The Eastasouth Journal of Information System and Computer Science
Vol. 2, No. 02, December 2024, pp. 377-385
ISSN: 3025-566X, DOI: 10.58812/esiscs.v2i02

Graph-Based Dependency Intelligence for Predictive Failure Analysis in Large-Scale Distributed Systems

Shubh Prabhat
Architect, GlobalLogic Inc., Richmond Hill, Ontario, Canada

Article Info

Article history:

Received Dec, 2024
Revised Dec, 2024
Accepted Dec, 2024

Keywords:

Dependency Graph;
Distributed Systems;
Failure Prediction;
Graph Neural Network;
Microservices;
Root Cause Analysis

ABSTRACT

Distributed systems with many services fail via the distributed processes of dependent services, and most monitoring systems do not respond to problems until the first symptom occur. This article introduces a graph-based dependency intelligence architecture, which is used to predict failures that are going to impact users. The framework constructs a weighted service dependency graph using traces, metrics and logs, and trains the graph in each time window, and uses a graph neural network to make a prediction about the likelihood of each service failing within the next fifteen minutes. The edge weight is defined by three equations, the message passing rule and the probability of failure and a fourth equation of the evaluation score. The paper provides a survey of the literature on microservice faults, root cause localization and graph learning, and a comparison of six recent diagnosis studies with the prediction gap. A benchmark of 41 services and 126 dependency edges on a cluster of 12 nodes was utilised in the experiment. The team infused 300 failure instances between four classes over 30 days and evaluated the models on 120 unknown instances. The proposed model achieved precision that was equal to 0.89, a recall of 0.87 and an F1 score that was 0.88. A model that was a static graph model achieved 0.79, service level LSTM reached 0.69 and threshold alerting reached 0.56. The suggested model provided a mean warning lead time of 8.7 minutes, reduced the false alarms to 2.1 per day and the true root service was ranked in top three ranked services in 81 percent of episodes. Findings indicate that the dynamic dependency weights provide predictive power over and above the service level signals.

This is an open access article under the CC BY-SA license.

Corresponding Author:

Name: Shubh Prabhat
Institution: Architect, GlobalLogic Inc., Cary, North Carolina, USA
Email: sprabhat.research@gmail.com

1. INTRODUCTION

Modern services are implementing small components that call each other on a network. This style was used by one of the largest online retailers in Europe to assist more than 500 live deployments each week and maintain a high degree of reliability by a robust automated testing and release pipeline [1]. Each new component introduces an additional dependency and each dependency introduces a route that a failure can follow. One compilation of the microservice journey tracks the history of the style and enumerates the issues that need to be tackled by practitioners [2].

A failure, an error and a fault are the three categories that are separated in the dependability literature. It describes how active fault causes an error and the error propagates through the system until it is provided by a service [3]. When the latency or the error rates exceed a certain limit, they give a signal, and engineers need to ascertain the cause in the dependency graph. The time between the first weak signal and the outage by the users is wasted by this order. That time can be used in a model that reads the dependency structure. The value is realistic. A pre-emptive alert that identifies the probable faulty service provides the team owning service with minutes to scale capacity, redistribute traffic or break down a component before a customer is aware of an issue.

In this paper, there are three objectives. The first one is to develop a framework converting service dependencies into a weighted graph which is traffic-dependent. The second is to test how a graph neural network on a graph predicts failures earlier and more accurately than techniques that do not consider dependencies. The third one is to provide operations teams with clear actions based on the evidence.

2. LITERATURE REVIEW

2.1 *Microservice Faults and Dependency Structure*

The authors used an industrial survey of faults of microservices to construct a benchmark consisting of 41 services and to run the replications of 22 actual cases of faults on this benchmark. The survey discovered that the practice of debugging had been limited at the time but it required more intelligent trace analysis [4]. The experiment presented in the paper is based on the same benchmark. Findings about service dependency graphs indicate that algorithms in a graph that use a graph database can highlight problematic nodes and other bad design patterns before they cause a crash [5]. In work on the origins of microservices, they are treated as small services that a team of developers and operators develops and deploys, this is why multiple teams and several owners have many dependencies [6]. There is an agreement on these studies. Call structure between services determines the distance that a fault is going to follow, and teams do not often witness that structure at the point a fault originates.

