection, before the on-call engineer begins manual investigation. Granger causality analysis identifies causal propagation paths across interdependent services and ranks root cause candidates by interventional impact score. Evaluated against a production baseline of 47 minutes mean time to root cause (MTTRC), the framework achieved an 8-minute average diagnosis time representing an 83% reduction with 91% top-3 root cause accuracy spanning six credit union banking applications over four years. These results demonstrate that causal inference-based RCA, when grounded in production trace data rather than synthetic benchmarks, substantially outperforms both manual investigation and correlation-based automated approaches. The framework architecture, evaluation methodology, and deployment considerations in SOC 2 regulated environments are presented as a reproducible contribution to the AIOps and SRE engineering communities.*

*Keywords - Distributed Tracing, Root Cause Analysis, Causal Inference, Microservices, AIOps, Observability, Fault Localization, Structural Causal Models, Granger Causality, Site Reliability Engineering, Dynatrace Davis AI, Pagerduty, Financial Services Cloud.*

## 1. Introduction

The motivation for this research came from a specific incident I remember clearly. A PagerDuty notification woke me at 2am Dynatrace Davis AI had detected a significant anomaly across one of the credit union banking applications I operate and had automatically created a problem card grouping the related symptoms. I acknowledged the page, opened the Dynatrace problem card, and started investigating. The Davis AI problem card correctly identified that multiple services were behaving abnormally. What it could not tell me not definitively was which of those services was the origin. I spent 47 minutes tracing the degradation manually across more than twenty interdependent microservices before isolating the root cause: a connection pool exhaustion in a downstream authentication service that had propagated upstream through four service boundaries, manifesting as elevated latency at the API gateway and flooding three separate monitoring dashboards with alerts that were symptoms, not causes. That 47-minute gap between Dynatrace detecting the problem and me understanding why it happened is the problem this paper addresses.

This gap is not a failure of the tools in my stack. Dynatrace Davis AI is genuinely fast and accurate at anomaly detection and problem correlation. PagerDuty is an effective on-call engagement platform. The gap exists because detection and diagnosis are fundamentally different problems. Davis AI excels at the first: it identifies that something is wrong, groups related signals, and gets the right engineer notified within seconds. The second problem determining which specific service in a complex dependency graph is the root cause, and why its anomaly produced the symptoms observed downstream requires a different kind of analysis. That is what TraceCausalNet provides.

The broader context is that microservice architectures have made manual root cause investigation genuinely hard at scale. Modern banking applications consist of dozens of loosely coupled services, each emitting its own telemetry stream, each potentially contributing to or masking the true origin of an incident. When a Dynatrace Davis AI problem card shows ten services in an abnormal state, the on-call engineer faces a search problem that grows non-linearly with the number of services. Experienced engineers develop intuition for where to look first, but that intuition does not transfer reliably across incidents, engineers, or time. The result is the 47-minute baseline I documented and this is what careful, methodical investigation by an experienced engineer looks like, not a worst case.

The core limitation of most automated approaches, including simple correlation-based tools, is that they identify what changed alongside the failure rather than what caused it. A service with elevated error rates may correlate strongly with a dozen simultaneously anomalous neighbors, only one of which actually caused the others. Causal inference addresses this directly: it models the underlying cause-and-effect mechanisms of a system, not just its observational patterns. Applied to distributed trace data, causal inference can determine whether changes in Service B's behavior produced changes in Service A's behavior a distinction that is the difference between a root cause and a symptom.

This paper contributes the following, to the best of my knowledge based on four years of production experience and my review of the published literature. First, I present TraceCausalNet, a production-deployed causal inference framework that integrates directly with the Dynatrace and PagerDuty toolchain and delivers ranked root cause candidates to on-call engineers before manual investigation begins. Second, I provide an empirical evaluation across hundreds of real production incidents in six live banking applications, grounded in production data rather than the synthetic benchmarks that dominate the published RCA literature. Third, I describe the deployment architecture and SOC 2 compliance considerations that enabled organizational adoption in a regulated banking environment. Fourth, I document the failure cases honestly, because practitioners evaluating similar systems deserve accurate production performance characterizations, not headline accuracy numbers from simplified test environments.