2.2 *Root Cause Localization and Failure Diagnosis*

The current researches are aimed at identifying the cause following the occurrence of an anomaly. Table 1 is a comparison of six of them and indicates the gap that this paper is filling. There are studies that model anomaly spread on service graphs and those that integrate traces, logs and metrics with graph neural networks. Each of the studies in the table happens when a symptom has occurred. None of them predicts the likelihood of a future failure of a window, and the breach is the contribution made by the current work. The results of the accuracy presented in Table 1 are very high, and this indicates that the structure of the graphs assists in diagnosis. Prediction should also be assisted by the same structure, as a weak signal of one service is likely to be evident among its neighbors and the service to fail before it.

Table 1. Six published diagnosis studies have been compared, their results reported and the gap in predictive dependency analysis they leave

Study focus	Main contribution	Gap for prediction	Source
Attributed graph localization	Models anomaly disperses among services and machines and has 89 percent of precision and 97 percent of mean average precision.	Starts after the anomaly appears	[7]
Trace analysis	Prioritizes the ranks by the percentage of abnormal traces through them.	Traces of a fault already in existence	[8]

Study focus	Main contribution	Gap for prediction	Source
Service call graph analysis	Breaks propagation chains, achieves top 3 hits ratio of 68 percent in production and reduces production search time by half (30 to 5 minutes)	Availability of targets following a report	[9]
Multimodal graph diagnosis	Identifies joins, logs and metrics in a dependency graph, and increases accuracy of root causes from 20.9 to 368 percent	Diagnoses that have already taken place	[10]
Joint detection and localization	Acknowledges behavior within services and dependency between services, where the highest hit rate improvements are 290-5068 percent across five baselines.	Has no future horizon	[11]
Edge attention graph network	Added edge features to an attention graph model and achieved 87 percent on average top one correctness	Finds reasons and does not predict	[12]

2.3 Graph Learning for Service Systems

Graph learning provides the means to read dependency structure. Geometric deep learning groups other non Euclidean data over graphs and regular grids on which neural networks are applied [13]. Service systems would suit this perspective since nodes are represented by services and edges are by calls. The practical challenge is to maintain the graph up-to-date as the traffic evolves. A fixed graph constructed initially with deployment files overlooks the fact that traffic moves between routes on a daily basis. A load, which during the night will be of a momentary nature, may be the most prolonged in the morning, and the load in the model will take the character of the morning.

3. DEPENDENCY INTELLIGENCE FRAMEWORK

In Figure 1, the framework consists of four layers. The collector collects traces, metrics and logs. The graph layer is used to create a dependency graph of the service and update it in each window. The graph neural network is performed on the graph by the learning layer. The action layer prioritises the services based on their risk of failure and provides early alerts to the service owners. Each warning will be verified or rejected by operators, and this feedback will be stored in both the operator's framework to be used in the following training round.

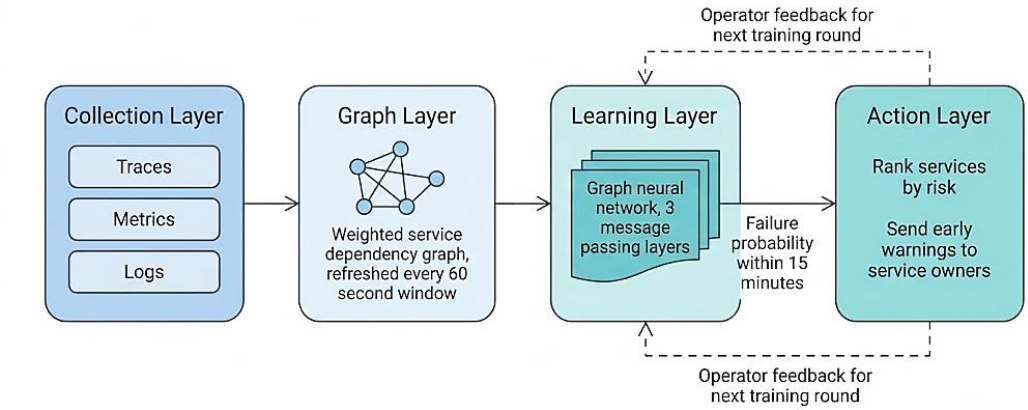


Figure 1. Diagram depicting the collection layer, graph construction layer, learning/action layer and such a feedback path of operators

3.1 Graph Construction

A node is formed by each service and each call observed by an edge that is directed. The node and edge features of this study are listed in Table 2. The dynamic network research considers an evolving system, as a series of snapshots of a graph [14]. The framework adheres

to that concept and constructs a single snapshot (60 second window). The edge weight is an addition of three signals as indicated by Equation (1).

Table 2. The features of the nodes and edges within each window, the source of the data and the predictive role of the feature

Element	Feature	Source	Prediction role
Node	CPU and memory use	Metrics	Resource pressure
Node	95th percentile latency and error rate	Traces and metrics	Service health
Node	Restart count and log error rate	Logs	Instability
Edge	Call frequency	Traces	Coupling strength
Edge	Edge latency	Traces	Delay propagation
Edge	Edge error rate	Traces	Fault transmission

$$w_{ij} = \alpha C_{ij} + \beta L_{ij} + \gamma E_{ij}, \quad with\ \alpha + \beta + \gamma = 1 \#(1)$$

In Equation (1), C_{ij} is the normalized rate of a call between service i and service j, L_{ij} is the normalized edge latency and E_{ij} is the edge error rate. The experiment employed $\alpha = 0.3$, $\beta = 0.3$ and $\gamma = 0.4$ since errors are the direct transmission of faults.

4. PREDICTIVE MODEL

Learning layer is based on the message passing perspective of graph neural networks, according to which each node acts on the information of its neighbors to update its state. A field survey clusters these networks into recurrent and convolutional families and autoencoders as well as spatial temporal families [15]. The workflow between raw telemetry to ranked warnings is depicted in Figure 2. The rule of update is given by equation (2).

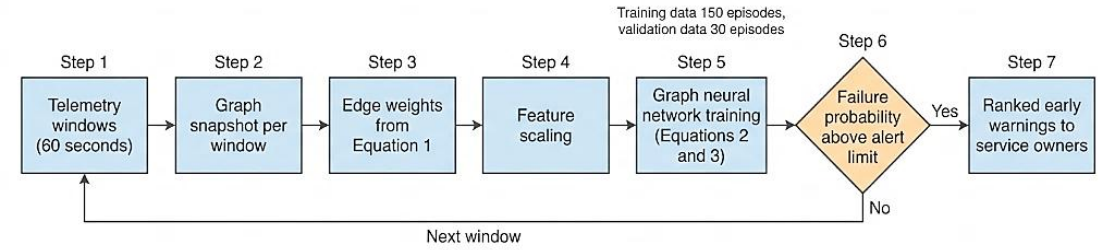


Figure 2. Telemetry window to graph snapshot workflow, feature scaling, neural network training, risk scoring and ranked early warning

$$h_i^{(l+1)} = \sigma\left(\sum_{j \in N(i)} a_{ij} W^{(l)} h_j^{(l)}\right), \quad with\ a_{ij} = \frac{w_{ij}}{\sum_k w_{ik}} \#(2)$$

In the equation (2), h_i denotes state of service i to the layer l, N(i) refers to the group of its neighbors, W denotes a learnt weighting matrices and σ is the activation function. Making the counts of the weights normal makes one dependency look busier than others. The model has three layers and so, each node communicates to the services, which are not more than three calls away. This is the majority of cascade paths in the test bed. A more refined model would be a combination of signals across services that are not strongly correlated, and would increase the price of inference. Equation (3) functions to transform the final state to a failure probability.

$$p_i = \sigma\left(u^T h_i^{(L)} + b\right) \#(3)$$

In Equation (3), p_i is the likelihood of the service i to fail in the next 15 minutes, u is a weight and b is bias. A window is labeled with one by the occurrence of a failure episode at any time, in less than 15 minutes. The loss function is a costly cross entropy that deals with the minimal portion of the positive windows. All four of the classes of failures

and the graph signal corresponding to each are listed in Table 3.

Table 3. Four classes of failure, how each failure was injected, the graph signal it gives and the common propagation mode

Failure class	Injection method	Graph signal	Propagation pattern
CPU saturation	Stress process within a container	Increasing latency of the node and callers	Moves upstream
Memory leak	Gradual memory allocation	Slow memory increase with restart	Dependent services are affected by restarts
Network delay	Shaping of traffic between pods	Increased latency at one edge	Caller Timeout
Cascading timeout	Slowness of a downstream database service	Increasing error rate in a chain	Retries amplify load

5. EXPERIMENTAL SETUP

The experiment was run on the 41 service graphon a Kubernetes cluster of 12 worker nodes. The dependency topology presented in Figure 3 has five call tiers and 126 edges. The entire configuration is given in Table 4.