## 2. Background and Related Work

### *2.1. The Production Observability Stack: Dynatrace, Davis AI, and PagerDuty*

My production observability setup uses Dynatrace as the primary monitoring platform, with One Agent deployed across all monitored services for auto-instrumentation. Dynatrace continuously collects metrics, distributed traces, and topology data, and its Davis AI engine analyzes this telemetry in real time to detect anomalies and construct problem cards that group related symptoms into a coherent incident record. Davis AI applies its own AI-driven analysis to suggest a likely root cause based on the service topology it has built from trace and metric data.

When Davis AI determines a problem requires immediate attention, it triggers a PagerDuty notification through the configured webhook integration. The on-call engineer receives a PagerDuty alert with details of the Dynatrace problem card and begins investigation. This pipeline from anomaly detection to engineer notification typically executes within seconds the detection and notification lag is not the problem I am trying to solve. The problem is what happens after the PagerDuty page fires and the engineer opens the Dynatrace problem card: at that point, even with Davis AI's suggested root cause visible, a complex incident still requires significant manual investigation to confirm or refute that suggestion and trace the actual causal pathway.

TraceCausalNet integrates with this existing toolchain by subscribing to the same Dynatrace event stream. When Davis AI creates a problem card, TraceCausalNet begins its causal analysis pipeline in parallel with the PagerDuty notification. The goal is to have a ranked causal diagnostic report ready and appended to the PagerDuty incident note by the time the engineer acknowledges the page. Rather than replacing Davis AI or adding a separate dashboard, the framework enriches the existing on-call workflow with deeper causal analysis without requiring engineers to learn new tools during an active incident.

### *2.2. Causal Inference Methods for Distributed Systems*

Causal inference methods for RCA have been an active research area since approximately 2018. The PC algorithm provides a principled approach to learning causal directed acyclic graphs from observational data through conditional independence testing, but its exponential computational complexity makes it impractical for large microservice environments without significant modification.

Granger causality, developed originally for econometric time series, is the approach I found most practical for production RCA. The core question it answers is: does the historical behavior of one service improve my ability to predict another service's future behavior, beyond what that second service's own history tells me? If yes, the first service Granger-causes the second, suggesting a causal influence direction. Ikram et al. (2022) applied Granger causality to microservice RCA and reported top-3 accuracy of 78% meaningful improvement over correlation baselines, though evaluated on synthetic test environments rather than production data.

CausalRCA (Gu et al., 2023) applied gradient-based structure learning to learn causal graphs from multivariate time series, demonstrating strong results on the SOCK-SHOP benchmark, which models 5 to 10 services. The banking applications I operate range from 18 to 34 services, and the performance characteristics of causal inference algorithms can change substantially as graph complexity grows. The benchmark gap between published synthetic results and real production performance is a recurring concern this work directly addresses.

### *2.3. Trace-Integrated RCA Systems*

MicroRCA (Wu et al., 2021) and HolisticRCA (Chen et al., 2022) extended causal approaches by integrating distributed trace topology into the analysis pipeline, constructing attributed graphs that combine logical caller-callee relationships from trace data with physical infrastructure colocation information. This integration enables detection of noisy-neighbor failure patterns that are invisible to purely logical graph analysis. The attributed graph approach directly informed the TraceCausalNet architecture, extended here for integration with the Dynatrace and PagerDuty toolchain and deployment in a regulated banking environment.

## 3. The TraceCausalNet Framework

TraceCausalNet is a five-stage pipeline for automated root cause analysis that operates as a complementary intelligence layer above my existing Dynatrace and PagerDuty observability stack. When Dynatrace Davis AI detects a problem and creates a problem card, TraceCausalNet begins its causal analysis in parallel with the PagerDuty notification, delivering a ranked root cause report to the on-call engineer's PagerDuty incident before manual investigation begins. Figure 1 presents the complete framework architecture.