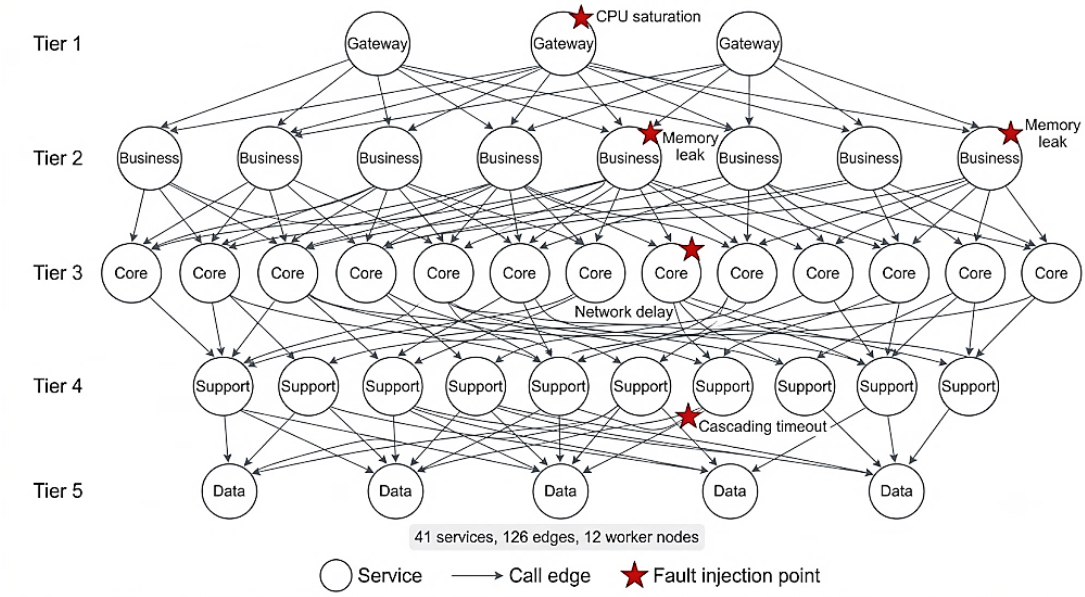


Figure 3. Dependency topology of the test bed consisting of 41 services, 126 call edges and marked fault injection points

Guidance of cloud native applications benchmarking includes controlled load and fixed resource configurations; therefore, the load generator maintained the constant load rates and node sizes to generate traffic across cloud computing runs [16]. The number of failure episodes that was injected by a chaos tool was 300 during 30 days. The training set consisted of the first 150 episodes in the first 15 days. The day following had 30 episodes and was the validation set. The 120 episodes were in the test set, which comprised 30 episodes in each class, which was the last 12 days.

Table 4. Configuration used in experimentation that includes the test bed, telemetry, fault injection, data split, baseline techniques and metrics used

Component	Configuration	Purpose
Test bed	41 services, 126 edges, 12 worker nodes	Realistic dependency depth
Telemetry	Traces, metrics and logs in 60 second windows	Multimodal input
Fault injection	300 episodes in four classes over 30 days	Labeled ground truth
Data split	15, 3 and 12 days with 150, 30 and 120 episodes	Time ordered evaluation
Baselines	Threshold alerting, service level LSTM, static graph model	Reference points
Measures	Precision, recall, F1, lead time, false alarms, top three hit rate, inference time	Accuracy and cost

The duration of each episode of failures was 4-9 minutes, and the injection tool remembered the exact time each episode started, resulting in all the windows being trustworthy in the labeling. The experiment was repeated three times using different random seeds and records the average of the three experiments. There were three baselines. Threshold alarm triggers at a set limit when the latency or errors are in excess of the set limit. The service level LSTM reads each service alone. The static graph model is employed to use a fixed graph and the weight of the edges is the same. The most scalable inference time was discovered to be logistic regression when comparing the classical and deep learners in predicting cloud failures and thus inference time is mentioned before accuracy [17].

6. RESULTS

The F1 score to compare the models is stated in equation (4).

$$F1 = \frac{2PR}{P+R} \#(4)$$

P is the precision and R is the recall (in Equation (4)) and they are calculated per episode. All measures on 120 test episodes, are recorded in Table 5. The model proposed attained F1 score of 0.88. This compares with 0.09 higher than the static graph model, 0.19 higher than service level LSTM and 0.32 higher than threshold alerting. An inter-episode paired test presented p less than 0.01 to the variance between the suggested model and the graph non dynamic model.

Table 5. Accuracy, lead time, false alarm, root cause ranking and cost results of four methods on 120 test cases of episodes

Measure	Threshold alerting	Service level LSTM	Static graph model	Proposed model
Precision	0.58	0.71	0.80	0.89
Recall	0.55	0.68	0.78	0.87
F1 score	0.56	0.69	0.79	0.88
Mean lead time (minutes)	2.1	4.3	6.0	8.7
False alarms per day	9.4	5.8	3.9	2.1
Top three root cause hit rate	0.41	0.52	0.68	0.81
Inference time per window (ms)	4	18	31	38

Mean lead time increased to 8.7 minutes as compared to 6.0, 4.3 and 2.1 minutes of the other methods. False alarms decreased to 2.1 days per day, which is 46 percent lower than the graph model that exhibited no movement. The inference time of 38 milliseconds remains small when compared to a window size of 60 seconds, hence the model will be able to score all 41 services within each window with no additional hardware. The highest overall percentage of 0.81 that engineers were given the true root service in a short-ranked list indicates that this occurred four out of five episodes. The F1 score of each type of failure is presented in Figure 4.

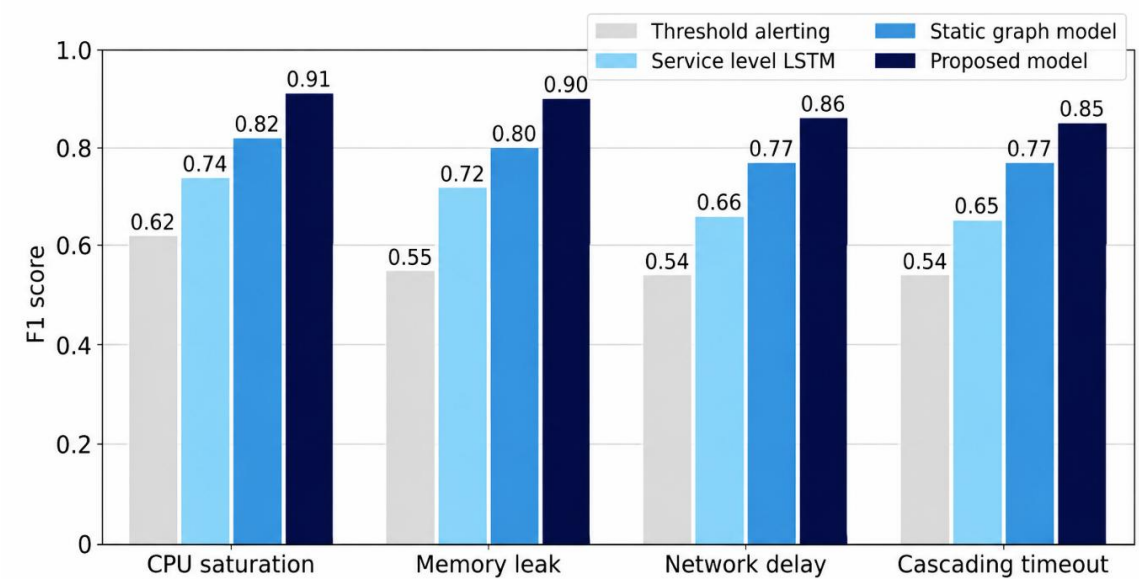


Figure 4. Failure class associated F1 scores of the four methods, with the proposed model highest in all classes

The proposed model is closest to the static graph model of gain and is used to describe the contribution of graph structure to much of the gain with the rest attributed to dynamic weights. The proposed model has the highest score in each of the classes. Its F1 score of CPU saturation, memory leak, network delay and cascading time were 0.91, 0.90, and 0.86, respectively. These statistics reveal that those faults that propagate over many hops take longer to time, as indicated by the lower scores of the latter two classes.