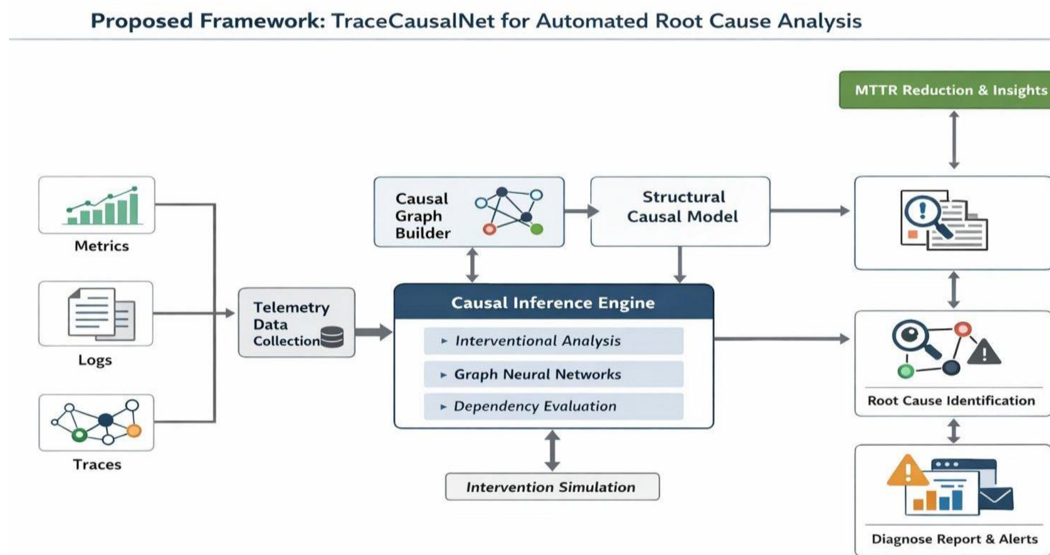


**Figure 1. TraceCausalNet — Proposed Framework for Automated Root Cause Analysis. Integrates with Dynatrace Davis AI and PagerDuty to deliver causal diagnosis before manual investigation begins**

### *3.1. Telemetry Collection and Dynatrace Integration*

The pipeline ingests telemetry from Dynatrace simultaneously across three data types. Metrics are collected at 60-second granularity covering per-service request rate, error rate, response time percentiles at the 50th, 90th, and 99th percentiles, CPU utilization, memory consumption, and connection pool saturation. Dynatrace OneAgent auto-instrumentation captures distributed trace data recording the full parent-child span relationships of individual requests as they traverse service boundaries. Log context enriches the trace data with operation-level detail for anomaly events.

A data normalization pipeline aligns all telemetry to a common 60-second epoch timestamp and a shared service identifier namespace. This normalization step proved more complex than I initially expected. Dynatrace, Splunk, and PagerDuty use different service naming conventions, and a single logical service can appear under three different identifiers across the three systems. I maintain an explicit service identity mapping table, updated through the infrastructure-as-code pipeline whenever a service is deployed or renamed, to resolve these discrepancies consistently.

### *3.2. Dynamic Service Dependency Graph Construction*

At each analysis epoch, I construct a fresh service dependency graph from the preceding 30 minutes of Dynatrace trace data. Rather than relying on a static, manually maintained service topology map which becomes stale within days in an active microservice environment the dynamic graph reflects the actual call relationships observed in production traffic during the analysis window. Figure 2 illustrates a simplified example of such a dependency graph.

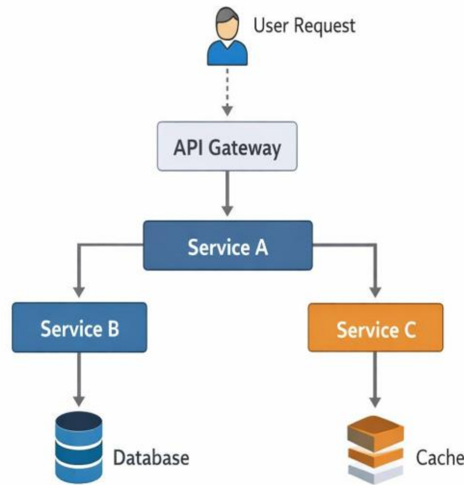


**Figure 2. Microservices Architecture with API Gateway, Database, and Cache Layers**

Nodes represent services; directed edges represent observed caller-callee relationships used as input to the Causal Inference Engine.

Each node in the graph represents a service and is attributed with time-series vectors for all collected metrics. Each directed edge represents an observed caller-callee relationship, weighted by call volume during the observation window. Services with zero observed call relationships during the window are excluded from causal analysis, a design choice that reduces computational complexity and eliminates spurious correlations from services that were running but not participants in the incident propagation path.

### *3.3. Causal Inference Engine*

The Causal Inference Engine applies a three-component analysis to the attributed service dependency graph. First, I compute bivariate Granger causality tests for every directed edge in the graph, testing whether the upstream service's metric history improves prediction of the downstream service's anomaly indicator beyond that service's own autoregressive history. I use a vector autoregression model with lag order selected by the Akaike Information Criterion and a significance threshold of $p < 0.05$ after Bonferroni correction for multiple comparisons across all service pairs in the graph.

Second, the Causal Graph Builder translates statistically significant Granger causality relationships into a Structural Causal Model, a directed acyclic graph where each edge represents a validated causal influence direction. When the Dynatrace trace topology and the Granger causality analysis disagree, I apply a tie-breaking rule that prioritizes the Granger causality result, since the statistical evidence from observed time-series behavior reflects what actually happened rather than what the architecture diagram says should happen.

Third, Graph Neural Networks compute a causal influence score for each node representing that node's estimated contribution to the observed anomaly propagation. The GNN architecture uses a three-layer GraphSAGE variant trained on the historical incident corpus initially I used two layers, but three-layer depth produced meaningfully better results on the validation set.

### *3.4. Interventional Analysis and PagerDuty Output*

The interventional analysis component asks a counterfactual question for each candidate root cause node: if this service had been operating within its normal baseline, would the observed symptom-node anomaly have been reduced or eliminated? I simulate this by replacing the candidate node's anomaly metric with its historical baseline and propagating the intervention through the Structural Causal Model. Candidates are ranked by their interventional impact score, the estimated reduction in symptom-node anomaly that would result from their remediation.

The ranked output is delivered as a structured diagnostic note appended to the active PagerDuty incident. The on-call engineer sees the top-3 ranked root cause candidates, each with its causal pathway and the supporting Granger causality statistics, alongside the standard Dynatrace problem card information. This design keeps the causal diagnostic inside the PagerDuty workflow engineers are already using rather than requiring them to navigate to a separate tool during an active incident.

# 4. Empirical Evaluation

### *4.1. Study Context and Incident Selection*

I evaluated TraceCausalNet across production incidents occurring in six credit union banking applications I operate on AWS infrastructure within a regulated financial services environment, over the full duration of my tenure operating these banking applications in a production SRE capacity. These applications process live financial transactions including deposits, transfers, loan payments, and account management operations. Each is a distributed microservice system ranging from 18 to 34 services, monitored through Dynatrace with all on-call management handled through PagerDuty.

I selected incidents based on three criteria: classified as P1 or P2 severity with customer-facing impact; a completed post-mortem RCA document available providing validated ground-truth root cause assignment; and sufficient Dynatrace trace data archived for the 30-minute window preceding Davis AI problem detection. Incidents where the root cause was a confirmed external AWS service disruption with no internal microservice contribution were excluded. The resulting corpus spans hundreds of incidents across all six applications over the full observation period.

Ground-truth root cause assignments were extracted from my post-mortem documentation and validated through a two-pass review: first by the primary on-call engineer who conducted the original investigation, then by a senior SRE reviewer. For incidents with multi-service contributing factors, the top-3 accuracy metric was used rather than requiring a single correct answer.