7. DISCUSSION

The findings concur with the previous studies in three aspects. To begin with, an analysis of cascading instances of cloud failures reveals that failures propagate over dependency graphs and learning to graphically represent them enables the grouping of cascading incidents [18]. Second, an invocation trace study with grouping of invocation on service pair, training the context with recurrent and attention layers reveals the importance of context between the services [19]. Third, large scale trace anomaly detection stresses speed and stability with change as systems grow [20]. All these three points reflect the idea that the gain is not as a result of single design choice. It consists of a dependency structure plus weights that derive as a result of traffic and a prediction target that is forward-looking.

There are limitations of the study. This experiment involved a single benchmark, faults were injected, and types of failures which are not covered by any injection tool are also found in production traffic. The edge weights of Equation (1) were the results of judgment and need tuning of field data. The large changes in the architecture of the architecture also require retraining of the model. Teams are thus expected to monitor the decrease or increase of the edge weights and to retrain itself when the call pattern of a service in question changes radically.

8. CONCLUSION

The current paper introduced a dependency intelligence model that constructs an evolving graph of services, utilizes a graph neural network and prioritizes services based on their risk of failure. The F1 score of 0.88, mean lead time of 8.7 minutes and 2.1 false alarms/day recorded by the experiment were better than the three baselines. The number of times that the static graph model achieved a 0.79 score indicates that the portion of improvement was a measurable portion of the gain and is not due solely to the edge weights in the dynamic model. The framework allows teams to refresh the graph in each window and deliver

prioritized warnings to the service owners. The framework needs to be tested with production traces and the frequency of retraining of the model should be tested in the future.

REFERENCES

[1] W. Hasselbring and G. Steinacker, "Microservice Architectures for Scalability, Agility and Reliability in E-Commerce," in *2017 IEEE International Conference on Software Architecture Workshops (ICSAW)*, 2017, pp. 243–246. doi: 10.1109/ICSAW.2017.11.

[2] P. Jamshidi, C. Pahl, N. C. Mendonça, J. Lewis, and S. Tilkov, "Microservices: The Journey So Far and Challenges Ahead," *IEEE Softw.*, vol. 35, no. 3, pp. 24–35, 2018, doi: 10.1109/MS.2018.2141039.

[3] A. Avizienis, J.-C. Laprie, B. Randell, and C. Landwehr, "Basic concepts and taxonomy of dependable and secure computing," *IEEE Trans. Dependable Secur. Comput.*, vol. 1, no. 1, pp. 11–33, 2004, doi: 10.1109/TDSC.2004.2.

[4] X. Zhou *et al.*, "Fault Analysis and Debugging of Microservice Systems: Industrial Survey, Benchmark System, and Empirical Study," *IEEE Trans. Softw. Eng.*, vol. 47, no. 2, pp. 243–260, 2021, doi: 10.1109/TSE.2018.2887384.

[5] E. Gaidels and M. Kirikova, "Service Dependency Graph Analysis in Microservice Architecture," in *Perspectives in Business Informatics Research*, R. A. Buchmann, A. Polini, B. Johansson, and D. Karagiannis, Eds., Cham: Springer International Publishing, 2020, pp. 128–139. doi: https://doi.org/10.1007/978-3-030-61140-8_9.

[6] N. Dragoni *et al.*, "Microservices: Yesterday, Today, and Tomorrow," in *Present and Ulterior Software Engineering*, M. Mazzara and B. Meyer, Eds., Cham: Springer International Publishing, 2017, pp. 195–216. doi: 10.1007/978-3-319-67425-4_12.

[7] L. Wu, J. Tordsson, E. Elmroth, and O. Kao, "MicroRCA: Root Cause Localization of Performance Issues in Microservices," in *NOMS 2020 - 2020 IEEE/IFIP Network Operations and Management Symposium*, IEEE Press, 2020, pp. 1–9. doi: 10.1109/NOMS47738.2020.9110353.

[8] Z. Li *et al.*, "Practical Root Cause Localization for Microservice Systems via Trace Analysis," in *2021 IEEE/ACM 29th International Symposium on Quality of Service (IWQOS)*, 2021, pp. 1–10. doi: 10.1109/IWQOS52092.2021.9521340.

[9] D. Liu *et al.*, "MicroHECL: High-Efficient Root Cause Localization in Large-Scale Microservice Systems," in *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, 2021, pp. 338–347. doi: 10.1109/ICSE-SEIP52600.2021.00043.