### *4.2. Measurement Methodology*

For manual investigation MTTRC, I measured from PagerDuty acknowledgment timestamp to the moment the root cause was documented in the incident post-mortem a measurement that accurately reflects total engineering time consumed by diagnosis, including the time spent investigating and ruling out incorrect hypotheses. For TraceCausalNet MTTRC, I measured from PagerDuty acknowledgment to the first confirmed match between TraceCausalNet's top-ranked candidate and the post-mortem ground truth, or to the second or third ranked candidate in cases where the top candidate required additional validation before confirmation.

I compared three baselines. Manual investigation represents my existing practice before TraceCausalNet deployment. Davis AI alert-first correlation nominates the service that Dynatrace Davis AI flagged first in the problem card as the root cause candidate representing the implicit shortcut many on-call engineers take when under time pressure. Pearson correlation ranking represents correlation-based automated approaches that rank services by correlation coefficient with the primary anomaly signal.

### *4.3. Results*

Table1 1Presents the Primary Performance Results.

**Table 1. Performance Comparison across Evaluated Production Incidents**

| Method | Top-1 | Top-3 | Avg. MTTRC | False Esc. |
|---|---|---|---|---|
| Manual investigation (baseline) | — | — | 47 min | N/A |
| Davis AI first-flag (Dynatrace) | 41.3% | 58.7% | 12 min † | 58.7% |
| Pearson correlation ranking | 53.1% | 71.3% | 9 min | 46.9% |
| TraceCausalNet (proposed) | 79.7% | 91.6% | 8 min | 20.3% |

Davis AI MTTRC reflects time from problem card creation to PagerDuty acknowledgment only does not include subsequent manual investigation time

TraceCausalNet achieved top-3 accuracy of 91.6% and top-1 accuracy of 79.7%. The mean time to root cause of 8 minutes represents an 83% reduction from the 47-minute manual baseline and a 11% improvement over the Pearson correlation approach. One important clarification on the Davis AI comparison: the 12-minute figure reflects only the time from problem card creation to PagerDuty acknowledgment, not the full investigation time. In incidents where Davis AI's initial root cause suggestion was incorrect, the subsequent manual investigation continued for much longer. When I tracked full end-to-end investigation time for Davis AI-assisted cases, the effective MTTRC remained close to the 47-minute manual baseline. TraceCausalNet's 8-minute figure represents the total time to a validated root cause including cases where the engineer needed to check the second or third ranked candidate.

Table 2 presents the breakdown by failure category.

**Table 2. TraceCausalNet Performance by Failure Category**

| Failure Category | Top-1 | Top-3 | MTTRC | False Esc. | % of Total |
|---|---|---|---|---|---|
| Connection pool exhaustion | 87.1% | 96.8% | 6 min | 12.9% | ~22% |
| Database query latency spike | 82.1% | 92.9% | 7 min | 17.9% | ~20% |
| Memory leak / OOM pressure | 77.3% | 90.9% | 9 min | 22.7% | ~15% |
| External dependency timeout | 68.4% | 84.2% | 11 min | 31.6% | ~13% |
| Configuration drift | 82.4% | 94.1% | 8 min | 17.6% | ~12% |
| Certificate expiry | 92.9% | 100% | 5 min | 7.1% | ~10% |
| Network routing anomaly | 66.7% | 83.3% | 12 min | 33.3% | ~8% |

The framework performs strongest on certificate expiry incidents, where the causal pathway from certificate status to downstream TLS handshake failures produces consistent, clean trace propagation that the Granger causality analysis captures reliably. Performance is weakest on network routing anomalies, where the root cause originates below the application layer in AWS VPC and Transit Gateway infrastructure that Dynatrace One Agent does not fully instrument at the network level. This limitation motivated subsequent investment in VPC Flow Log and Transit Gateway telemetry integration described in the future directions section.

### *4.4. Honest Account of Failure Cases*

I want to document where the framework fails, because I think this is more useful to other practitioners than headline accuracy numbers alone. Among the incidents where TraceCausalNet did not rank the correct root cause in the top 3, three categories of failure emerged.

In roughly a third of the failure cases, the root cause service had very low call volume during the analysis window either an infrequently-called dependency that becomes critical only under specific load conditions, or a service that had stopped responding entirely, reducing its trace visibility precisely when it was most relevant. The Granger causality tests require sufficient time-series data to produce statistically meaningful results, and sparse trace coverage is a genuine limitation I have not yet solved.

In another third, the root cause was a shared infrastructure component a centralized RabbitMQ message broker or a shared database that appeared as a dependency of many services simultaneously. When such a component fails, it generates near-uniform causal influence scores across all dependent services, producing a diffuse ranking that does not direct the engineer to a useful starting point. I partially address this through infrastructure-node special-casing in the graph construction, but it remains an open problem.

In the remaining failure cases, a deployment occurred within the analysis window and changed a service's behavior fundamentally mid-window, violating the stationarity assumption of the VAR model. These were typically post-deployment incidents where the new behavior was so different from the training distribution that the Granger causality tests produced misleading results. Detecting and handling within-window structural breaks is a methodological problem I discuss further in the future directions.

## 5. Production Deployment and SOC 2 Considerations

### *5.1. Shadow Validation before Activation*

I did not activate TraceCausalNet in production immediately upon development completion. I ran a shadow validation period in which the framework operated in parallel with my existing investigation process generating root cause predictions logged and compared against what the on-call engineers actually identified through their own investigation, without influencing the PagerDuty workflow. During the shadow period, I tracked the framework's predictions systematically against post-mortem ground truth outcomes.

The shadow validation data was then presented to my engineering leadership and compliance teams as the evidence basis for production activation authorization. In my experience in regulated financial environments, this proof-before-activation approach is not optional it is what makes organizational trust possible. Compliance teams in banking organizations are structurally risk-averse toward automated systems that influence operational decisions, and shadow accuracy data gives them the concrete evidence they need to authorize activation.

### *5.2. PagerDuty Workflow Integration*

The production deployment appends TraceCausalNet's diagnostic report to the PagerDuty incident as a structured note before the on-call engineer receives the page. When they acknowledge the PagerDuty alert and open the incident details, the causal diagnostic is already there alongside the Dynatrace problem card link. The report presents the top-3 ranked root cause candidates, each with its causal pathway and supporting Granger causality statistics.

This integration design was intentional. Adding a separate dashboard or tool would require engineers to navigate to it under incident pressure which they generally do not do. Delivering the causal diagnostic through PagerDuty, which they are already using to manage the incident, means it gets read. The friction of adoption was the design constraint I optimized for first, before technical accuracy.

### *5.3. SOC 2 Audit Trail*

For each TraceCausalNet analysis execution, I generate a structured audit record covering: the analysis timestamp and Dynatrace problem card identifier; the complete list of services evaluated with their metric time-series summaries; the Granger causality test statistics and p-values for each evaluated service pair; the GNN causal influence scores; the interventional impact scores driving the final ranking; and the top-5 ranked candidates with causal pathways. This audit record is stored in an immutable append-only log in Amazon S3 with object lock enabled, satisfying the tamper-evidence requirements of SOC 2 audit trail controls.

### *5.4. Human Judgment Remains the Decision Point*

TraceCausalNet supports the on-call engineer's diagnosis; it does not replace their judgment. The framework generates ranked candidates and causal pathway evidence the engineer decides what to do about it. The system takes no automated action: it does not suppress PagerDuty alerts, modify infrastructure, or escalate incidents independently. This boundary was non-negotiable in my deployment, both for compliance reasons and because getting the design wrong in a regulated environment has consequences that take much longer to undo than to prevent.

## 6. Future Directions

Several extensions are planned based on the failure modes and gaps I identified during the production evaluation.

Network telemetry integration is my highest-priority near-term extension. The incidents where TraceCausalNet performed weakest were concentrated in network-originated failures where Dynatrace application-layer traces do not capture the root cause signal. I am currently building an integration that incorporates AWS VPC Flow Logs, Transit Gateway routing telemetry, and Aviatrix policy event data into the attributed graph. In my production environment, which uses Aviatrix for centralized network policy management across a Transit Gateway hub-and-spoke topology connecting multiple client VPCs, network-layer events often precede application-layer anomalies that Dynatrace Davis AI detects providing a prediction window that application-only observability cannot exploit.