[10] S. Zhang *et al.*, "Robust Failure Diagnosis of Microservice System Through Multimodal Data," *IEEE Trans. Serv. Comput.*, vol. 16, no. 6, pp. 3851–3864, 2023, doi: 10.1109/TSC.2023.3290018.

[11] C. Lee, T. Yang, Z. Chen, Y. Su, and M. R. Lyu, "Eadro: An End-to-End Troubleshooting Framework for Microservices on Multi-source Data," in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, 2023, pp. 1750–1762. doi: 10.1109/ICSE48619.2023.00150.

[12] R. Chen, J. Ren, L. Wang, Y. Pu, K. Yang, and W. Wu, "MicroEGRCL: An Edge-Attention-Based Graph Neural Network Approach for Root Cause Localization in Microservice Systems," in *Service-Oriented Computing*, J. Troya, B. Medjahed, M. Piattini, L. Yao, P. Fernández, and A. Ruiz-Cortés, Eds., Cham: Springer Nature Switzerland, 2022, pp. 264–272. doi: https://doi.org/10.1007/978-3-031-20984-0_18.

[13] M. M. Bronstein, J. Bruna, Y. LeCun, A. Szlam, and P. Vandergheynst, "Geometric Deep Learning: Going beyond Euclidean data," *IEEE Signal Process. Mag.*, vol. 34, no. 4, pp. 18–42, 2017, doi: 10.1109/MSP.2017.2693418.

[14] J. Skarding, B. Gabrys, and K. Musial, "Foundations and Modeling of Dynamic Networks Using Dynamic Graph Neural Networks: A Survey," *IEEE Access*, vol. 9, pp. 79143–79168, 2021, doi: 10.1109/ACCESS.2021.3082932.

[15] Z. Wu, S. Pan, F. Chen, G. Long, C. Zhang, and P. S. Yu, "A Comprehensive Survey on Graph Neural Networks," *IEEE Trans. Neural Networks Learn. Syst.*, vol. 32, no. 1, pp. 4–24, 2021, doi: 10.1109/TNNLS.2020.2978386.

[16] S. Henning and W. Hasselbring, "A configurable method for benchmarking scalability of cloud-native applications," *Empir. Softw. Eng.*, vol. 27, no. 6, p. 143, 2022, doi: 10.1007/s10664-022-10162-1.

[17] T. N. Tengku Asmawi, A. Ismail, and J. Shen, "Cloud failure prediction based on traditional machine learning and deep learning," *J. Cloud Comput.*, vol. 11, no. 1, p. 47, 2022, doi: 10.1186/s13677-022-00327-0.

[18] Z. Chen *et al.*, "Graph-based Incident Aggregation for Large-Scale Online Service Systems," in *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2021, pp. 430–442. doi: 10.1109/ASE51524.2021.9678746.

[19] Q. Du, L. Zhao, F. Tian, and Y. Han, "Trace-Based Anomaly Detection with Contextual Sequential Invocations," in *Database and Expert Systems Applications*, C. Strauss, T. Amagasa, G. Kotsis, A. M. Tjoa, and I. Khalil, Eds., Cham: Springer Nature Switzerland, 2023, pp. 113–118. doi: https://doi.org/10.1007/978-3-031-39821-6_8.

[20] S. Zhang *et al.*, "Efficient and Robust Trace Anomaly Detection for Large-Scale Microservice Systems," in *2023 IEEE 34th International Symposium on Software Reliability Engineering (ISSRE)*, 2023, pp. 69–79. doi: 10.1109/ISSRE59848.2023.00012.

BIOGRAPHIES OF AUTHORS



Shubh Prabhat — Researcher Biography

Shubh Prabhat is a Software Engineering Leader, Solution Architect, and Independent Researcher with over 18 years of experience in distributed systems, cloud computing, cybersecurity, and AI-driven enterprise solutions.

He earned his **Bachelor of Technology (B.Tech.) in Electronics and Communication Engineering in 2008**.

His research interests include Artificial Intelligence, Machine Learning, Cloud Security, Distributed Systems, Identity and Access Management, and Cybersecurity.

Research Profiles:

- **ORCID:** https://orcid.org/0009-0008-1716-8789
- **Google Scholar:** https://scholar.google.com/citations?user=HrEzOfIAAAAJ
- **Email:** sprabhat.research@gmail.com