Large Language Model integration is the next significant capability extension. TraceCausalNet currently identifies where and why a failure occurred in terms of service topology and metric behavior. It does not translate this diagnosis into specific remediation actions. I am prototyping an LLM agent that takes the TraceCausalNet diagnostic output alongside my post-mortem documentation corpus and recent GitHub Actions deployment history, generating specific remediation suggestions based on historical incidents with similar causal pathways. This work is described in more detail in my companion paper on post-mortem intelligence.

Adaptive analysis window selection addresses the stationarity limitation I described in Section 4.4. The current fixed 30-minute window violates the VAR model's stationarity assumption during rapid infrastructure change precisely the periods of highest failure risk. I plan to integrate a change detection layer that monitors Dynatrace deployment events and GitHub Actions pipeline completions, dynamically shortening the analysis window when a significant structural change is detected within the current window period.

## 7. Conclusion

The 47-minute root cause diagnosis time I documented at the start of this paper stays with me as a concrete number because I know exactly what happened during those 47 minutes: I was investigating a live banking system, working through hypothesis after hypothesis, ruling out services one at a time while the application remained degraded. Dynatrace Davis AI did its job, it detected the problem quickly and got me paged. PagerDuty did its job and I was engaged within seconds of the problem card creation. The gap was in what happened after that. TraceCausalNet is my attempt to close that gap.

I documented an 83% reduction in mean time to root cause, 91% top-3 accuracy across hundreds of real production incidents in live banking applications that are not from a benchmark environment. They are from the same Dynatrace and PagerDuty observability stack used in production operations every day, across the same applications I am responsible for, measured against the same on-call investigations I would have conducted manually. I documented the failure cases honestly because a framework that performs well 91% of the time but fails silently on the other 9% creates its own risks, and practitioners evaluating similar systems deserve to understand where those risks lie.

The deployment methodology I described before activation, PagerDuty workflow integration, SOC 2 audit trail design, and a hard human-in-the-loop boundary is presented as a practical template for SRE practitioners in regulated environments.

The technical accuracy of the framework matters. But the organizational and compliance infrastructure surrounding its deployment is equally determinative of whether it reaches production and whether it actually improves what happens at 2am when a PagerDuty alert fires.

## References

[1] S. Shan, X. Luo, and M. Lyu, "Root cause localization of microservice anomalies using distributed tracing," in Proc. IEEE ISSRE, 2019, pp. 177–188.
[2] M. Ikram, N. Chakraborty, S. Mitra, S. Saini, S. Bagchi, and M. Koperberg, "Root cause analysis of failures in microservices through causal discovery," in Proc. NeurIPS, 2022, pp. 31158–31170.
[3] C. Gu, B. Jing, X. Sun, Z. Yang, and L. Wan, "CausalRCA: Causal inference-based root cause analysis for microservices," in Proc. IEEE ICWS, 2023, pp. 135–144.
[4] L. Wu, J. Du, and J. Wu, "MicroRCA: Root cause localization of performance issues in microservices," in Proc. IEEE/IFIP NOMS, 2020, pp. 1–9.
[5] M. Chen, X. Han, and S. Lu, "HolisticRCA: Holistic root cause analysis for distributed microservice systems," ACM SIGOPS Oper. Syst. Rev., vol. 56, no. 1, pp. 68–75, 2022.
[6] J. Pearl, Causality: Models, Reasoning, and Inference, 2nd ed. Cambridge University Press, 2009.
[7] P. Spirtes, C. Glymour, and R. Scheines, Causation, Prediction, and Search, 2nd ed. MIT Press, 2000.
[8] C. W. J. Granger, "Investigating causal relations by econometric models and cross-spectral methods," Econometrica, vol. 37, no. 3, pp. 424–438, 1969.
[9] B. H. Sigelman et al., "Dapper, a large-scale distributed systems tracing infrastructure," Google Technical Report, 2010.